webix

Webix UI Documentation

1 Overview
2 Getting Started
    2.1 Quick Start Guide
        2.1.1 Introduction to Webix
    2.2 Start Coding with Webix
        2.2.1 Debug Version of the Library
    2.3 Creating Basic Apps
        2.3.1 Setting Components
        2.3.2 Interaction Between App Elements
        2.3.3 Server-Side Integration
    2.4 Introduction to Events
    2.5 Webix Docs Guide
3 Compatibility
4 What's new
    4.1 Version 2.4
    4.2 Version 2.3
    4.3 Version 2.2
    4.4 Version 2.1
    4.5 Version 2.0
    4.6 Version 1.10
    4.7 Version 1.9
    4.8 Version 1.8
    4.9 Version 1.7
    4.10 Version 1.6
    4.11 Version 1.5
    4.12 Version 1.4
    4.13 Version 1.3
    4.14 Version 1.2
    4.15 Version 1.1
    4.16 Version 1.0.2
    4.17 Version 1.0.1
5 UI Components
    5.1 Accordion
        5.1.1 Accordionitem
    5.2 Barcode
    5.3 Calendar
    5.4 Carousel
    5.5 Chart
        5.5.1 Chart Types
            5.5.1.1 Settings for Pie Charts
            5.5.1.2 Settings for Bar Charts
                5.5.1.2.1 Color Effects
            5.5.1.3 Setting for Line Charts
            5.5.1.4 Settings for Area Charts
            5.5.1.5 Settings for Radar Chart
            5.5.1.6 Settings for Scatter Chart
        5.5.2 Setting Chart Scales
            5.5.2.1 Custom X-Axis Formation
        5.5.3 Chart Series
        5.5.4 Operations with Chart Data
        5.5.5 Setting Legend for a Chart
        5.5.6 Chart Tooltip
    5.6 Colorboard
    5.7 Context
    5.8 ContextMenu
    5.9 DataTable
        5.9.1 Overview
        5.9.2 Getting started

5.9.3 Quick start with DataTable
5.9.4 Getting/setting the cell value
5.9.5 Configuration
5.9.5.1 Columns Configuration
5.9.5.2 Creating Index Columns
5.9.5.3 Headers and Footers
5.9.5.3.1 Datatable Header Menu
5.9.5.4 Datatable Rowspans and Colspans
5.9.5.5 Data loading
5.9.5.5.1 Examples of data formats
5.9.5.5.2 Data mapping
5.9.5.6 Sizing and Resizing
5.9.5.7 Internationalization
5.9.5.8 External configuration
5.9.6 Look and Feel
5.9.6.1 Cells Templates
5.9.6.1.1 Templates.Syntax
5.9.6.2 Number and Date Formatting
5.9.6.3 Styling
5.9.6.4 Split Mode or 'Frozen Columns'
5.9.7 Operations with data
5.9.7.1 Copy-pasting data
5.9.7.2 DataTable Filtering
5.9.7.3 Math Operations
5.9.7.4 Paging with Datatable
5.9.7.5 Selection with Datatable
5.9.7.6 DataTable Sorting
5.9.7.7 DataTable Editing
5.9.7.8 Checkbox and Radio in DataTable
5.9.8 Integration
5.9.9 Keyboard navigation
5.9.10 Drag-and-drop with Datatable
5.9.11 Data export to PDF, Excel
5.9.12 Events Handling with DataTable
5.10 Dataview
5.11 Form
5.12 Form Controls
5.12.1 Manipulating Groups of Buttons
5.12.2 Suggest List
5.12.3 Dynamic Tabbar
5.12.4 Advanced Configuration of Popup Selectors
5.13 Grouplist
5.14 HTMLform
5.15 IFrame
5.16 Layout and Resizer
5.17 List
5.17.1 X-list
5.18 Menu
5.19 Multiview
5.19.1 Controls to Switch Between View Cells
5.19.2 Common Functions to Switch Between Views
5.20 Organogram
5.20.1 Initializing Organogram on a page
5.20.2 Loading Data
5.20.3 Selecting Items
5.20.4 Customizing Organogram
5.20.5 Styling
5.20.6 Event Handling

5.20.7 General How-Tos
5.21 Popup
5.22 Portlet
5.23 Property Sheet
5.24 Scrollview
5.25 Tabview
5.26 Template
5.27 Toolbar
5.28 Tree
5.28.1 Quick start with Tree
5.28.2 Getting/setting the node
5.28.3 Loading data
5.28.3.1 Details and examples of data formats
5.28.4 Tree Nodes manipulations
5.28.5 Items Selection
5.28.6 Node templates
5.28.7 Checkboxes in the tree
5.28.8 Sorting in the Tree
5.28.9 Filtering in the Tree
5.28.10 Editing in the Tree
5.28.11 Copy-pasting data
5.28.12 Styling
5.28.13 Drag-and-drop support
5.28.14 Event handling with Tree
5.29 Treetable
5.30 Unitlist
5.31 Uploader
5.32 Window
5.32.1 Popup and Window Positioning
5.33 Video
6 UI Extensions
6.1 Pivot Table
6.2 Pivot Chart
6.3 Dataview Suggest
6.4 Grid Suggest
6.5 Text Editor Integration
6.6 Chart Integration
6.7 Maps Integration
6.8 Scheduler and Gantt Integration
6.9 Mobile Scheduler
6.9.1 Creating Scheduler
6.9.2 Loading and Saving Data
6.9.3 Customizing the "Edit" Form
6.9.4 Specifying Templates
6.9.5 Sizing Elements
6.9.6 Date Formats
6.9.7 Styling Scheduler
6.9.8 Localizing Scheduler
6.9.9 Read-only Mode
6.9.10 Recurring Events
6.9.11 Dynamic Loading
6.9.12 Changing Scheduler's Structure
6.9.13 Handling Events
6.9.14 General How-Tos
6.9.15 Tips for dhtmlxScheduler's users
6.10 Kanban Board
6.10.1 Creating Kanban Board on a page
6.10.2 Loading and Saving Data

6.10.3 Data Visualization
6.10.4 Customizing Kanban's structure
6.10.5 Adding and Deleting Items
6.10.6 Working with Data
6.10.7 Editing Items
6.10.8 Handling Events
6.10.9 API Reference
6.10.10 General How-Tos
6.11 File Manager
6.11.1 Creating File Manager on a page
6.11.2 Loading and Saving Data
6.11.3 Data Visualization
6.11.4 File Manager Structure
6.11.5 Localizing File Manager
6.11.6 Manipulating Items
6.11.7 Handling Events
6.11.8 API Reference
6.11.9 General How-Tos
7 Guides
7.1 Designing User Interface
7.1.1 Building App Architecture
7.1.1.1 View
7.1.1.2 Responsive Layouts
7.1.2 Structuring a Big Application
7.1.3 Initialization from Markup
7.1.3.1 Initialization from HTML Markup
7.1.3.2 Initialization from XML Markup
7.2 Webix in Touch Environment
7.3 Loading and Storing Data
7.3.1 Data Loading
7.3.1.1 Supported Data Types
7.3.1.1.1 Customizing Data Types
7.3.1.2 Dynamic Loading
7.3.1.2.1 Dynamic Loading for Plain Structures
7.3.1.2.2 Dynamic Loading for Hierarchical Data Structures
7.3.1.3 Management of loading errors
7.3.2 Defining Data Scheme
7.3.3 Server-side Integration
7.3.3.1 Custom Server Scripts
7.3.3.2 REST Mode Support
7.3.3.3 Saving Data with Webix Ajax Helper
7.3.3.4 Customizing Loading Pattern. Ajax Loading and Parsing
7.3.3.5 Offline Support
7.3.3.6 Webix Proxy Objects
7.3.3.7 Working with IndexedDB
7.3.3.8 Websockets (Faye)
7.3.3.9 DataProcessor
7.3.3.10 Data Connector
7.3.3.11 Cross-Domain Data Loading
7.3.4 Data Binding and Syncing
7.3.4.1 Data Binding and Syncing with Non-UI Objects
7.3.4.2 Working with Bound Components
7.3.5 Data Updating
7.4 Configuring Components
7.4.1 Managing Visibility of Components
7.4.2 Disabling/Enabling Components
7.4.3 Adding Headers to Components
7.4.4 Dynamic UI Modifications

7.4.5 Adding and Deleting Multiview Cells Dynamically
7.4.6 Changing Properties of Components
7.4.7 Sizing components
7.4.8 Resizing
7.4.9 Scroll Control
7.4.10 Date and Number Localization
7.5 Managing Data in a Component
7.5.1 Data Object Operations
7.5.2 Selection
7.5.3 Adding/Deleting Data Items
7.5.4 Data Templates
7.5.4.1 Tooltip Implementation
7.5.4.2 Type Implementation
7.5.5 Working with Dates
7.5.6 Data Filtering and Sorting
7.5.7 Data Validation
7.5.8 Editing Data Items
7.5.9 Data editors
7.5.10 Paging
7.5.11 Drag-n-Drop within Webix Views
7.5.11.1 Defining Drag Area
7.5.11.2 Defining Drag Marker
7.5.11.3 Drag-and-Drop with Tree and TreeTable
7.5.11.4 Drag-and-Drop between Different Components
7.5.11.5 Drag-and-Drop Events
7.5.11.6 HTML5 DnD with Webix Views
7.5.12 Working with the clipboard
7.5.13 Grouping Data Items
7.5.13.1 Custom Functor for Data Grouping
7.6 Styling and Animation
7.6.1 Skins
7.6.1.1 Material Design Skin
7.6.2 Component Styling
7.6.2.1 CSS image maps
7.6.2.1.1 Accordion
7.6.2.1.2 Calendar
7.6.2.1.3 Carousel
7.6.2.1.4 Chart
7.6.2.1.5 Colorboard
7.6.2.1.6 Colorpicker
7.6.2.1.7 Datepicker
7.6.2.1.8 Datatable
7.6.2.1.9 Tree
7.6.2.1.10 Dataview
7.6.2.1.11 Groulist
7.6.2.1.12 Layout
7.6.2.1.13 List
7.6.2.1.14 Menu
7.6.2.1.15 Message Box
7.6.2.1.16 Popup
7.6.2.1.17 Property Sheet
7.6.2.1.18 Toolbar
7.6.2.1.19 Treetable
7.6.2.1.20 Unitlist
7.6.2.1.21 Window
7.6.2.1.22 XList
7.6.2.1.23 Uploader
7.6.2.1.24 Button

7.6.2.1.25 Checkbox
7.6.2.1.26 Combo
7.6.2.1.27 Counter
7.6.2.1.28 Label
7.6.2.1.29 Radio
7.6.2.1.30 Select
7.6.2.1.31 Segmented
7.6.2.1.32 Tabbar
7.6.2.1.33 Richselect
7.6.2.1.34 Text
7.6.2.1.35 Textarea
7.6.2.1.36 Toggle
7.6.3 Animation
7.6.4 Icons with UI Components
7.6.5 Setting Borders to the Components
7.7 Interacting with Users
7.7.1 Creating Message Boxes
7.7.2 Setting Progress Bar and Overlay Messages
7.7.3 HTMLForm and Form Treatment
7.7.4 Uploading Files
7.7.4.1 Configuring Uploader
7.7.4.2 Managing Server Part of Uploading
7.7.4.3 Progress Bar for Uploader
7.7.4.4 Binding Uploader to Any Page Element
7.7.5 Uploader and Form Integration
7.7.6 Controls Guide
7.7.6.1 Hidden Input Fields
7.7.6.2 Common Configuration for Form and Toolbar Controls
7.7.7 Event Handling
7.7.8 Keyboard Events. Focusing. Navigation
7.7.9 Adding Active Elements to Components
7.8 Web and Internet
7.8.1 Webix Storage Interface
7.8.2 Saving Application State
7.8.2.1 Saving and Restoring Application State
7.8.2.2 Saving and Restoring Component State
7.8.2.3 Tracking the Latest Opened Tab
7.9 Extending components
7.9.1 Component Copying and Extending
7.9.2 Creating a Custom Component
7.10 Third party integration
7.10.1 JQuery Integration
7.10.2 Angular JS Integration
7.10.2.1 Angular Controllers and Data Loading
7.10.2.2 Event Handling With Angular
7.10.2.3 Changing View Visibility
7.10.3 Backbone Integration
7.10.3.1 Backbone Collections
7.10.3.2 Backbone Views
7.10.3.3 Working with Backbone Routers
7.10.4 Integration with DHTMLX Components
7.10.5 DIY - Custom Integration Pattern
7.11 Auxiliary Resources
7.11.1 Top 10 Helpers You Should Know About
7.11.2 Ajax operations
7.11.3 Advanced Drag-and-Drop
7.11.4 Data drivers
7.11.5 Number formatting methods

7.11.6 Date formatting methods
7.11.7 Ui.resizearea. Common tips
8 API Reference
8.1 common helpers
8.1.1 Methods
8.1.1.1 alert
8.1.1.2 animate
8.1.1.3 attachEvent
8.1.1.4 bind
8.1.1.5 blockEvent
8.1.1.6 callEvent
8.1.1.7 clone
8.1.1.8 confirm
8.1.1.9 copy
8.1.1.10 delay
8.1.1.11 detachEvent
8.1.1.12 dp
8.1.1.13 editStop
8.1.1.14 event
8.1.1.15 eventRemove
8.1.1.16 exec
8.1.1.17 extend
8.1.1.18 hasEvent
8.1.1.19 isArray
8.1.1.20 isDate
8.1.1.21 isUndefined
8.1.1.22 jsonp
8.1.1.23 mapEvent
8.1.1.24 message
8.1.1.25 modalbox
8.1.1.26 once
8.1.1.27 proto
8.1.1.28 protoUI
8.1.1.29 proxy
8.1.1.30 ready
8.1.1.31 remote
8.1.1.32 require
8.1.1.33 send
8.1.1.34 single
8.1.1.35 template
8.1.1.36 toArray
8.1.1.37 toFunctor
8.1.1.38 toNode
8.1.1.39 type
8.1.1.40 ui
8.1.1.41 uid
8.1.1.42 unblockEvent
8.1.1.43 wrap
8.1.2 Events
8.1.2.1 onClick
8.1.2.2 onRotate
8.1.3 Others
8.1.3.1 codebase
8.1.3.2 name
8.1.3.3 version
8.1.3.4 scrollSize
8.1.3.5 zIndexBase
8.2 ui helpers

8.2.1 Methods
8.2.1.1 delay
8.2.1.2 fullScreen
8.2.1.3 hasMethod
8.2.1.4 resize
8.2.1.5 zIndex
8.2.2 Others
8.3 ajax
8.3.1 Methods
8.3.1.1 bind
8.3.1.2 del
8.3.1.3 get
8.3.1.4 getXHR
8.3.1.5 headers
8.3.1.6 post
8.3.1.7 put
8.3.1.8 response
8.3.1.9 stringify
8.3.1.10 sync
8.3.2 Others
8.3.2.1 master
8.4 clipbuffer
8.4.1 Methods
8.4.1.1 destructor
8.4.1.2 focus
8.4.1.3 init
8.4.1.4 set
8.5 color
8.5.1 Methods
8.5.1.1 hexToDec
8.5.1.2 hsvToRgb
8.5.1.3 rgbToHsv
8.5.1.4 toHex
8.5.1.5 toRgb
8.6 csv
8.6.1 Methods
8.6.1.1 parse
8.6.1.2 stringify
8.6.2 Others
8.6.2.1 delimiter
8.6.2.2 escape
8.7 editors
8.7.1 Others
8.7.1.1 $popup
8.7.1.2 checkbox
8.7.1.3 color
8.7.1.4 combo
8.7.1.5 date
8.7.1.6 inline-checkbox
8.7.1.7 inline-text
8.7.1.8 multiselect
8.7.1.9 password
8.7.1.10 popup
8.7.1.11 richselect
8.7.1.12 select
8.7.1.13 text
8.8 env
8.8.1 Others

8.8.1.1 cssPrefix
8.8.1.2 isFF
8.8.1.3 isIE
8.8.1.4 isSafari
8.8.1.5 isWebKit
8.8.1.6 jsPrefix
8.8.1.7 mouse
8.8.1.8 strict
8.8.1.9 svg
8.8.1.10 transform
8.8.1.11 transition
8.8.1.12 transitionDuration
8.8.1.13 transitionEnd
8.8.1.14 translate

8.9 history

8.9.1 Methods

8.9.1.1 push
8.9.1.2 track

8.10 html

8.10.1 Methods

8.10.1.1 addCss
8.10.1.2 addMeta
8.10.1.3 addStyle
8.10.1.4 allowSelect
8.10.1.5 create
8.10.1.6 createCss
8.10.1.7 denySelect
8.10.1.8 getValue
8.10.1.9 index
8.10.1.10 insertBefore
8.10.1.11 locate
8.10.1.12 offset
8.10.1.13 pos
8.10.1.14 posRelative
8.10.1.15 preventEvent
8.10.1.16 remove
8.10.1.17 removeCss
8.10.1.18 stopEvent

8.11 i18n

8.11.1 Methods

8.11.1.1 dateFormatDate
8.11.1.2 dateFormatStr
8.11.1.3 fullDateFormatDate
8.11.1.4 fullDateFormatStr
8.11.1.5 intFormat
8.11.1.6 longDateFormatDate
8.11.1.7 longDateFormatStr
8.11.1.8 numberFormat
8.11.1.9 parseFormatDate
8.11.1.10 parseFormatStr
8.11.1.11 parseTimeFormatDate
8.11.1.12 parseTimeFormatStr
8.11.1.13 priceFormat
8.11.1.14 setLocale
8.11.1.15 timeFormatDate
8.11.1.16 timeFormatStr

8.11.2 Others

8.11.2.1 calendar

8.11.2.2 controls
8.11.2.3 dateFormat
8.11.2.4 decimalDelimiter
8.11.2.5 decimalSize
8.11.2.6 fileSize
8.11.2.7 fullDateFormat
8.11.2.8 groupDelimiter
8.11.2.9 groupSize
8.11.2.10 locales
8.11.2.11 longDateFormat
8.11.2.12 parseFormat
8.11.2.13 parseTimeFormat
8.11.2.14 price
8.11.2.15 priceSettings
8.11.2.16 timeFormat
8.12 locale
8.12.1 Others
8.12.1.1 pager
8.13 markup
8.13.1 Methods
8.13.1.1 init
8.13.1.2 parse
8.13.2 Others
8.13.2.1 attribute
8.13.2.2 dataTag
8.13.2.3 namespace
8.14 promise
8.14.1 Methods
8.14.1.1 all
8.14.1.2 defer
8.14.1.3 fcall
8.14.1.4 nfcall
8.15 rules
8.15.1 Methods
8.15.1.1 isChecked
8.15.1.2 isEmail
8.15.1.3 isNotEmpty
8.15.1.4 isNumber
8.16 storage.cookie
8.16.1 Methods
8.16.1.1 clear
8.16.1.2 get
8.16.1.3 put
8.16.1.4 remove
8.17 storage.local
8.17.1 Methods
8.17.1.1 clear
8.17.1.2 get
8.17.1.3 put
8.17.1.4 remove
8.18 storage.session
8.18.1 Methods
8.18.1.1 clear
8.18.1.2 get
8.18.1.3 put
8.18.1.4 remove
8.19 ui.accordion
8.19.1 Methods

8.19.1.1 addView
8.19.1.2 adjust
8.19.1.3 attachEvent
8.19.1.4 bind
8.19.1.5 blockEvent
8.19.1.6 callEvent
8.19.1.7 define
8.19.1.8 destructor
8.19.1.9 detachEvent
8.19.1.10 disable
8.19.1.11 enable
8.19.1.12 getChildViews
8.19.1.13 getFormView
8.19.1.14 getNode
8.19.1.15 getParentView
8.19.1.16 getTopParentView
8.19.1.17 hasEvent
8.19.1.18 hide
8.19.1.19 index
8.19.1.20 isEnabled
8.19.1.21 isVisible
8.19.1.22 mapEvent
8.19.1.23 reconstruct
8.19.1.24 removeView
8.19.1.25 resize
8.19.1.26 resizeChildren
8.19.1.27 show
8.19.1.28 showBatch
8.19.1.29 unbind
8.19.1.30 unblockEvent

8.19.2 Properties
8.19.2.1 animate
8.19.2.2 borderless
8.19.2.3 collapsed
8.19.2.4 cols
8.19.2.5 container
8.19.2.6 css
8.19.2.7 disabled
8.19.2.8 gravity
8.19.2.9 height
8.19.2.10 hidden
8.19.2.11 id
8.19.2.12 isolate
8.19.2.13 margin
8.19.2.14 maxHeight
8.19.2.15 maxWidth
8.19.2.16 minHeight
8.19.2.17 minWidth
8.19.2.18 multi
8.19.2.19 on
8.19.2.20 padding
8.19.2.21 paddingX
8.19.2.22 paddingY
8.19.2.23 panelClass
8.19.2.24 responsive
8.19.2.25 rows
8.19.2.26 type
8.19.2.27 visibleBatch

8.19.2.28 width
8.19.3 Events
8.19.3.1 onAfterCollapse
8.19.3.2 onAfterExpand
8.19.3.3 onBindRequest
8.19.3.4 onDestruct
8.19.4 Others
8.19.4.1 $getSize
8.19.4.2 $height
8.19.4.3 $setSize
8.19.4.4 $skin
8.19.4.5 $view
8.19.4.6 $width
8.19.4.7 config
8.19.4.8 name
8.20 ui.accordionitem
8.20.1 Methods
8.20.1.1 adjust
8.20.1.2 attachEvent
8.20.1.3 bind
8.20.1.4 blockEvent
8.20.1.5 callEvent
8.20.1.6 collapse
8.20.1.7 define
8.20.1.8 destructor
8.20.1.9 detachEvent
8.20.1.10 disable
8.20.1.11 enable
8.20.1.12 expand
8.20.1.13 getChildViews
8.20.1.14 getFormView
8.20.1.15 getNode
8.20.1.16 getParentView
8.20.1.17 getTopParentView
8.20.1.18 hasEvent
8.20.1.19 hide
8.20.1.20 isEnabled
8.20.1.21 isVisible
8.20.1.22 mapEvent
8.20.1.23 refresh
8.20.1.24 resize
8.20.1.25 show
8.20.1.26 unbind
8.20.1.27 unblockEvent
8.20.2 Properties
8.20.2.1 animate
8.20.2.2 body
8.20.2.3 borderless
8.20.2.4 collapsed
8.20.2.5 container
8.20.2.6 css
8.20.2.7 disabled
8.20.2.8 gravity
8.20.2.9 header
8.20.2.10 headerAlt
8.20.2.11 headerAltHeight
8.20.2.12 headerHeight
8.20.2.13 height

8.20.2.14 hidden
8.20.2.15 id
8.20.2.16 maxHeight
8.20.2.17 maxWidth
8.20.2.18 minHeight
8.20.2.19 minWidth
8.20.2.20 mouseEventDelay
8.20.2.21 on
8.20.2.22 onClick
8.20.2.23 onContext
8.20.2.24 onDblClick
8.20.2.25 onMouseMove
8.20.2.26 width
8.20.3 Events
8.20.3.1 onAfterContextMenu
8.20.3.2 onAfterScroll
8.20.3.3 onBeforeContextMenu
8.20.3.4 onBindRequest
8.20.3.5 onDestruct
8.20.3.6 onItemClick
8.20.3.7 onItemDblClick
8.20.3.8 onKeyPress
8.20.3.9 onLongTouch
8.20.3.10 onMouseMove
8.20.3.11 onMouseMoving
8.20.3.12 onMouseOut
8.20.3.13 onSwipeX
8.20.3.14 onSwipeY
8.20.3.15 onTimedKeyPress
8.20.3.16 onTouchEnd
8.20.3.17 onTouchMove
8.20.3.18 onTouchStart
8.20.3.19 onViewResize
8.20.4 Others
8.20.4.1 $getSize
8.20.4.2 $height
8.20.4.3 $scope
8.20.4.4 $setSize
8.20.4.5 $skin
8.20.4.6 $view
8.20.4.7 $width
8.20.4.8 config
8.20.4.9 name
8.20.4.10 on_click
8.20.4.11 on_context
8.20.4.12 on_dblclick
8.20.4.13 on_mouse_move
8.21 ui.barcode
8.21.1 Methods
8.21.1.1 adjust
8.21.1.2 bind
8.21.1.3 define
8.21.1.4 destructor
8.21.1.5 disable
8.21.1.6 enable
8.21.1.7 getChildViews
8.21.1.8 getFormView
8.21.1.9 getNode

8.21.1.10 getParentView
8.21.1.11 getTopParentView
8.21.1.12 getValue
8.21.1.13 hide
8.21.1.14 isEnabled
8.21.1.15 isVisible
8.21.1.16 render
8.21.1.17 resize
8.21.1.18 setValue
8.21.1.19 show
8.21.1.20 unbind
8.21.2 Properties
8.21.2.1 animate
8.21.2.2 borderless
8.21.2.3 color
8.21.2.4 container
8.21.2.5 css
8.21.2.6 disabled
8.21.2.7 gravity
8.21.2.8 height
8.21.2.9 hidden
8.21.2.10 id
8.21.2.11 maxHeight
8.21.2.12 maxWidth
8.21.2.13 minHeight
8.21.2.14 minWidth
8.21.2.15 paddingX
8.21.2.16 paddingY
8.21.2.17 textHeight
8.21.2.18 type
8.21.2.19 value
8.21.2.20 width
8.21.3 Events
8.21.3.1 onAfterScroll
8.21.3.2 onBindRequest
8.21.3.3 onDestruct
8.21.3.4 onKeyPress
8.21.3.5 onLongTouch
8.21.3.6 onSwipeX
8.21.3.7 onSwipeY
8.21.3.8 onTimedKeyPress
8.21.3.9 onTouchEnd
8.21.3.10 onTouchMove
8.21.3.11 onTouchStart
8.21.3.12 onViewResize
8.21.4 Others
8.21.4.1 $getSize
8.21.4.2 $height
8.21.4.3 $scope
8.21.4.4 $setSize
8.21.4.5 $skin
8.21.4.6 $view
8.21.4.7 $width
8.21.4.8 config
8.21.4.9 name
8.21.4.10 types
8.22 ui.button
8.22.1 Methods

8.22.1.1 adjust
8.22.1.2 attachEvent
8.22.1.3 bind
8.22.1.4 blockEvent
8.22.1.5 blur
8.22.1.6 callEvent
8.22.1.7 define
8.22.1.8 destructor
8.22.1.9 detachEvent
8.22.1.10 disable
8.22.1.11 enable
8.22.1.12 focus
8.22.1.13 getChildViews
8.22.1.14 getFormView
8.22.1.15 getInputNode
8.22.1.16 getNode
8.22.1.17 getParentView
8.22.1.18 getTopParentView
8.22.1.19 getValue
8.22.1.20 hasEvent
8.22.1.21 hide
8.22.1.22 isEnabled
8.22.1.23 isVisible
8.22.1.24 mapEvent
8.22.1.25 refresh
8.22.1.26 render
8.22.1.27 resize
8.22.1.28 setValue
8.22.1.29 show
8.22.1.30 sync
8.22.1.31 unbind
8.22.1.32 unblockEvent
8.22.2 Properties
8.22.2.1 align
8.22.2.2 animate
8.22.2.3 borderless
8.22.2.4 click
8.22.2.5 container
8.22.2.6 content
8.22.2.7 css
8.22.2.8 disabled
8.22.2.9 gravity
8.22.2.10 height
8.22.2.11 hidden
8.22.2.12 hotkey
8.22.2.13 id
8.22.2.14 inputHeight
8.22.2.15 inputWidth
8.22.2.16 label
8.22.2.17 labelPosition
8.22.2.18 maxHeight
8.22.2.19 maxWidth
8.22.2.20 minHeight
8.22.2.21 minWidth
8.22.2.22 name
8.22.2.23 on
8.22.2.24 placeholder
8.22.2.25 popup

8.22.2.26 tabFocus
8.22.2.27 template
8.22.2.28 tooltip
8.22.2.29 type
8.22.2.30 value
8.22.2.31 width
8.22.3 Events
8.22.3.1 onAfterRender
8.22.3.2 onAfterScroll
8.22.3.3 onBeforeRender
8.22.3.4 onBindRequest
8.22.3.5 onChange
8.22.3.6 onDestruct
8.22.3.7 onItemClick
8.22.3.8 onKeyPress
8.22.3.9 onLongTouch
8.22.3.10 onSwipeX
8.22.3.11 onSwipeY
8.22.3.12 onTimedKeyPress
8.22.3.13 onTouchEnd
8.22.3.14 onTouchMove
8.22.3.15 onTouchStart
8.22.3.16 onViewResize
8.22.4 Others
8.22.4.1 $getSize
8.22.4.2 $getValue
8.22.4.3 $height
8.22.4.4 $scope
8.22.4.5 $setSize
8.22.4.6 $setValue
8.22.4.7 $skin
8.22.4.8 $view
8.22.4.9 $width
8.22.4.10 config
8.22.4.11 name
8.22.4.12 on_click
8.22.4.13 touchable
8.23 ui.calendar
8.23.1 Methods
8.23.1.1 adjust
8.23.1.2 attachEvent
8.23.1.3 bind
8.23.1.4 blockEvent
8.23.1.5 callEvent
8.23.1.6 define
8.23.1.7 destructor
8.23.1.8 detachEvent
8.23.1.9 disable
8.23.1.10 enable
8.23.1.11 getChildViews
8.23.1.12 getFormView
8.23.1.13 getNode
8.23.1.14 getParentView
8.23.1.15 getSelectedDate
8.23.1.16 getTopParentView
8.23.1.17 getValue
8.23.1.18 getVisibleDate
8.23.1.19 hasEvent

8.23.1.20 hide
8.23.1.21 isEnabled
8.23.1.22 isVisible
8.23.1.23 locate
8.23.1.24 mapEvent
8.23.1.25 refresh
8.23.1.26 render
8.23.1.27 resize
8.23.1.28 selectDate
8.23.1.29 setValue
8.23.1.30 show
8.23.1.31 showCalendar
8.23.1.32 unbind
8.23.1.33 unblockEvent
8.23.2 Properties
8.23.2.1 animate
8.23.2.2 blockDates
8.23.2.3 borderless
8.23.2.4 calendarHeader
8.23.2.5 calendarTime
8.23.2.6 calendarWeekHeader
8.23.2.7 cellHeight
8.23.2.8 container
8.23.2.9 css
8.23.2.10 date
8.23.2.11 dayTemplate
8.23.2.12 disabled
8.23.2.13 events
8.23.2.14 gravity
8.23.2.15 headerHeight
8.23.2.16 height
8.23.2.17 hidden
8.23.2.18 icons
8.23.2.19 id
8.23.2.20 maxDate
8.23.2.21 maxHeight
8.23.2.22 maxWidth
8.23.2.23 minDate
8.23.2.24 minHeight
8.23.2.25 minWidth
8.23.2.26 minuteStep
8.23.2.27 monthSelect
8.23.2.28 mouseEventDelay
8.23.2.29 navigation
8.23.2.30 on
8.23.2.31 onClick
8.23.2.32 onContext
8.23.2.33 onDblClick
8.23.2.34 onMouseMove
8.23.2.35 select
8.23.2.36 skipEmptyWeeks
8.23.2.37 timepicker
8.23.2.38 timepickerHeight
8.23.2.39 type
8.23.2.40 weekHeader
8.23.2.41 weekNumber
8.23.2.42 width
8.23.3 Events

8.23.3.1 onAfterContextMenu
8.23.3.2 onAfterDateSelect
8.23.3.3 onAfterMonthChange
8.23.3.4 onAfterRender
8.23.3.5 onAfterScroll
8.23.3.6 onAfterZoom
8.23.3.7 onBeforeContextMenu
8.23.3.8 onBeforeDateSelect
8.23.3.9 onBeforeMonthChange
8.23.3.10 onBeforeRender
8.23.3.11 onBeforeZoom
8.23.3.12 onBindRequest
8.23.3.13 onChange
8.23.3.14 onDateClear
8.23.3.15 onDateSelect
8.23.3.16 onDestruct
8.23.3.17 onItemClick
8.23.3.18 onItemDblClick
8.23.3.19 onKeyPress
8.23.3.20 onLongTouch
8.23.3.21 onMouseMove
8.23.3.22 onMouseMoving
8.23.3.23 onMouseOut
8.23.3.24 onSwipeX
8.23.3.25 onSwipeY
8.23.3.26 onTimedKeyPress
8.23.3.27 onTodaySet
8.23.3.28 onTouchEnd
8.23.3.29 onTouchMove
8.23.3.30 onTouchStart
8.23.3.31 onViewResize
8.23.4 Others
8.23.4.1 $getSize
8.23.4.2 $height
8.23.4.3 $scope
8.23.4.4 $setSize
8.23.4.5 $skin
8.23.4.6 $view
8.23.4.7 $width
8.23.4.8 config
8.23.4.9 name
8.23.4.10 on_click
8.23.4.11 on_context
8.23.4.12 on_dblclick
8.23.4.13 on_mouse_move
8.24 ui.carousel
8.24.1 Methods
8.24.1.1 adjust
8.24.1.2 adjustScroll
8.24.1.3 attachEvent
8.24.1.4 bind
8.24.1.5 blockEvent
8.24.1.6 callEvent
8.24.1.7 define
8.24.1.8 destructor
8.24.1.9 detachEvent
8.24.1.10 disable
8.24.1.11 enable

8.24.1.12 getActiveId
8.24.1.13 getActiveIndex
8.24.1.14 getChildViews
8.24.1.15 getFormView
8.24.1.16 getLayout
8.24.1.17 getNode
8.24.1.18 getParentView
8.24.1.19 getTopParentView
8.24.1.20 hasEvent
8.24.1.21 hide
8.24.1.22 isEnabled
8.24.1.23 isVisible
8.24.1.24 mapEvent
8.24.1.25 resize
8.24.1.26 scrollTo
8.24.1.27 setActive
8.24.1.28 setActiveIndex
8.24.1.29 show
8.24.1.30 showNext
8.24.1.31 showPrev
8.24.1.32 unbind
8.24.1.33 unblockEvent
8.24.2 Properties
8.24.2.1 animate
8.24.2.2 borderless
8.24.2.3 cols
8.24.2.4 container
8.24.2.5 css
8.24.2.6 disabled
8.24.2.7 gravity
8.24.2.8 height
8.24.2.9 hidden
8.24.2.10 id
8.24.2.11 maxHeight
8.24.2.12 maxWidth
8.24.2.13 minHeight
8.24.2.14 minWidth
8.24.2.15 navigation
8.24.2.16 on
8.24.2.17 rows
8.24.2.18 scrollSpeed
8.24.2.19 type
8.24.2.20 width
8.24.3 Events
8.24.3.1 onAfterScroll
8.24.3.2 onBindRequest
8.24.3.3 onDestruct
8.24.3.4 onKeyPress
8.24.3.5 onLongTouch
8.24.3.6 onShow
8.24.3.7 onSwipeX
8.24.3.8 onSwipeY
8.24.3.9 onTimedKeyPress
8.24.3.10 onTouchEnd
8.24.3.11 onTouchMove
8.24.3.12 onTouchStart
8.24.3.13 onViewResize
8.24.4 Others

8.24.4.1 $getSize
8.24.4.2 $height
8.24.4.3 $scope
8.24.4.4 $setSize
8.24.4.5 $skin
8.24.4.6 $view
8.24.4.7 $width
8.24.4.8 config
8.24.4.9 name
8.25 ui.chart
8.25.1 Methods
8.25.1.1 add
8.25.1.2 addSeries
8.25.1.3 adjust
8.25.1.4 attachEvent
8.25.1.5 bind
8.25.1.6 blockEvent
8.25.1.7 callEvent
8.25.1.8 clearAll
8.25.1.9 clearCanvas
8.25.1.10 count
8.25.1.11 define
8.25.1.12 destructor
8.25.1.13 detachEvent
8.25.1.14 disable
8.25.1.15 enable
8.25.1.16 exists
8.25.1.17 filter
8.25.1.18 getChildViews
8.25.1.19 getFirstId
8.25.1.20 getFormView
8.25.1.21 getIdByIndex
8.25.1.22 getIndexById
8.25.1.23 getItem
8.25.1.24 getLastId
8.25.1.25 getNextId
8.25.1.26 getNode
8.25.1.27 getParentView
8.25.1.28 getPrevId
8.25.1.29 getTopParentView
8.25.1.30 group
8.25.1.31 hasEvent
8.25.1.32 hide
8.25.1.33 hideSeries
8.25.1.34 isEnabled
8.25.1.35 isVisible
8.25.1.36 load
8.25.1.37 loadNext
8.25.1.38 locate
8.25.1.39 mapEvent
8.25.1.40 parse
8.25.1.41 refresh
8.25.1.42 remove
8.25.1.43 removeAllSeries
8.25.1.44 render
8.25.1.45 resize
8.25.1.46 serialize
8.25.1.47 show

8.25.1.48 showSeries
8.25.1.49 sort
8.25.1.50 sync
8.25.1.51 unbind
8.25.1.52 unblockEvent
8.25.1.53 ungroup
8.25.1.54 updateItem
8.25.2 Properties
8.25.2.1 alpha
8.25.2.2 animate
8.25.2.3 barWidth
8.25.2.4 border
8.25.2.5 borderColor
8.25.2.6 borderless
8.25.2.7 cant
8.25.2.8 color
8.25.2.9 container
8.25.2.10 css
8.25.2.11 data
8.25.2.12 dataFeed
8.25.2.13 datathrottle
8.25.2.14 datatype
8.25.2.15 disableLines
8.25.2.16 disabled
8.25.2.17 eventRadius
8.25.2.18 fill
8.25.2.19 fixOverflow
8.25.2.20 gradient
8.25.2.21 gravity
8.25.2.22 height
8.25.2.23 hidden
8.25.2.24 id
8.25.2.25 item
8.25.2.26 label
8.25.2.27 labelOffset
8.25.2.28 legend
8.25.2.29 line
8.25.2.30 lineColor
8.25.2.31 maxHeight
8.25.2.32 maxWidth
8.25.2.33 minHeight
8.25.2.34 minWidth
8.25.2.35 mouseEventDelay
8.25.2.36 offset
8.25.2.37 on
8.25.2.38 onClick
8.25.2.39 onContext
8.25.2.40 onDblClick
8.25.2.41 onMouseMove
8.25.2.42 origin
8.25.2.43 padding
8.25.2.44 pieHeight
8.25.2.45 pieInnerText
8.25.2.46 preset
8.25.2.47 radius
8.25.2.48 ready
8.25.2.49 removeMissed
8.25.2.50 save

8.25.2.51 scale
8.25.2.52 scheme
8.25.2.53 series
8.25.2.54 shadow
8.25.2.55 tooltip
8.25.2.56 type
8.25.2.57 url
8.25.2.58 value
8.25.2.59 width
8.25.2.60 x
8.25.2.61 xAxis
8.25.2.62 xValue
8.25.2.63 y
8.25.2.64 yAxis
8.25.2.65 yValue
8.25.3 Events
8.25.3.1 onAfterAdd
8.25.3.2 onAfterContextMenu
8.25.3.3 onAfterDelete
8.25.3.4 onAfterLoad
8.25.3.5 onAfterRender
8.25.3.6 onAfterScroll
8.25.3.7 onAfterSort
8.25.3.8 onBeforeAdd
8.25.3.9 onBeforeContextMenu
8.25.3.10 onBeforeDelete
8.25.3.11 onBeforeLoad
8.25.3.12 onBeforeRender
8.25.3.13 onBeforeSort
8.25.3.14 onBindRequest
8.25.3.15 onDataRequest
8.25.3.16 onDataUpdate
8.25.3.17 onDestruct
8.25.3.18 onItemClick
8.25.3.19 onItemDblClick
8.25.3.20 onKeyPress
8.25.3.21 onLoadError
8.25.3.22 onLongTouch
8.25.3.23 onMouseMove
8.25.3.24 onMouseMoving
8.25.3.25 onMouseOut
8.25.3.26 onSwipeX
8.25.3.27 onSwipeY
8.25.3.28 onTimedKeyPress
8.25.3.29 onTouchEnd
8.25.3.30 onTouchMove
8.25.3.31 onTouchStart
8.25.3.32 onViewResize
8.25.4 Others
8.25.4.1 $getSize
8.25.4.2 $height
8.25.4.3 $scope
8.25.4.4 $setSize
8.25.4.5 $skin
8.25.4.6 $view
8.25.4.7 $width
8.25.4.8 colormap
8.25.4.9 config

8.25.4.10 name
8.25.4.11 on_click
8.25.4.12 on_context
8.25.4.13 on_dblclick
8.25.4.14 on_mouse_move
8.25.4.15 presets
8.26 ui.checkbox
8.26.1 Methods
8.26.1.1 adjust
8.26.1.2 attachEvent
8.26.1.3 bind
8.26.1.4 blockEvent
8.26.1.5 blur
8.26.1.6 callEvent
8.26.1.7 define
8.26.1.8 destructor
8.26.1.9 detachEvent
8.26.1.10 disable
8.26.1.11 enable
8.26.1.12 focus
8.26.1.13 getChildViews
8.26.1.14 getFormView
8.26.1.15 getInputNode
8.26.1.16 getNode
8.26.1.17 getParentView
8.26.1.18 getTopParentView
8.26.1.19 getValue
8.26.1.20 hasEvent
8.26.1.21 hide
8.26.1.22 isEnabled
8.26.1.23 isVisible
8.26.1.24 mapEvent
8.26.1.25 refresh
8.26.1.26 render
8.26.1.27 resize
8.26.1.28 setValue
8.26.1.29 show
8.26.1.30 sync
8.26.1.31 toggle
8.26.1.32 unbind
8.26.1.33 unblockEvent
8.26.1.34 validate
8.26.2 Properties
8.26.2.1 align
8.26.2.2 animate
8.26.2.3 attributes
8.26.2.4 borderless
8.26.2.5 bottomLabel
8.26.2.6 bottomPadding
8.26.2.7 checkValue
8.26.2.8 click
8.26.2.9 container
8.26.2.10 content
8.26.2.11 css
8.26.2.12 customCheckbox
8.26.2.13 disabled
8.26.2.14 gravity
8.26.2.15 height

8.26.2.16 hidden
8.26.2.17 hotkey
8.26.2.18 id
8.26.2.19 inputHeight
8.26.2.20 inputPadding
8.26.2.21 inputWidth
8.26.2.22 invalidMessage
8.26.2.23 label
8.26.2.24 labelPosition
8.26.2.25 labelWidth
8.26.2.26 maxHeight
8.26.2.27 maxWidth
8.26.2.28 minHeight
8.26.2.29 minWidth
8.26.2.30 name
8.26.2.31 on
8.26.2.32 placeholder
8.26.2.33 popup
8.26.2.34 readonly
8.26.2.35 relatedAction
8.26.2.36 relatedView
8.26.2.37 required
8.26.2.38 suggest
8.26.2.39 tabFocus
8.26.2.40 template
8.26.2.41 tooltip
8.26.2.42 type
8.26.2.43 uncheckValue
8.26.2.44 validate
8.26.2.45 validateEvent
8.26.2.46 value
8.26.2.47 width
8.26.3 Events
8.26.3.1 onAfterRender
8.26.3.2 onAfterScroll
8.26.3.3 onBeforeRender
8.26.3.4 onBindRequest
8.26.3.5 onChange
8.26.3.6 onDestruct
8.26.3.7 onItemClick
8.26.3.8 onKeyPress
8.26.3.9 onLongTouch
8.26.3.10 onSwipeX
8.26.3.11 onSwipeY
8.26.3.12 onTimedKeyPress
8.26.3.13 onTouchEnd
8.26.3.14 onTouchMove
8.26.3.15 onTouchStart
8.26.3.16 onViewResize
8.26.4 Others
8.26.4.1 $getSize
8.26.4.2 $getValue
8.26.4.3 $height
8.26.4.4 $render
8.26.4.5 $renderIcon
8.26.4.6 $renderInput
8.26.4.7 $renderLabel
8.26.4.8 $scope

8.26.4.9 $setSize
8.26.4.10 $setValue
8.26.4.11 $skin
8.26.4.12 $view
8.26.4.13 $width
8.26.4.14 config
8.26.4.15 name
8.26.4.16 on_click
8.26.4.17 touchable
8.27 ui.colorboard
8.27.1 Methods
8.27.1.1 adjust
8.27.1.2 attachEvent
8.27.1.3 bind
8.27.1.4 blockEvent
8.27.1.5 callEvent
8.27.1.6 define
8.27.1.7 destructor
8.27.1.8 detachEvent
8.27.1.9 disable
8.27.1.10 enable
8.27.1.11 getChildViews
8.27.1.12 getFormView
8.27.1.13 getNode
8.27.1.14 getParentView
8.27.1.15 getTopParentView
8.27.1.16 getValue
8.27.1.17 hasEvent
8.27.1.18 hide
8.27.1.19 isEnabled
8.27.1.20 isVisible
8.27.1.21 mapEvent
8.27.1.22 refresh
8.27.1.23 render
8.27.1.24 resize
8.27.1.25 setValue
8.27.1.26 show
8.27.1.27 unbind
8.27.1.28 unblockEvent
8.27.2 Properties
8.27.2.1 animate
8.27.2.2 borderless
8.27.2.3 cols
8.27.2.4 container
8.27.2.5 css
8.27.2.6 disabled
8.27.2.7 gravity
8.27.2.8 height
8.27.2.9 hidden
8.27.2.10 id
8.27.2.11 maxHeight
8.27.2.12 maxLightness
8.27.2.13 maxWidth
8.27.2.14 minHeight
8.27.2.15 minLightness
8.27.2.16 minWidth
8.27.2.17 on
8.27.2.18 palette

8.27.2.19 rows
8.27.2.20 template
8.27.2.21 value
8.27.2.22 width
8.27.3 Events
8.27.3.1 onAfterRender
8.27.3.2 onAfterScroll
8.27.3.3 onBeforeRender
8.27.3.4 onBindRequest
8.27.3.5 onDestruct
8.27.3.6 onKeyPress
8.27.3.7 onLongTouch
8.27.3.8 onSwipeX
8.27.3.9 onSwipeY
8.27.3.10 onTimedKeyPress
8.27.3.11 onTouchEnd
8.27.3.12 onTouchMove
8.27.3.13 onTouchStart
8.27.3.14 onViewResize
8.27.4 Others
8.27.4.1 $getSize
8.27.4.2 $height
8.27.4.3 $scope
8.27.4.4 $setSize
8.27.4.5 $skin
8.27.4.6 $view
8.27.4.7 $width
8.27.4.8 config
8.27.4.9 name
8.28 ui.colorpicker
8.28.1 Methods
8.28.1.1 adjust
8.28.1.2 attachEvent
8.28.1.3 bind
8.28.1.4 blockEvent
8.28.1.5 blur
8.28.1.6 callEvent
8.28.1.7 define
8.28.1.8 destructor
8.28.1.9 detachEvent
8.28.1.10 disable
8.28.1.11 enable
8.28.1.12 focus
8.28.1.13 getChildViews
8.28.1.14 getFormView
8.28.1.15 getInputNode
8.28.1.16 getNode
8.28.1.17 getParentView
8.28.1.18 getPopup
8.28.1.19 getText
8.28.1.20 getTopParentView
8.28.1.21 getValue
8.28.1.22 hasEvent
8.28.1.23 hide
8.28.1.24 isEnabled
8.28.1.25 isVisible
8.28.1.26 mapEvent
8.28.1.27 refresh

8.28.1.28 render
8.28.1.29 resize
8.28.1.30 setValue
8.28.1.31 show
8.28.1.32 sync
8.28.1.33 unbind
8.28.1.34 unblockEvent
8.28.1.35 validate
8.28.2 Properties
8.28.2.1 align
8.28.2.2 animate
8.28.2.3 attributes
8.28.2.4 borderless
8.28.2.5 bottomLabel
8.28.2.6 bottomPadding
8.28.2.7 click
8.28.2.8 container
8.28.2.9 content
8.28.2.10 css
8.28.2.11 disabled
8.28.2.12 editable
8.28.2.13 format
8.28.2.14 gravity
8.28.2.15 height
8.28.2.16 hidden
8.28.2.17 hotkey
8.28.2.18 icon
8.28.2.19 icons
8.28.2.20 id
8.28.2.21 inputHeight
8.28.2.22 inputPadding
8.28.2.23 inputWidth
8.28.2.24 invalidMessage
8.28.2.25 label
8.28.2.26 labelPosition
8.28.2.27 labelWidth
8.28.2.28 maxHeight
8.28.2.29 maxWidth
8.28.2.30 minHeight
8.28.2.31 minWidth
8.28.2.32 name
8.28.2.33 on
8.28.2.34 placeholder
8.28.2.35 popup
8.28.2.36 readonly
8.28.2.37 relatedAction
8.28.2.38 relatedView
8.28.2.39 required
8.28.2.40 stringResult
8.28.2.41 suggest
8.28.2.42 tabFocus
8.28.2.43 template
8.28.2.44 timeIcon
8.28.2.45 timepicker
8.28.2.46 tooltip
8.28.2.47 type
8.28.2.48 validate
8.28.2.49 validateEvent

8.28.2.50 value
8.28.2.51 width
8.28.3 Events
8.28.3.1 onAfterRender
8.28.3.2 onAfterScroll
8.28.3.3 onBeforeRender
8.28.3.4 onBindRequest
8.28.3.5 onChange
8.28.3.6 onDestruct
8.28.3.7 onItemClick
8.28.3.8 onKeyPress
8.28.3.9 onLongTouch
8.28.3.10 onSwipeX
8.28.3.11 onSwipeY
8.28.3.12 onTimedKeyPress
8.28.3.13 onTouchEnd
8.28.3.14 onTouchMove
8.28.3.15 onTouchStart
8.28.3.16 onViewResize
8.28.4 Others
8.28.4.1 $getSize
8.28.4.2 $getValue
8.28.4.3 $height
8.28.4.4 $render
8.28.4.5 $renderIcon
8.28.4.6 $renderInput
8.28.4.7 $renderLabel
8.28.4.8 $scope
8.28.4.9 $setSize
8.28.4.10 $setValue
8.28.4.11 $skin
8.28.4.12 $view
8.28.4.13 $width
8.28.4.14 config
8.28.4.15 name
8.28.4.16 on_click
8.28.4.17 touchable
8.29 ui.combo
8.29.1 Methods
8.29.1.1 adjust
8.29.1.2 attachEvent
8.29.1.3 bind
8.29.1.4 blockEvent
8.29.1.5 blur
8.29.1.6 callEvent
8.29.1.7 define
8.29.1.8 destructor
8.29.1.9 detachEvent
8.29.1.10 disable
8.29.1.11 enable
8.29.1.12 focus
8.29.1.13 getChildViews
8.29.1.14 getFormView
8.29.1.15 getInputNode
8.29.1.16 getNode
8.29.1.17 getParentView
8.29.1.18 getPopup
8.29.1.19 getText

8.29.1.20 getTopParentView
8.29.1.21 getValue
8.29.1.22 hasEvent
8.29.1.23 hide
8.29.1.24 isEnabled
8.29.1.25 isVisible
8.29.1.26 mapEvent
8.29.1.27 refresh
8.29.1.28 render
8.29.1.29 resize
8.29.1.30 setValue
8.29.1.31 show
8.29.1.32 sync
8.29.1.33 unbind
8.29.1.34 unblockEvent
8.29.1.35 validate
8.29.2 Properties
8.29.2.1 align
8.29.2.2 animate
8.29.2.3 attributes
8.29.2.4 borderless
8.29.2.5 bottomLabel
8.29.2.6 bottomPadding
8.29.2.7 click
8.29.2.8 container
8.29.2.9 content
8.29.2.10 css
8.29.2.11 disabled
8.29.2.12 gravity
8.29.2.13 height
8.29.2.14 hidden
8.29.2.15 hotkey
8.29.2.16 icon
8.29.2.17 id
8.29.2.18 inputHeight
8.29.2.19 inputPadding
8.29.2.20 inputWidth
8.29.2.21 invalidMessage
8.29.2.22 label
8.29.2.23 labelPosition
8.29.2.24 labelWidth
8.29.2.25 maxHeight
8.29.2.26 maxWidth
8.29.2.27 minHeight
8.29.2.28 minWidth
8.29.2.29 name
8.29.2.30 on
8.29.2.31 options
8.29.2.32 placeholder
8.29.2.33 popup
8.29.2.34 popupWidth
8.29.2.35 readonly
8.29.2.36 relatedAction
8.29.2.37 relatedView
8.29.2.38 required
8.29.2.39 suggest
8.29.2.40 tabFocus
8.29.2.41 template

8.29.2.42 text
8.29.2.43 tooltip
8.29.2.44 type
8.29.2.45 validate
8.29.2.46 validateEvent
8.29.2.47 value
8.29.2.48 width
8.29.3 Events
8.29.3.1 onAfterRender
8.29.3.2 onAfterScroll
8.29.3.3 onBeforeRender
8.29.3.4 onBindRequest
8.29.3.5 onChange
8.29.3.6 onDestruct
8.29.3.7 onItemClick
8.29.3.8 onKeyPress
8.29.3.9 onLongTouch
8.29.3.10 onSwipeX
8.29.3.11 onSwipeY
8.29.3.12 onTimedKeyPress
8.29.3.13 onTouchEnd
8.29.3.14 onTouchMove
8.29.3.15 onTouchStart
8.29.3.16 onViewResize
8.29.4 Others
8.29.4.1 $getSize
8.29.4.2 $getValue
8.29.4.3 $height
8.29.4.4 $render
8.29.4.5 $renderIcon
8.29.4.6 $renderInput
8.29.4.7 $renderLabel
8.29.4.8 $scope
8.29.4.9 $setSize
8.29.4.10 $setValue
8.29.4.11 $skin
8.29.4.12 $view
8.29.4.13 $width
8.29.4.14 config
8.29.4.15 name
8.29.4.16 on_click
8.29.4.17 touchable
8.30 ui.context
8.30.1 Methods
8.30.1.1 adjust
8.30.1.2 attachEvent
8.30.1.3 attachTo
8.30.1.4 bind
8.30.1.5 blockEvent
8.30.1.6 callEvent
8.30.1.7 close
8.30.1.8 define
8.30.1.9 destructor
8.30.1.10 detachEvent
8.30.1.11 disable
8.30.1.12 enable
8.30.1.13 getBody
8.30.1.14 getChildViews

8.30.1.15 getContext
8.30.1.16 getFormView
8.30.1.17 getHead
8.30.1.18 getNode
8.30.1.19 getParentView
8.30.1.20 getTopParentView
8.30.1.21 hasEvent
8.30.1.22 hide
8.30.1.23 isEnabled
8.30.1.24 isVisible
8.30.1.25 mapEvent
8.30.1.26 resize
8.30.1.27 resizeChildren
8.30.1.28 setPosition
8.30.1.29 show
8.30.1.30 unbind
8.30.1.31 unblockEvent
8.30.2 Properties
8.30.2.1 animate
8.30.2.2 autofit
8.30.2.3 autofocus
8.30.2.4 body
8.30.2.5 borderless
8.30.2.6 container
8.30.2.7 css
8.30.2.8 disabled
8.30.2.9 gravity
8.30.2.10 head
8.30.2.11 headHeight
8.30.2.12 height
8.30.2.13 hidden
8.30.2.14 id
8.30.2.15 left
8.30.2.16 master
8.30.2.17 maxHeight
8.30.2.18 maxWidth
8.30.2.19 minHeight
8.30.2.20 minWidth
8.30.2.21 modal
8.30.2.22 move
8.30.2.23 on
8.30.2.24 padding
8.30.2.25 position
8.30.2.26 relative
8.30.2.27 top
8.30.2.28 width
8.30.2.29 zIndex
8.30.3 Events
8.30.3.1 onAfterScroll
8.30.3.2 onBeforeShow
8.30.3.3 onBindRequest
8.30.3.4 onDestruct
8.30.3.5 onHide
8.30.3.6 onKeyPress
8.30.3.7 onLongTouch
8.30.3.8 onShow
8.30.3.9 onSwipeX
8.30.3.10 onSwipeY

8.30.3.11 onTimedKeyPress
8.30.3.12 onTouchEnd
8.30.3.13 onTouchMove
8.30.3.14 onTouchStart
8.30.3.15 onViewMove
8.30.3.16 onViewMoveEnd
8.30.3.17 onViewResize
8.30.4 Others
8.30.4.1 $getSize
8.30.4.2 $height
8.30.4.3 $scope
8.30.4.4 $setSize
8.30.4.5 $skin
8.30.4.6 $view
8.30.4.7 $width
8.30.4.8 config
8.30.4.9 name
8.31 ui.contextmenu
8.31.1 Methods
8.31.1.1 add
8.31.1.2 addCss
8.31.1.3 adjust
8.31.1.4 attachEvent
8.31.1.5 attachTo
8.31.1.6 bind
8.31.1.7 blockEvent
8.31.1.8 callEvent
8.31.1.9 clearAll
8.31.1.10 clearCss
8.31.1.11 clearValidation
8.31.1.12 close
8.31.1.13 copy
8.31.1.14 count
8.31.1.15 customize
8.31.1.16 define
8.31.1.17 destructor
8.31.1.18 detachEvent
8.31.1.19 disable
8.31.1.20 disableItem
8.31.1.21 enable
8.31.1.22 enableItem
8.31.1.23 exists
8.31.1.24 filter
8.31.1.25 getBody
8.31.1.26 getChildViews
8.31.1.27 getContext
8.31.1.28 getFirstId
8.31.1.29 getFormView
8.31.1.30 getHead
8.31.1.31 getIdByIndex
8.31.1.32 getIndexById
8.31.1.33 getItem
8.31.1.34 getItemNode
8.31.1.35 getLastId
8.31.1.36 getMenu
8.31.1.37 getMenuItem
8.31.1.38 getNextId
8.31.1.39 getNode

8.31.1.40 getPage
8.31.1.41 getPager
8.31.1.42 getParentView
8.31.1.43 getPrevId
8.31.1.44 getScrollState
8.31.1.45 getSelectedId
8.31.1.46 getSelectedItem
8.31.1.47 getSubMenu
8.31.1.48 getTopMenu
8.31.1.49 getTopParentView
8.31.1.50 getVisibleCount
8.31.1.51 hasCss
8.31.1.52 hasEvent
8.31.1.53 hide
8.31.1.54 hideItem
8.31.1.55 isEnabled
8.31.1.56 isSelected
8.31.1.57 isVisible
8.31.1.58 load
8.31.1.59 loadNext
8.31.1.60 locate
8.31.1.61 mapEvent
8.31.1.62 move
8.31.1.63 moveBottom
8.31.1.64 moveDown
8.31.1.65 moveSelection
8.31.1.66 moveTop
8.31.1.67 moveUp
8.31.1.68 parse
8.31.1.69 refresh
8.31.1.70 remove
8.31.1.71 removeCss
8.31.1.72 render
8.31.1.73 resize
8.31.1.74 resizeChildren
8.31.1.75 scrollTo
8.31.1.76 select
8.31.1.77 selectAll
8.31.1.78 serialize
8.31.1.79 setPage
8.31.1.80 setPosition
8.31.1.81 show
8.31.1.82 showItem
8.31.1.83 sort
8.31.1.84 sync
8.31.1.85 unbind
8.31.1.86 unblockEvent
8.31.1.87 unselect
8.31.1.88 unselectAll
8.31.1.89 updateItem
8.31.1.90 validate
8.31.2 Properties
8.31.2.1 animate
8.31.2.2 autofit
8.31.2.3 autofocus
8.31.2.4 autoheight
8.31.2.5 autowidth
8.31.2.6 body

8.31.2.7 borderless
8.31.2.8 click
8.31.2.9 clipboard
8.31.2.10 container
8.31.2.11 css
8.31.2.12 data
8.31.2.13 dataFeed
8.31.2.14 datathrottle
8.31.2.15 datatype
8.31.2.16 disabled
8.31.2.17 drag
8.31.2.18 dragscroll
8.31.2.19 externalData
8.31.2.20 gravity
8.31.2.21 head
8.31.2.22 headHeight
8.31.2.23 height
8.31.2.24 hidden
8.31.2.25 id
8.31.2.26 layout
8.31.2.27 left
8.31.2.28 master
8.31.2.29 maxHeight
8.31.2.30 maxWidth
8.31.2.31 minHeight
8.31.2.32 minWidth
8.31.2.33 modal
8.31.2.34 mouseEventDelay
8.31.2.35 move
8.31.2.36 multiselect
8.31.2.37 navigation
8.31.2.38 on
8.31.2.39 onClick
8.31.2.40 onContext
8.31.2.41 onDblClick
8.31.2.42 onMouseMove
8.31.2.43 padding
8.31.2.44 pager
8.31.2.45 position
8.31.2.46 ready
8.31.2.47 relative
8.31.2.48 removeMissed
8.31.2.49 rules
8.31.2.50 save
8.31.2.51 scheme
8.31.2.52 scroll
8.31.2.53 scrollSpeed
8.31.2.54 select
8.31.2.55 subMenuPos
8.31.2.56 submenu
8.31.2.57 template
8.31.2.58 templateCopy
8.31.2.59 tooltip
8.31.2.60 top
8.31.2.61 type
8.31.2.62 url
8.31.2.63 width
8.31.2.64 xCount

8.31.2.65 yCount
8.31.2.66 zIndex
8.31.3 Events
8.31.3.1 onAfterAdd
8.31.3.2 onAfterContextMenu
8.31.3.3 onAfterDelete
8.31.3.4 onAfterDrop
8.31.3.5 onAfterLoad
8.31.3.6 onAfterRender
8.31.3.7 onAfterScroll
8.31.3.8 onAfterSelect
8.31.3.9 onAfterSort
8.31.3.10 onBeforeAdd
8.31.3.11 onBeforeContextMenu
8.31.3.12 onBeforeDelete
8.31.3.13 onBeforeDrag
8.31.3.14 onBeforeDragIn
8.31.3.15 onBeforeDrop
8.31.3.16 onBeforeDropOut
8.31.3.17 onBeforeLoad
8.31.3.18 onBeforeRender
8.31.3.19 onBeforeSelect
8.31.3.20 onBeforeShow
8.31.3.21 onBeforeSort
8.31.3.22 onBindRequest
8.31.3.23 onDataRequest
8.31.3.24 onDataUpdate
8.31.3.25 onDestruct
8.31.3.26 onDragOut
8.31.3.27 onHide
8.31.3.28 onItemClick
8.31.3.29 onItemDblClick
8.31.3.30 onItemRender
8.31.3.31 onKeyPress
8.31.3.32 onLoadError
8.31.3.33 onLongTouch
8.31.3.34 onMenuItemClick
8.31.3.35 onMouseMove
8.31.3.36 onMouseMoving
8.31.3.37 onMouseOut
8.31.3.38 onPaste
8.31.3.39 onSelectChange
8.31.3.40 onShow
8.31.3.41 onSwipeX
8.31.3.42 onSwipeY
8.31.3.43 onTimedKeyPress
8.31.3.44 onTouchEnd
8.31.3.45 onTouchMove
8.31.3.46 onTouchStart
8.31.3.47 onValidationError
8.31.3.48 onValidationSuccess
8.31.3.49 onViewMove
8.31.3.50 onViewMoveEnd
8.31.3.51 onViewResize
8.31.4 Others
8.31.4.1 $drag
8.31.4.2 $dragHTML
8.31.4.3 $dragIn

8.31.4.4 $dragMark
8.31.4.5 $dragOut
8.31.4.6 $drop
8.31.4.7 $dropAllow
8.31.4.8 $getSize
8.31.4.9 $height
8.31.4.10 $scope
8.31.4.11 $setSize
8.31.4.12 $skin
8.31.4.13 $view
8.31.4.14 $width
8.31.4.15 config
8.31.4.16 name
8.31.4.17 on_click
8.31.4.18 on_context
8.31.4.19 on_dblclick
8.31.4.20 on_mouse_move
8.31.4.21 type
8.31.4.22 types
8.32 ui.counter
8.32.1 Methods
8.32.1.1 adjust
8.32.1.2 attachEvent
8.32.1.3 bind
8.32.1.4 blockEvent
8.32.1.5 blur
8.32.1.6 callEvent
8.32.1.7 define
8.32.1.8 destructor
8.32.1.9 detachEvent
8.32.1.10 disable
8.32.1.11 enable
8.32.1.12 focus
8.32.1.13 getChildViews
8.32.1.14 getFormView
8.32.1.15 getInputNode
8.32.1.16 getNode
8.32.1.17 getParentView
8.32.1.18 getTopParentView
8.32.1.19 getValue
8.32.1.20 hasEvent
8.32.1.21 hide
8.32.1.22 isEnabled
8.32.1.23 isVisible
8.32.1.24 mapEvent
8.32.1.25 next
8.32.1.26 prev
8.32.1.27 refresh
8.32.1.28 render
8.32.1.29 resize
8.32.1.30 setValue
8.32.1.31 shift
8.32.1.32 show
8.32.1.33 sync
8.32.1.34 unbind
8.32.1.35 unblockEvent
8.32.1.36 validate
8.32.2 Properties

8.32.2.1 align
8.32.2.2 animate
8.32.2.3 attributes
8.32.2.4 borderless
8.32.2.5 bottomLabel
8.32.2.6 bottomPadding
8.32.2.7 click
8.32.2.8 container
8.32.2.9 content
8.32.2.10 css
8.32.2.11 disabled
8.32.2.12 gravity
8.32.2.13 height
8.32.2.14 hidden
8.32.2.15 hotkey
8.32.2.16 id
8.32.2.17 inputHeight
8.32.2.18 inputPadding
8.32.2.19 inputWidth
8.32.2.20 invalidMessage
8.32.2.21 label
8.32.2.22 labelPosition
8.32.2.23 labelWidth
8.32.2.24 max
8.32.2.25 maxHeight
8.32.2.26 maxWidth
8.32.2.27 min
8.32.2.28 minHeight
8.32.2.29 minWidth
8.32.2.30 name
8.32.2.31 on
8.32.2.32 placeholder
8.32.2.33 popup
8.32.2.34 readonly
8.32.2.35 relatedAction
8.32.2.36 relatedView
8.32.2.37 required
8.32.2.38 step
8.32.2.39 suggest
8.32.2.40 tabFocus
8.32.2.41 template
8.32.2.42 tooltip
8.32.2.43 type
8.32.2.44 validate
8.32.2.45 validateEvent
8.32.2.46 value
8.32.2.47 width
8.32.3 Events
8.32.3.1 onAfterRender
8.32.3.2 onAfterScroll
8.32.3.3 onBeforeRender
8.32.3.4 onBindRequest
8.32.3.5 onChange
8.32.3.6 onDestruct
8.32.3.7 onItemClick
8.32.3.8 onKeyPress
8.32.3.9 onLongTouch
8.32.3.10 onSwipeX

8.32.3.11 onSwipeY
8.32.3.12 onTimedKeyPress
8.32.3.13 onTouchEnd
8.32.3.14 onTouchMove
8.32.3.15 onTouchStart
8.32.3.16 onViewResize
8.32.4 Others
8.32.4.1 $getSize
8.32.4.2 $getValue
8.32.4.3 $height
8.32.4.4 $render
8.32.4.5 $renderIcon
8.32.4.6 $renderInput
8.32.4.7 $renderLabel
8.32.4.8 $scope
8.32.4.9 $setSize
8.32.4.10 $setValue
8.32.4.11 $skin
8.32.4.12 $view
8.32.4.13 $width
8.32.4.14 config
8.32.4.15 name
8.32.4.16 on_click
8.32.4.17 touchable
8.33 ui.datasuggest
8.33.1 Methods
8.33.1.1 adjust
8.33.1.2 attachEvent
8.33.1.3 bind
8.33.1.4 blockEvent
8.33.1.5 callEvent
8.33.1.6 close
8.33.1.7 define
8.33.1.8 destructor
8.33.1.9 detachEvent
8.33.1.10 disable
8.33.1.11 enable
8.33.1.12 getBody
8.33.1.13 getChildViews
8.33.1.14 getFormView
8.33.1.15 getHead
8.33.1.16 getItemText
8.33.1.17 getList
8.33.1.18 getMasterValue
8.33.1.19 getNode
8.33.1.20 getParentView
8.33.1.21 getSuggestion
8.33.1.22 getTopParentView
8.33.1.23 getValue
8.33.1.24 hasEvent
8.33.1.25 hide
8.33.1.26 isEnabled
8.33.1.27 isVisible
8.33.1.28 linkInput
8.33.1.29 mapEvent
8.33.1.30 resize
8.33.1.31 resizeChildren
8.33.1.32 setMasterValue

8.33.1.33 setPosition
8.33.1.34 setValue
8.33.1.35 show
8.33.1.36 unbind
8.33.1.37 unblockEvent
8.33.2 Properties
8.33.2.1 animate
8.33.2.2 autofit
8.33.2.3 autofocus
8.33.2.4 body
8.33.2.5 borderless
8.33.2.6 container
8.33.2.7 css
8.33.2.8 disabled
8.33.2.9 filter
8.33.2.10 fitMaster
8.33.2.11 gravity
8.33.2.12 head
8.33.2.13 headHeight
8.33.2.14 height
8.33.2.15 hidden
8.33.2.16 id
8.33.2.17 input
8.33.2.18 keyPressTimeout
8.33.2.19 left
8.33.2.20 master
8.33.2.21 maxHeight
8.33.2.22 maxWidth
8.33.2.23 minHeight
8.33.2.24 minWidth
8.33.2.25 modal
8.33.2.26 move
8.33.2.27 on
8.33.2.28 padding
8.33.2.29 position
8.33.2.30 relative
8.33.2.31 template
8.33.2.32 textValue
8.33.2.33 top
8.33.2.34 type
8.33.2.35 width
8.33.2.36 zIndex
8.33.3 Events
8.33.3.1 onAfterScroll
8.33.3.2 onBeforeShow
8.33.3.3 onBindRequest
8.33.3.4 onDestruct
8.33.3.5 onHide
8.33.3.6 onKeyPress
8.33.3.7 onLongTouch
8.33.3.8 onShow
8.33.3.9 onSwipeX
8.33.3.10 onSwipeY
8.33.3.11 onTimedKeyPress
8.33.3.12 onTouchEnd
8.33.3.13 onTouchMove
8.33.3.14 onTouchStart
8.33.3.15 onValueSuggest

8.33.3.16 onViewMove
8.33.3.17 onViewMoveEnd
8.33.3.18 onViewResize
8.33.4 Others
8.33.4.1 $getSize
8.33.4.2 $height
8.33.4.3 $scope
8.33.4.4 $setSize
8.33.4.5 $skin
8.33.4.6 $view
8.33.4.7 $width
8.33.4.8 config
8.33.4.9 name
8.34 ui.datatable
8.34.1 Methods
8.34.1.1 add
8.34.1.2 addCellCss
8.34.1.3 addCss
8.34.1.4 addRowCss
8.34.1.5 addSpan
8.34.1.6 adjust
8.34.1.7 adjustColumn
8.34.1.8 adjustRowHeight
8.34.1.9 attachEvent
8.34.1.10 bind
8.34.1.11 blockEvent
8.34.1.12 callEvent
8.34.1.13 clearAll
8.34.1.14 clearCss
8.34.1.15 clearSelection
8.34.1.16 clearValidation
8.34.1.17 collectValues
8.34.1.18 columnId
8.34.1.19 copy
8.34.1.20 count
8.34.1.21 define
8.34.1.22 destructor
8.34.1.23 detachEvent
8.34.1.24 disable
8.34.1.25 eachColumn
8.34.1.26 eachRow
8.34.1.27 edit
8.34.1.28 editCancel
8.34.1.29 editCell
8.34.1.30 editColumn
8.34.1.31 editNext
8.34.1.32 editRow
8.34.1.33 editStop
8.34.1.34 enable
8.34.1.35 exists
8.34.1.36 exportToExcel
8.34.1.37 exportToPDF
8.34.1.38 filter
8.34.1.39 filterByAll
8.34.1.40 find
8.34.1.41 focusEditor
8.34.1.42 getChildViews
8.34.1.43 getColumnConfig

8.34.1.44 getColumnIndex
8.34.1.45 getEditState
8.34.1.46 getEditor
8.34.1.47 getEditorValue
8.34.1.48 getFilter
8.34.1.49 getFirstId
8.34.1.50 getFormView
8.34.1.51 getHeaderContent
8.34.1.52 getHeaderNode
8.34.1.53 getIdByIndex
8.34.1.54 getIndexById
8.34.1.55 getItem
8.34.1.56 getItemNode
8.34.1.57 getLastId
8.34.1.58 getNextId
8.34.1.59 getNode
8.34.1.60 getPage
8.34.1.61 getPager
8.34.1.62 getParentView
8.34.1.63 getPrevId
8.34.1.64 getScrollState
8.34.1.65 getSelectedId
8.34.1.66 getSelectedItem
8.34.1.67 getState
8.34.1.68 getText
8.34.1.69 getTopParentView
8.34.1.70 getVisibleCount
8.34.1.71 group
8.34.1.72 hasCss
8.34.1.73 hasEvent
8.34.1.74 hide
8.34.1.75 hideColumn
8.34.1.76 hideOverlay
8.34.1.77 isColumnVisible
8.34.1.78 isEnabled
8.34.1.79 isVisible
8.34.1.80 load
8.34.1.81 loadNext
8.34.1.82 locate
8.34.1.83 mapCells
8.34.1.84 mapEvent
8.34.1.85 mapSelection
8.34.1.86 markSorting
8.34.1.87 move
8.34.1.88 moveBottom
8.34.1.89 moveColumn
8.34.1.90 moveDown
8.34.1.91 moveSelection
8.34.1.92 moveTop
8.34.1.93 moveUp
8.34.1.94 parse
8.34.1.95 refresh
8.34.1.96 refreshColumns
8.34.1.97 refreshFilter
8.34.1.98 refreshHeaderContent
8.34.1.99 registerFilter
8.34.1.100 remove
8.34.1.101 removeCellCss

8.34.1.102 removeCss
8.34.1.103 removeRowCss
8.34.1.104 removeSpan
8.34.1.105 render
8.34.1.106 resize
8.34.1.107 scrollTo
8.34.1.108 select
8.34.1.109 selectRange
8.34.1.110 serialize
8.34.1.111 setColumnWidth
8.34.1.112 setPage
8.34.1.113 setRowHeight
8.34.1.114 setState
8.34.1.115 show
8.34.1.116 showCell
8.34.1.117 showColumn
8.34.1.118 showColumnBatch
8.34.1.119 showItem
8.34.1.120 showItemByIndex
8.34.1.121 showOverlay
8.34.1.122 sort
8.34.1.123 sync
8.34.1.124 unbind
8.34.1.125 unblockEvent
8.34.1.126 ungroup
8.34.1.127 unselect
8.34.1.128 updateItem
8.34.1.129 validate
8.34.1.130 validateEditor
8.34.2 Properties
8.34.2.1 animate
8.34.2.2 autoConfig
8.34.2.3 autoheight
8.34.2.4 autowidth
8.34.2.5 blockselect
8.34.2.6 borderless
8.34.2.7 checkboxRefresh
8.34.2.8 clipboard
8.34.2.9 columnWidth
8.34.2.10 columns
8.34.2.11 container
8.34.2.12 css
8.34.2.13 data
8.34.2.14 dataFeed
8.34.2.15 datafetch
8.34.2.16 datathrottle
8.34.2.17 datatype
8.34.2.18 delimiter
8.34.2.19 disabled
8.34.2.20 drag
8.34.2.21 dragColumn
8.34.2.22 dragscroll
8.34.2.23 editMath
8.34.2.24 editValue
8.34.2.25 editable
8.34.2.26 editaction
8.34.2.27 externalData
8.34.2.28 filterMode

8.34.2.29 fixedRowHeight
8.34.2.30 footer
8.34.2.31 form
8.34.2.32 gravity
8.34.2.33 header
8.34.2.34 headerRowHeight
8.34.2.35 headermenu
8.34.2.36 height
8.34.2.37 hidden
8.34.2.38 hover
8.34.2.39 id
8.34.2.40 leftSplit
8.34.2.41 liveValidation
8.34.2.42 loadahead
8.34.2.43 math
8.34.2.44 maxHeight
8.34.2.45 maxWidth
8.34.2.46 minColumnHeight
8.34.2.47 minColumnWidth
8.34.2.48 minHeight
8.34.2.49 minWidth
8.34.2.50 mouseEventDelay
8.34.2.51 multiselect
8.34.2.52 navigation
8.34.2.53 on
8.34.2.54 onClick
8.34.2.55 onContext
8.34.2.56 onDblClick
8.34.2.57 onMouseMove
8.34.2.58 pager
8.34.2.59 prerender
8.34.2.60 ready
8.34.2.61 removeMissed
8.34.2.62 resizeColumn
8.34.2.63 resizeRow
8.34.2.64 rightSplit
8.34.2.65 rowHeight
8.34.2.66 rowLineHeight
8.34.2.67 rules
8.34.2.68 save
8.34.2.69 scheme
8.34.2.70 scroll
8.34.2.71 scrollAlignY
8.34.2.72 scrollX
8.34.2.73 scrollY
8.34.2.74 select
8.34.2.75 spans
8.34.2.76 tooltip
8.34.2.77 type
8.34.2.78 url
8.34.2.79 width
8.34.2.80 yCount
8.34.3 Events
8.34.3.1 onAfterAdd
8.34.3.2 onAfterColumnDrop
8.34.3.3 onAfterColumnDropOrder
8.34.3.4 onAfterContextMenu
8.34.3.5 onAfterDelete

8.34.3.6 onAfterDrop
8.34.3.7 onAfterEditStart
8.34.3.8 onAfterEditStop
8.34.3.9 onAfterFilter
8.34.3.10 onAfterLoad
8.34.3.11 onAfterRender
8.34.3.12 onAfterScroll
8.34.3.13 onAfterSelect
8.34.3.14 onAfterSort
8.34.3.15 onAfterUnselect
8.34.3.16 onBeforeAdd
8.34.3.17 onBeforeColumnDrag
8.34.3.18 onBeforeColumnDrop
8.34.3.19 onBeforeColumnDropOrder
8.34.3.20 onBeforeContextMenu
8.34.3.21 onBeforeDelete
8.34.3.22 onBeforeDrag
8.34.3.23 onBeforeDragIn
8.34.3.24 onBeforeDrop
8.34.3.25 onBeforeDropOut
8.34.3.26 onBeforeEditStart
8.34.3.27 onBeforeEditStop
8.34.3.28 onBeforeFilter
8.34.3.29 onBeforeLoad
8.34.3.30 onBeforeRender
8.34.3.31 onBeforeSelect
8.34.3.32 onBeforeSort
8.34.3.33 onBeforeUnSelect
8.34.3.34 onBindRequest
8.34.3.35 onCheck
8.34.3.36 onColumnResize
8.34.3.37 onDataRequest
8.34.3.38 onDataUpdate
8.34.3.39 onDestruct
8.34.3.40 onDragOut
8.34.3.41 onHeaderClick
8.34.3.42 onItemClick
8.34.3.43 onItemDblClick
8.34.3.44 onKeyPress
8.34.3.45 onLiveEdit
8.34.3.46 onLoadError
8.34.3.47 onLongTouch
8.34.3.48 onMouseMove
8.34.3.49 onMouseMoving
8.34.3.50 onMouseOut
8.34.3.51 onResize
8.34.3.52 onRowResize
8.34.3.53 onScrollX
8.34.3.54 onScrollY
8.34.3.55 onSelectChange
8.34.3.56 onStructureLoad
8.34.3.57 onStructureUpdate
8.34.3.58 onSwipeX
8.34.3.59 onSwipeY
8.34.3.60 onTimedKeyPress
8.34.3.61 onTouchEnd
8.34.3.62 onTouchMove
8.34.3.63 onTouchStart

8.34.3.64 onValidationError
8.34.3.65 onValidationSuccess
8.34.3.66 onViewResize
8.34.4 Others
8.34.4.1 $drag
8.34.4.2 $dragHTML
8.34.4.3 $dragIn
8.34.4.4 $dragMark
8.34.4.5 $dragOut
8.34.4.6 $drop
8.34.4.7 $dropAllow
8.34.4.8 $getSize
8.34.4.9 $height
8.34.4.10 $scope
8.34.4.11 $setSize
8.34.4.12 $skin
8.34.4.13 $view
8.34.4.14 $width
8.34.4.15 config
8.34.4.16 headerContent
8.34.4.17 name
8.34.4.18 on_click
8.34.4.19 on_context
8.34.4.20 on_dblclick
8.34.4.21 on_mouse_move
8.34.4.22 waitData
8.35 ui.dataview
8.35.1 Methods
8.35.1.1 add
8.35.1.2 addCss
8.35.1.3 adjust
8.35.1.4 attachEvent
8.35.1.5 bind
8.35.1.6 blockEvent
8.35.1.7 callEvent
8.35.1.8 clearAll
8.35.1.9 clearCss
8.35.1.10 clearValidation
8.35.1.11 copy
8.35.1.12 count
8.35.1.13 customize
8.35.1.14 define
8.35.1.15 destructor
8.35.1.16 detachEvent
8.35.1.17 disable
8.35.1.18 enable
8.35.1.19 exists
8.35.1.20 filter
8.35.1.21 getChildViews
8.35.1.22 getFirstId
8.35.1.23 getFormView
8.35.1.24 getIdByIndex
8.35.1.25 getIndexById
8.35.1.26 getItem
8.35.1.27 getItemNode
8.35.1.28 getLastId
8.35.1.29 getNextId
8.35.1.30 getNode

8.35.1.31 getPage
8.35.1.32 getPager
8.35.1.33 getParentView
8.35.1.34 getPrevId
8.35.1.35 getScrollState
8.35.1.36 getSelectedId
8.35.1.37 getSelectedItem
8.35.1.38 getTopParentView
8.35.1.39 hasCss
8.35.1.40 hasEvent
8.35.1.41 hide
8.35.1.42 isEnabled
8.35.1.43 isSelected
8.35.1.44 isVisible
8.35.1.45 load
8.35.1.46 loadNext
8.35.1.47 locate
8.35.1.48 mapEvent
8.35.1.49 move
8.35.1.50 moveBottom
8.35.1.51 moveDown
8.35.1.52 moveSelection
8.35.1.53 moveTop
8.35.1.54 moveUp
8.35.1.55 parse
8.35.1.56 refresh
8.35.1.57 remove
8.35.1.58 removeCss
8.35.1.59 render
8.35.1.60 resize
8.35.1.61 scrollTo
8.35.1.62 select
8.35.1.63 selectAll
8.35.1.64 serialize
8.35.1.65 setPage
8.35.1.66 show
8.35.1.67 showItem
8.35.1.68 sort
8.35.1.69 sync
8.35.1.70 unbind
8.35.1.71 unblockEvent
8.35.1.72 unselect
8.35.1.73 unselectAll
8.35.1.74 updateItem
8.35.1.75 validate
8.35.2 Properties
8.35.2.1 animate
8.35.2.2 autoheight
8.35.2.3 borderless
8.35.2.4 click
8.35.2.5 container
8.35.2.6 css
8.35.2.7 data
8.35.2.8 dataFeed
8.35.2.9 datafetch
8.35.2.10 datathrottle
8.35.2.11 datatype
8.35.2.12 disabled

8.35.2.13 drag
8.35.2.14 dragscroll
8.35.2.15 externalData
8.35.2.16 gravity
8.35.2.17 height
8.35.2.18 hidden
8.35.2.19 id
8.35.2.20 loadahead
8.35.2.21 maxHeight
8.35.2.22 maxWidth
8.35.2.23 minHeight
8.35.2.24 minWidth
8.35.2.25 mouseEventDelay
8.35.2.26 multiselect
8.35.2.27 navigation
8.35.2.28 on
8.35.2.29 onClick
8.35.2.30 onContext
8.35.2.31 onDblClick
8.35.2.32 onMouseMove
8.35.2.33 pager
8.35.2.34 ready
8.35.2.35 removeMissed
8.35.2.36 rules
8.35.2.37 save
8.35.2.38 scheme
8.35.2.39 scroll
8.35.2.40 scrollSpeed
8.35.2.41 select
8.35.2.42 template
8.35.2.43 tooltip
8.35.2.44 type
8.35.2.45 url
8.35.2.46 width
8.35.2.47 xCount
8.35.2.48 yCount
8.35.3 Events
8.35.3.1 onAfterAdd
8.35.3.2 onAfterContextMenu
8.35.3.3 onAfterDelete
8.35.3.4 onAfterDrop
8.35.3.5 onAfterLoad
8.35.3.6 onAfterRender
8.35.3.7 onAfterScroll
8.35.3.8 onAfterSelect
8.35.3.9 onAfterSort
8.35.3.10 onBeforeAdd
8.35.3.11 onBeforeContextMenu
8.35.3.12 onBeforeDelete
8.35.3.13 onBeforeDrag
8.35.3.14 onBeforeDragIn
8.35.3.15 onBeforeDrop
8.35.3.16 onBeforeDropOut
8.35.3.17 onBeforeLoad
8.35.3.18 onBeforeRender
8.35.3.19 onBeforeSelect
8.35.3.20 onBeforeSort
8.35.3.21 onBindRequest

8.35.3.22 onDataRequest
8.35.3.23 onDataUpdate
8.35.3.24 onDestruct
8.35.3.25 onDragOut
8.35.3.26 onItemClick
8.35.3.27 onItemDblClick
8.35.3.28 onItemRender
8.35.3.29 onKeyPress
8.35.3.30 onLoadError
8.35.3.31 onLongTouch
8.35.3.32 onMouseMove
8.35.3.33 onMouseMoving
8.35.3.34 onMouseOut
8.35.3.35 onSelectChange
8.35.3.36 onSwipeX
8.35.3.37 onSwipeY
8.35.3.38 onTimedKeyPress
8.35.3.39 onTouchEnd
8.35.3.40 onTouchMove
8.35.3.41 onTouchStart
8.35.3.42 onValidationError
8.35.3.43 onValidationSuccess
8.35.3.44 onViewResize
8.35.4 Others
8.35.4.1 $drag
8.35.4.2 $dragHTML
8.35.4.3 $dragIn
8.35.4.4 $dragMark
8.35.4.5 $dragOut
8.35.4.6 $drop
8.35.4.7 $dropAllow
8.35.4.8 $getSize
8.35.4.9 $height
8.35.4.10 $scope
8.35.4.11 $setSize
8.35.4.12 $skin
8.35.4.13 $view
8.35.4.14 $width
8.35.4.15 config
8.35.4.16 name
8.35.4.17 on_click
8.35.4.18 on_context
8.35.4.19 on_dblclick
8.35.4.20 on_mouse_move
8.35.4.21 type
8.35.4.22 types
8.35.4.23 waitData
8.36 ui.datepicker
8.36.1 Methods
8.36.1.1 adjust
8.36.1.2 attachEvent
8.36.1.3 bind
8.36.1.4 blockEvent
8.36.1.5 blur
8.36.1.6 callEvent
8.36.1.7 define
8.36.1.8 destructor
8.36.1.9 detachEvent

8.36.1.10 disable
8.36.1.11 enable
8.36.1.12 focus
8.36.1.13 getChildViews
8.36.1.14 getFormView
8.36.1.15 getInputNode
8.36.1.16 getNode
8.36.1.17 getParentView
8.36.1.18 getPopup
8.36.1.19 getText
8.36.1.20 getTopParentView
8.36.1.21 getValue
8.36.1.22 hasEvent
8.36.1.23 hide
8.36.1.24 isEnabled
8.36.1.25 isVisible
8.36.1.26 mapEvent
8.36.1.27 refresh
8.36.1.28 render
8.36.1.29 resize
8.36.1.30 setValue
8.36.1.31 show
8.36.1.32 sync
8.36.1.33 unbind
8.36.1.34 unblockEvent
8.36.1.35 validate
8.36.2 Properties
8.36.2.1 align
8.36.2.2 animate
8.36.2.3 attributes
8.36.2.4 borderless
8.36.2.5 bottomLabel
8.36.2.6 bottomPadding
8.36.2.7 click
8.36.2.8 container
8.36.2.9 content
8.36.2.10 css
8.36.2.11 disabled
8.36.2.12 editable
8.36.2.13 format
8.36.2.14 gravity
8.36.2.15 height
8.36.2.16 hidden
8.36.2.17 hotkey
8.36.2.18 icon
8.36.2.19 icons
8.36.2.20 id
8.36.2.21 inputHeight
8.36.2.22 inputPadding
8.36.2.23 inputWidth
8.36.2.24 invalidMessage
8.36.2.25 label
8.36.2.26 labelPosition
8.36.2.27 labelWidth
8.36.2.28 maxHeight
8.36.2.29 maxWidth
8.36.2.30 minHeight
8.36.2.31 minWidth

8.36.2.32 name
8.36.2.33 on
8.36.2.34 placeholder
8.36.2.35 popup
8.36.2.36 readonly
8.36.2.37 relatedAction
8.36.2.38 relatedView
8.36.2.39 required
8.36.2.40 stringResult
8.36.2.41 suggest
8.36.2.42 tabFocus
8.36.2.43 template
8.36.2.44 text
8.36.2.45 timeIcon
8.36.2.46 timepicker
8.36.2.47 tooltip
8.36.2.48 type
8.36.2.49 validate
8.36.2.50 validateEvent
8.36.2.51 value
8.36.2.52 width
8.36.3 Events
8.36.3.1 onAfterRender
8.36.3.2 onAfterScroll
8.36.3.3 onBeforeRender
8.36.3.4 onBindRequest
8.36.3.5 onChange
8.36.3.6 onDestruct
8.36.3.7 onItemClick
8.36.3.8 onKeyPress
8.36.3.9 onLongTouch
8.36.3.10 onSwipeX
8.36.3.11 onSwipeY
8.36.3.12 onTimedKeyPress
8.36.3.13 onTouchEnd
8.36.3.14 onTouchMove
8.36.3.15 onTouchStart
8.36.3.16 onViewResize
8.36.4 Others
8.36.4.1 $getSize
8.36.4.2 $getValue
8.36.4.3 $height
8.36.4.4 $render
8.36.4.5 $renderIcon
8.36.4.6 $renderInput
8.36.4.7 $renderLabel
8.36.4.8 $scope
8.36.4.9 $setSize
8.36.4.10 $setValue
8.36.4.11 $skin
8.36.4.12 $view
8.36.4.13 $width
8.36.4.14 config
8.36.4.15 name
8.36.4.16 on_click
8.36.4.17 touchable
8.37 ui.fieldset
8.37.1 Methods

8.37.1.1 adjust
8.37.1.2 bind
8.37.1.3 define
8.37.1.4 destructor
8.37.1.5 disable
8.37.1.6 enable
8.37.1.7 getBody
8.37.1.8 getChildViews
8.37.1.9 getFormView
8.37.1.10 getNode
8.37.1.11 getParentView
8.37.1.12 getTopParentView
8.37.1.13 hide
8.37.1.14 isEnabled
8.37.1.15 isVisible
8.37.1.16 resize
8.37.1.17 show
8.37.1.18 unbind

8.37.2 Properties
8.37.2.1 animate
8.37.2.2 body
8.37.2.3 borderless
8.37.2.4 container
8.37.2.5 css
8.37.2.6 disabled
8.37.2.7 gravity
8.37.2.8 height
8.37.2.9 hidden
8.37.2.10 id
8.37.2.11 label
8.37.2.12 maxHeight
8.37.2.13 maxWidth
8.37.2.14 minHeight
8.37.2.15 minWidth
8.37.2.16 width

8.37.3 Events
8.37.3.1 onAfterScroll
8.37.3.2 onBindRequest
8.37.3.3 onDestruct
8.37.3.4 onKeyPress
8.37.3.5 onLongTouch
8.37.3.6 onSwipeX
8.37.3.7 onSwipeY
8.37.3.8 onTimedKeyPress
8.37.3.9 onTouchEnd
8.37.3.10 onTouchMove
8.37.3.11 onTouchStart
8.37.3.12 onViewResize

8.37.4 Others
8.37.4.1 $getSize
8.37.4.2 $height
8.37.4.3 $scope
8.37.4.4 $setSize
8.37.4.5 $skin
8.37.4.6 $view
8.37.4.7 $width
8.37.4.8 config
8.37.4.9 name

8.38 ui.form
8.38.1 Methods
8.38.1.1 addView
8.38.1.2 adjust
8.38.1.3 attachEvent
8.38.1.4 bind
8.38.1.5 blockEvent
8.38.1.6 callEvent
8.38.1.7 clear
8.38.1.8 clearValidation
8.38.1.9 define
8.38.1.10 destructor
8.38.1.11 detachEvent
8.38.1.12 disable
8.38.1.13 enable
8.38.1.14 focus
8.38.1.15 getChildViews
8.38.1.16 getCleanValues
8.38.1.17 getDirtyValues
8.38.1.18 getFormView
8.38.1.19 getNode
8.38.1.20 getParentView
8.38.1.21 getScrollState
8.38.1.22 getTopParentView
8.38.1.23 getValues
8.38.1.24 hasEvent
8.38.1.25 hide
8.38.1.26 index
8.38.1.27 isDirty
8.38.1.28 isEnabled
8.38.1.29 isVisible
8.38.1.30 load
8.38.1.31 mapEvent
8.38.1.32 parse
8.38.1.33 reconstruct
8.38.1.34 refresh
8.38.1.35 removeView
8.38.1.36 render
8.38.1.37 resize
8.38.1.38 resizeChildren
8.38.1.39 scrollTo
8.38.1.40 setDirty
8.38.1.41 setValues
8.38.1.42 show
8.38.1.43 showBatch
8.38.1.44 unbind
8.38.1.45 unblockEvent
8.38.1.46 validate
8.38.2 Properties
8.38.2.1 animate
8.38.2.2 autoheight
8.38.2.3 borderless
8.38.2.4 cols
8.38.2.5 container
8.38.2.6 css
8.38.2.7 data
8.38.2.8 dataFeed
8.38.2.9 datatype

8.38.2.10 disabled
8.38.2.11 elements
8.38.2.12 elementsConfig
8.38.2.13 gravity
8.38.2.14 height
8.38.2.15 hidden
8.38.2.16 id
8.38.2.17 isolate
8.38.2.18 margin
8.38.2.19 maxHeight
8.38.2.20 maxWidth
8.38.2.21 minHeight
8.38.2.22 minWidth
8.38.2.23 on
8.38.2.24 padding
8.38.2.25 paddingX
8.38.2.26 paddingY
8.38.2.27 responsive
8.38.2.28 rows
8.38.2.29 rules
8.38.2.30 scroll
8.38.2.31 scrollSpeed
8.38.2.32 type
8.38.2.33 url
8.38.2.34 visibleBatch
8.38.2.35 width
8.38.3 Events
8.38.3.1 onAfterLoad
8.38.3.2 onAfterValidation
8.38.3.3 onBeforeLoad
8.38.3.4 onBeforeValidate
8.38.3.5 onBindRequest
8.38.3.6 onChange
8.38.3.7 onDestruct
8.38.3.8 onLoadError
8.38.3.9 onSubmit
8.38.3.10 onValidationError
8.38.3.11 onValidationSuccess
8.38.3.12 onValues
8.38.4 Others
8.38.4.1 $getSize
8.38.4.2 $height
8.38.4.3 $setSize
8.38.4.4 $skin
8.38.4.5 $view
8.38.4.6 $width
8.38.4.7 config
8.38.4.8 name
8.39 ui.gridsuggest
8.39.1 Methods
8.39.1.1 adjust
8.39.1.2 attachEvent
8.39.1.3 bind
8.39.1.4 blockEvent
8.39.1.5 callEvent
8.39.1.6 close
8.39.1.7 define
8.39.1.8 destructor

8.39.1.9 detachEvent
8.39.1.10 disable
8.39.1.11 enable
8.39.1.12 getBody
8.39.1.13 getChildViews
8.39.1.14 getFormView
8.39.1.15 getHead
8.39.1.16 getItemText
8.39.1.17 getList
8.39.1.18 getMasterValue
8.39.1.19 getNode
8.39.1.20 getParentView
8.39.1.21 getSuggestion
8.39.1.22 getTopParentView
8.39.1.23 getValue
8.39.1.24 hasEvent
8.39.1.25 hide
8.39.1.26 isEnabled
8.39.1.27 isVisible
8.39.1.28 linkInput
8.39.1.29 mapEvent
8.39.1.30 resize
8.39.1.31 resizeChildren
8.39.1.32 setMasterValue
8.39.1.33 setPosition
8.39.1.34 setValue
8.39.1.35 show
8.39.1.36 unbind
8.39.1.37 unblockEvent
8.39.2 Properties
8.39.2.1 animate
8.39.2.2 autofit
8.39.2.3 autofocus
8.39.2.4 body
8.39.2.5 borderless
8.39.2.6 container
8.39.2.7 css
8.39.2.8 disabled
8.39.2.9 filter
8.39.2.10 fitMaster
8.39.2.11 gravity
8.39.2.12 head
8.39.2.13 headHeight
8.39.2.14 height
8.39.2.15 hidden
8.39.2.16 id
8.39.2.17 input
8.39.2.18 keyPressTimeout
8.39.2.19 left
8.39.2.20 master
8.39.2.21 maxHeight
8.39.2.22 maxWidth
8.39.2.23 minHeight
8.39.2.24 minWidth
8.39.2.25 modal
8.39.2.26 move
8.39.2.27 on
8.39.2.28 padding

8.39.2.29 position
8.39.2.30 relative
8.39.2.31 template
8.39.2.32 textValue
8.39.2.33 top
8.39.2.34 type
8.39.2.35 width
8.39.2.36 zIndex
8.39.3 Events
8.39.3.1 onAfterScroll
8.39.3.2 onBeforeShow
8.39.3.3 onBindRequest
8.39.3.4 onDestruct
8.39.3.5 onHide
8.39.3.6 onKeyPress
8.39.3.7 onLongTouch
8.39.3.8 onShow
8.39.3.9 onSwipeX
8.39.3.10 onSwipeY
8.39.3.11 onTimedKeyPress
8.39.3.12 onTouchEnd
8.39.3.13 onTouchMove
8.39.3.14 onTouchStart
8.39.3.15 onValueSuggest
8.39.3.16 onViewMove
8.39.3.17 onViewMoveEnd
8.39.3.18 onViewResize
8.39.4 Others
8.39.4.1 $getSize
8.39.4.2 $height
8.39.4.3 $scope
8.39.4.4 $setSize
8.39.4.5 $skin
8.39.4.6 $view
8.39.4.7 $width
8.39.4.8 config
8.39.4.9 name
8.40 ui.grouplist
8.40.1 Methods
8.40.1.1 add
8.40.1.2 addCss
8.40.1.3 adjust
8.40.1.4 attachEvent
8.40.1.5 bind
8.40.1.6 blockEvent
8.40.1.7 callEvent
8.40.1.8 clearAll
8.40.1.9 clearCss
8.40.1.10 clearValidation
8.40.1.11 copy
8.40.1.12 count
8.40.1.13 customize
8.40.1.14 define
8.40.1.15 destructor
8.40.1.16 detachEvent
8.40.1.17 disable
8.40.1.18 enable
8.40.1.19 exists

8.40.1.20 filter
8.40.1.21 getChildViews
8.40.1.22 getFirstId
8.40.1.23 getFormView
8.40.1.24 getIdByIndex
8.40.1.25 getIndexById
8.40.1.26 getItem
8.40.1.27 getItemNode
8.40.1.28 getLastId
8.40.1.29 getNextId
8.40.1.30 getNode
8.40.1.31 getOpenState
8.40.1.32 getPage
8.40.1.33 getPager
8.40.1.34 getParentView
8.40.1.35 getPrevId
8.40.1.36 getScrollState
8.40.1.37 getSelectedId
8.40.1.38 getSelectedItem
8.40.1.39 getTopParentView
8.40.1.40 getVisibleCount
8.40.1.41 group
8.40.1.42 hasCss
8.40.1.43 hasEvent
8.40.1.44 hide
8.40.1.45 isEnabled
8.40.1.46 isSelected
8.40.1.47 isVisible
8.40.1.48 load
8.40.1.49 loadNext
8.40.1.50 locate
8.40.1.51 mapEvent
8.40.1.52 move
8.40.1.53 moveBottom
8.40.1.54 moveDown
8.40.1.55 moveSelection
8.40.1.56 moveTop
8.40.1.57 moveUp
8.40.1.58 parse
8.40.1.59 refresh
8.40.1.60 remove
8.40.1.61 removeCss
8.40.1.62 render
8.40.1.63 resize
8.40.1.64 scrollTo
8.40.1.65 select
8.40.1.66 selectAll
8.40.1.67 serialize
8.40.1.68 setPage
8.40.1.69 show
8.40.1.70 showItem
8.40.1.71 sort
8.40.1.72 sync
8.40.1.73 unbind
8.40.1.74 unblockEvent
8.40.1.75 ungroup
8.40.1.76 unselect
8.40.1.77 unselectAll

8.40.1.78 updateItem
8.40.1.79 validate
8.40.2 Properties
8.40.2.1 animate
8.40.2.2 autoheight
8.40.2.3 autowidth
8.40.2.4 borderless
8.40.2.5 click
8.40.2.6 clipboard
8.40.2.7 container
8.40.2.8 css
8.40.2.9 data
8.40.2.10 dataFeed
8.40.2.11 datathrottle
8.40.2.12 datatype
8.40.2.13 disabled
8.40.2.14 drag
8.40.2.15 dragscroll
8.40.2.16 externalData
8.40.2.17 gravity
8.40.2.18 height
8.40.2.19 hidden
8.40.2.20 id
8.40.2.21 layout
8.40.2.22 maxHeight
8.40.2.23 maxWidth
8.40.2.24 minHeight
8.40.2.25 minWidth
8.40.2.26 mouseEventDelay
8.40.2.27 multiselect
8.40.2.28 navigation
8.40.2.29 on
8.40.2.30 onClick
8.40.2.31 onContext
8.40.2.32 onDblClick
8.40.2.33 onMouseMove
8.40.2.34 pager
8.40.2.35 ready
8.40.2.36 removeMissed
8.40.2.37 rules
8.40.2.38 save
8.40.2.39 scheme
8.40.2.40 scroll
8.40.2.41 scrollSpeed
8.40.2.42 select
8.40.2.43 template
8.40.2.44 templateBack
8.40.2.45 templateCopy
8.40.2.46 templateGroup
8.40.2.47 templateItem
8.40.2.48 tooltip
8.40.2.49 type
8.40.2.50 url
8.40.2.51 width
8.40.2.52 xCount
8.40.2.53 yCount
8.40.3 Events
8.40.3.1 onAfterAdd

8.40.3.2 onAfterContextMenu
8.40.3.3 onAfterDelete
8.40.3.4 onAfterDrop
8.40.3.5 onAfterLoad
8.40.3.6 onAfterRender
8.40.3.7 onAfterScroll
8.40.3.8 onAfterSelect
8.40.3.9 onAfterSort
8.40.3.10 onBeforeAdd
8.40.3.11 onBeforeContextMenu
8.40.3.12 onBeforeDelete
8.40.3.13 onBeforeDrag
8.40.3.14 onBeforeDragIn
8.40.3.15 onBeforeDrop
8.40.3.16 onBeforeDropOut
8.40.3.17 onBeforeLoad
8.40.3.18 onBeforeRender
8.40.3.19 onBeforeSelect
8.40.3.20 onBeforeSort
8.40.3.21 onBindRequest
8.40.3.22 onDataRequest
8.40.3.23 onDataUpdate
8.40.3.24 onDestruct
8.40.3.25 onDragOut
8.40.3.26 onItemClick
8.40.3.27 onItemDblClick
8.40.3.28 onItemRender
8.40.3.29 onKeyPress
8.40.3.30 onLoadError
8.40.3.31 onLongTouch
8.40.3.32 onMouseMove
8.40.3.33 onMouseMoving
8.40.3.34 onMouseOut
8.40.3.35 onPaste
8.40.3.36 onSelectChange
8.40.3.37 onSwipeX
8.40.3.38 onSwipeY
8.40.3.39 onTimedKeyPress
8.40.3.40 onTouchEnd
8.40.3.41 onTouchMove
8.40.3.42 onTouchStart
8.40.3.43 onValidationError
8.40.3.44 onValidationSuccess
8.40.3.45 onViewResize
8.40.4 Others
8.40.4.1 $drag
8.40.4.2 $dragHTML
8.40.4.3 $dragIn
8.40.4.4 $dragMark
8.40.4.5 $dragOut
8.40.4.6 $drop
8.40.4.7 $dropAllow
8.40.4.8 $getSize
8.40.4.9 $height
8.40.4.10 $scope
8.40.4.11 $setSize
8.40.4.12 $skin
8.40.4.13 $view

8.40.4.14 $width
8.40.4.15 config
8.40.4.16 name
8.40.4.17 on_click
8.40.4.18 on_context
8.40.4.19 on_dblclick
8.40.4.20 on_mouse_move
8.40.4.21 type
8.40.4.22 types
8.41 ui.headerlayout
8.41.1 Methods
8.41.1.1 addView
8.41.1.2 adjust
8.41.1.3 attachEvent
8.41.1.4 bind
8.41.1.5 blockEvent
8.41.1.6 callEvent
8.41.1.7 define
8.41.1.8 destructor
8.41.1.9 detachEvent
8.41.1.10 disable
8.41.1.11 enable
8.41.1.12 getChildViews
8.41.1.13 getFormView
8.41.1.14 getNode
8.41.1.15 getParentView
8.41.1.16 getTopParentView
8.41.1.17 hasEvent
8.41.1.18 hide
8.41.1.19 index
8.41.1.20 isEnabled
8.41.1.21 isVisible
8.41.1.22 mapEvent
8.41.1.23 reconstruct
8.41.1.24 removeView
8.41.1.25 resize
8.41.1.26 resizeChildren
8.41.1.27 show
8.41.1.28 showBatch
8.41.1.29 unbind
8.41.1.30 unblockEvent
8.41.2 Properties
8.41.2.1 animate
8.41.2.2 borderless
8.41.2.3 collapsed
8.41.2.4 cols
8.41.2.5 container
8.41.2.6 css
8.41.2.7 disabled
8.41.2.8 gravity
8.41.2.9 height
8.41.2.10 hidden
8.41.2.11 id
8.41.2.12 isolate
8.41.2.13 margin
8.41.2.14 maxHeight
8.41.2.15 maxWidth
8.41.2.16 minHeight

8.41.2.17 minWidth
8.41.2.18 multi
8.41.2.19 on
8.41.2.20 padding
8.41.2.21 paddingX
8.41.2.22 paddingY
8.41.2.23 panelClass
8.41.2.24 responsive
8.41.2.25 rows
8.41.2.26 type
8.41.2.27 visibleBatch
8.41.2.28 width
8.41.3 Events
8.41.3.1 onAfterCollapse
8.41.3.2 onAfterExpand
8.41.3.3 onBindRequest
8.41.3.4 onDestruct
8.41.4 Others
8.41.4.1 $getSize
8.41.4.2 $height
8.41.4.3 $setSize
8.41.4.4 $skin
8.41.4.5 $view
8.41.4.6 $width
8.41.4.7 config
8.41.4.8 name
8.42 ui.htmlform
8.42.1 Methods
8.42.1.1 adjust
8.42.1.2 attachEvent
8.42.1.3 bind
8.42.1.4 blockEvent
8.42.1.5 callEvent
8.42.1.6 clear
8.42.1.7 define
8.42.1.8 destructor
8.42.1.9 detachEvent
8.42.1.10 disable
8.42.1.11 enable
8.42.1.12 focus
8.42.1.13 getChildViews
8.42.1.14 getCleanValues
8.42.1.15 getDirtyValues
8.42.1.16 getFormView
8.42.1.17 getNode
8.42.1.18 getParentView
8.42.1.19 getScrollState
8.42.1.20 getTopParentView
8.42.1.21 getValues
8.42.1.22 hasEvent
8.42.1.23 hide
8.42.1.24 isDirty
8.42.1.25 isEnabled
8.42.1.26 isVisible
8.42.1.27 load
8.42.1.28 mapEvent
8.42.1.29 parse
8.42.1.30 refresh

8.42.1.31 render
8.42.1.32 resize
8.42.1.33 scrollTo
8.42.1.34 setContent
8.42.1.35 setDirty
8.42.1.36 setHTML
8.42.1.37 setValues
8.42.1.38 show
8.42.1.39 sync
8.42.1.40 unbind
8.42.1.41 unblockEvent
8.42.2 Properties
8.42.2.1 animate
8.42.2.2 autoheight
8.42.2.3 borderless
8.42.2.4 container
8.42.2.5 content
8.42.2.6 css
8.42.2.7 data
8.42.2.8 dataFeed
8.42.2.9 datatype
8.42.2.10 disabled
8.42.2.11 gravity
8.42.2.12 height
8.42.2.13 hidden
8.42.2.14 id
8.42.2.15 maxHeight
8.42.2.16 maxWidth
8.42.2.17 minHeight
8.42.2.18 minWidth
8.42.2.19 on
8.42.2.20 onClick
8.42.2.21 scroll
8.42.2.22 scrollSpeed
8.42.2.23 src
8.42.2.24 template
8.42.2.25 type
8.42.2.26 url
8.42.2.27 width
8.42.3 Events
8.42.3.1 onAfterLoad
8.42.3.2 onAfterRender
8.42.3.3 onAfterScroll
8.42.3.4 onBeforeLoad
8.42.3.5 onBeforeRender
8.42.3.6 onBindRequest
8.42.3.7 onChange
8.42.3.8 onDestruct
8.42.3.9 onKeyPress
8.42.3.10 onLoadError
8.42.3.11 onLongTouch
8.42.3.12 onSwipeX
8.42.3.13 onSwipeY
8.42.3.14 onTimedKeyPress
8.42.3.15 onTouchEnd
8.42.3.16 onTouchMove
8.42.3.17 onTouchStart
8.42.3.18 onValues

8.42.3.19 onViewResize
8.42.4 Others
8.42.4.1 $getSize
8.42.4.2 $height
8.42.4.3 $scope
8.42.4.4 $setSize
8.42.4.5 $skin
8.42.4.6 $view
8.42.4.7 $width
8.42.4.8 config
8.42.4.9 name
8.43 ui.icon
8.43.1 Methods
8.43.1.1 adjust
8.43.1.2 attachEvent
8.43.1.3 bind
8.43.1.4 blockEvent
8.43.1.5 blur
8.43.1.6 callEvent
8.43.1.7 define
8.43.1.8 destructor
8.43.1.9 detachEvent
8.43.1.10 disable
8.43.1.11 enable
8.43.1.12 focus
8.43.1.13 getChildViews
8.43.1.14 getFormView
8.43.1.15 getInputNode
8.43.1.16 getNode
8.43.1.17 getParentView
8.43.1.18 getTopParentView
8.43.1.19 getValue
8.43.1.20 hasEvent
8.43.1.21 hide
8.43.1.22 isEnabled
8.43.1.23 isVisible
8.43.1.24 mapEvent
8.43.1.25 refresh
8.43.1.26 render
8.43.1.27 resize
8.43.1.28 setValue
8.43.1.29 show
8.43.1.30 sync
8.43.1.31 unbind
8.43.1.32 unblockEvent
8.43.2 Properties
8.43.2.1 align
8.43.2.2 animate
8.43.2.3 borderless
8.43.2.4 click
8.43.2.5 container
8.43.2.6 content
8.43.2.7 css
8.43.2.8 disabled
8.43.2.9 gravity
8.43.2.10 height
8.43.2.11 hidden
8.43.2.12 hotkey

8.43.2.13 id
8.43.2.14 inputHeight
8.43.2.15 inputWidth
8.43.2.16 label
8.43.2.17 labelPosition
8.43.2.18 maxHeight
8.43.2.19 maxWidth
8.43.2.20 minHeight
8.43.2.21 minWidth
8.43.2.22 name
8.43.2.23 on
8.43.2.24 placeholder
8.43.2.25 popup
8.43.2.26 tabFocus
8.43.2.27 template
8.43.2.28 tooltip
8.43.2.29 type
8.43.2.30 value
8.43.2.31 width
8.43.3 Events
8.43.3.1 onAfterRender
8.43.3.2 onAfterScroll
8.43.3.3 onBeforeRender
8.43.3.4 onBindRequest
8.43.3.5 onChange
8.43.3.6 onDestruct
8.43.3.7 onItemClick
8.43.3.8 onKeyPress
8.43.3.9 onLongTouch
8.43.3.10 onSwipeX
8.43.3.11 onSwipeY
8.43.3.12 onTimedKeyPress
8.43.3.13 onTouchEnd
8.43.3.14 onTouchMove
8.43.3.15 onTouchStart
8.43.3.16 onViewResize
8.43.4 Others
8.43.4.1 $getSize
8.43.4.2 $getValue
8.43.4.3 $height
8.43.4.4 $scope
8.43.4.5 $setSize
8.43.4.6 $setValue
8.43.4.7 $skin
8.43.4.8 $view
8.43.4.9 $width
8.43.4.10 config
8.43.4.11 name
8.43.4.12 on_click
8.43.4.13 touchable
8.44 ui.iframe
8.44.1 Methods
8.44.1.1 adjust
8.44.1.2 attachEvent
8.44.1.3 bind
8.44.1.4 blockEvent
8.44.1.5 callEvent
8.44.1.6 define

8.44.1.7 destructor
8.44.1.8 detachEvent
8.44.1.9 disable
8.44.1.10 enable
8.44.1.11 getChildViews
8.44.1.12 getFormView
8.44.1.13 getIframe
8.44.1.14 getNode
8.44.1.15 getParentView
8.44.1.16 getTopParentView
8.44.1.17 getWindow
8.44.1.18 hasEvent
8.44.1.19 hide
8.44.1.20 isEnabled
8.44.1.21 isVisible
8.44.1.22 load
8.44.1.23 mapEvent
8.44.1.24 resize
8.44.1.25 show
8.44.1.26 unbind
8.44.1.27 unblockEvent

8.44.2 Properties
8.44.2.1 animate
8.44.2.2 borderless
8.44.2.3 container
8.44.2.4 css
8.44.2.5 disabled
8.44.2.6 gravity
8.44.2.7 height
8.44.2.8 hidden
8.44.2.9 id
8.44.2.10 maxHeight
8.44.2.11 maxWidth
8.44.2.12 minHeight
8.44.2.13 minWidth
8.44.2.14 on
8.44.2.15 src
8.44.2.16 width

8.44.3 Events
8.44.3.1 onAfterLoad
8.44.3.2 onAfterScroll
8.44.3.3 onBeforeLoad
8.44.3.4 onBindRequest
8.44.3.5 onDestruct
8.44.3.6 onKeyPress
8.44.3.7 onLongTouch
8.44.3.8 onSwipeX
8.44.3.9 onSwipeY
8.44.3.10 onTimedKeyPress
8.44.3.11 onTouchEnd
8.44.3.12 onTouchMove
8.44.3.13 onTouchStart
8.44.3.14 onViewResize

8.44.4 Others
8.44.4.1 $getSize
8.44.4.2 $height
8.44.4.3 $scope
8.44.4.4 $setSize

8.44.4.5 $skin
8.44.4.6 $view
8.44.4.7 $width
8.44.4.8 config
8.44.4.9 name
8.45 ui.label
8.45.1 Methods
8.45.1.1 adjust
8.45.1.2 attachEvent
8.45.1.3 bind
8.45.1.4 blockEvent
8.45.1.5 blur
8.45.1.6 callEvent
8.45.1.7 define
8.45.1.8 destructor
8.45.1.9 detachEvent
8.45.1.10 disable
8.45.1.11 enable
8.45.1.12 focus
8.45.1.13 getChildViews
8.45.1.14 getFormView
8.45.1.15 getInputNode
8.45.1.16 getNode
8.45.1.17 getParentView
8.45.1.18 getTopParentView
8.45.1.19 getValue
8.45.1.20 hasEvent
8.45.1.21 hide
8.45.1.22 isEnabled
8.45.1.23 isVisible
8.45.1.24 mapEvent
8.45.1.25 refresh
8.45.1.26 render
8.45.1.27 resize
8.45.1.28 setHTML
8.45.1.29 setValue
8.45.1.30 show
8.45.1.31 sync
8.45.1.32 unbind
8.45.1.33 unblockEvent
8.45.2 Properties
8.45.2.1 align
8.45.2.2 animate
8.45.2.3 borderless
8.45.2.4 click
8.45.2.5 container
8.45.2.6 content
8.45.2.7 css
8.45.2.8 disabled
8.45.2.9 gravity
8.45.2.10 height
8.45.2.11 hidden
8.45.2.12 hotkey
8.45.2.13 id
8.45.2.14 inputHeight
8.45.2.15 inputWidth
8.45.2.16 label
8.45.2.17 labelPosition

8.45.2.18 maxHeight
8.45.2.19 maxWidth
8.45.2.20 minHeight
8.45.2.21 minWidth
8.45.2.22 name
8.45.2.23 on
8.45.2.24 placeholder
8.45.2.25 popup
8.45.2.26 tabFocus
8.45.2.27 template
8.45.2.28 tooltip
8.45.2.29 type
8.45.2.30 value
8.45.2.31 width
8.45.3 Events
8.45.3.1 onAfterRender
8.45.3.2 onAfterScroll
8.45.3.3 onBeforeRender
8.45.3.4 onBindRequest
8.45.3.5 onChange
8.45.3.6 onDestruct
8.45.3.7 onItemClick
8.45.3.8 onKeyPress
8.45.3.9 onLongTouch
8.45.3.10 onSwipeX
8.45.3.11 onSwipeY
8.45.3.12 onTimedKeyPress
8.45.3.13 onTouchEnd
8.45.3.14 onTouchMove
8.45.3.15 onTouchStart
8.45.3.16 onViewResize
8.45.4 Others
8.45.4.1 $getSize
8.45.4.2 $getValue
8.45.4.3 $height
8.45.4.4 $scope
8.45.4.5 $setSize
8.45.4.6 $setValue
8.45.4.7 $skin
8.45.4.8 $view
8.45.4.9 $width
8.45.4.10 config
8.45.4.11 name
8.45.4.12 on_click
8.45.4.13 touchable
8.46 ui.layout
8.46.1 Methods
8.46.1.1 addView
8.46.1.2 adjust
8.46.1.3 attachEvent
8.46.1.4 bind
8.46.1.5 blockEvent
8.46.1.6 callEvent
8.46.1.7 define
8.46.1.8 destructor
8.46.1.9 detachEvent
8.46.1.10 disable
8.46.1.11 enable

8.46.1.12 getChildViews
8.46.1.13 getFormView
8.46.1.14 getNode
8.46.1.15 getParentView
8.46.1.16 getTopParentView
8.46.1.17 hasEvent
8.46.1.18 hide
8.46.1.19 index
8.46.1.20 isEnabled
8.46.1.21 isVisible
8.46.1.22 mapEvent
8.46.1.23 reconstruct
8.46.1.24 removeView
8.46.1.25 resize
8.46.1.26 resizeChildren
8.46.1.27 show
8.46.1.28 showBatch
8.46.1.29 unbind
8.46.1.30 unblockEvent
8.46.2 Properties
8.46.2.1 animate
8.46.2.2 borderless
8.46.2.3 cols
8.46.2.4 container
8.46.2.5 css
8.46.2.6 disabled
8.46.2.7 gravity
8.46.2.8 height
8.46.2.9 hidden
8.46.2.10 id
8.46.2.11 isolate
8.46.2.12 margin
8.46.2.13 maxHeight
8.46.2.14 maxWidth
8.46.2.15 minHeight
8.46.2.16 minWidth
8.46.2.17 on
8.46.2.18 padding
8.46.2.19 paddingX
8.46.2.20 paddingY
8.46.2.21 responsive
8.46.2.22 rows
8.46.2.23 type
8.46.2.24 visibleBatch
8.46.2.25 width
8.46.3 Events
8.46.3.1 onBindRequest
8.46.3.2 onDestruct
8.46.4 Others
8.46.4.1 $getSize
8.46.4.2 $height
8.46.4.3 $setSize
8.46.4.4 $skin
8.46.4.5 $view
8.46.4.6 $width
8.46.4.7 config
8.46.4.8 name
8.47 ui.list

8.47.1 Methods
8.47.1.1 add
8.47.1.2 addCss
8.47.1.3 adjust
8.47.1.4 attachEvent
8.47.1.5 bind
8.47.1.6 blockEvent
8.47.1.7 callEvent
8.47.1.8 clearAll
8.47.1.9 clearCss
8.47.1.10 clearValidation
8.47.1.11 copy
8.47.1.12 count
8.47.1.13 customize
8.47.1.14 define
8.47.1.15 destructor
8.47.1.16 detachEvent
8.47.1.17 disable
8.47.1.18 enable
8.47.1.19 exists
8.47.1.20 filter
8.47.1.21 getChildViews
8.47.1.22 getFirstId
8.47.1.23 getFormView
8.47.1.24 getIdByIndex
8.47.1.25 getIndexById
8.47.1.26 getItem
8.47.1.27 getItemNode
8.47.1.28 getLastId
8.47.1.29 getNextId
8.47.1.30 getNode
8.47.1.31 getPage
8.47.1.32 getPager
8.47.1.33 getParentView
8.47.1.34 getPrevId
8.47.1.35 getScrollState
8.47.1.36 getSelectedId
8.47.1.37 getSelectedItem
8.47.1.38 getTopParentView
8.47.1.39 getVisibleCount
8.47.1.40 hasCss
8.47.1.41 hasEvent
8.47.1.42 hide
8.47.1.43 isEnabled
8.47.1.44 isSelected
8.47.1.45 isVisible
8.47.1.46 load
8.47.1.47 loadNext
8.47.1.48 locate
8.47.1.49 mapEvent
8.47.1.50 move
8.47.1.51 moveBottom
8.47.1.52 moveDown
8.47.1.53 moveSelection
8.47.1.54 moveTop
8.47.1.55 moveUp
8.47.1.56 parse
8.47.1.57 refresh

8.47.1.58 remove
8.47.1.59 removeCss
8.47.1.60 render
8.47.1.61 resize
8.47.1.62 scrollTo
8.47.1.63 select
8.47.1.64 selectAll
8.47.1.65 serialize
8.47.1.66 setPage
8.47.1.67 show
8.47.1.68 showItem
8.47.1.69 sort
8.47.1.70 sync
8.47.1.71 unbind
8.47.1.72 unblockEvent
8.47.1.73 unselect
8.47.1.74 unselectAll
8.47.1.75 updateItem
8.47.1.76 validate
8.47.2 Properties
8.47.2.1 animate
8.47.2.2 autoheight
8.47.2.3 autowidth
8.47.2.4 borderless
8.47.2.5 click
8.47.2.6 clipboard
8.47.2.7 container
8.47.2.8 css
8.47.2.9 data
8.47.2.10 dataFeed
8.47.2.11 datathrottle
8.47.2.12 datatype
8.47.2.13 disabled
8.47.2.14 drag
8.47.2.15 dragscroll
8.47.2.16 externalData
8.47.2.17 gravity
8.47.2.18 height
8.47.2.19 hidden
8.47.2.20 id
8.47.2.21 layout
8.47.2.22 maxHeight
8.47.2.23 maxWidth
8.47.2.24 minHeight
8.47.2.25 minWidth
8.47.2.26 mouseEventDelay
8.47.2.27 multiselect
8.47.2.28 navigation
8.47.2.29 on
8.47.2.30 onClick
8.47.2.31 onContext
8.47.2.32 onDblClick
8.47.2.33 onMouseMove
8.47.2.34 pager
8.47.2.35 ready
8.47.2.36 removeMissed
8.47.2.37 rules
8.47.2.38 save

8.47.2.39 scheme
8.47.2.40 scroll
8.47.2.41 scrollSpeed
8.47.2.42 select
8.47.2.43 template
8.47.2.44 templateCopy
8.47.2.45 tooltip
8.47.2.46 type
8.47.2.47 url
8.47.2.48 width
8.47.2.49 xCount
8.47.2.50 yCount
8.47.3 Events
8.47.3.1 onAfterAdd
8.47.3.2 onAfterContextMenu
8.47.3.3 onAfterDelete
8.47.3.4 onAfterDrop
8.47.3.5 onAfterLoad
8.47.3.6 onAfterRender
8.47.3.7 onAfterScroll
8.47.3.8 onAfterSelect
8.47.3.9 onAfterSort
8.47.3.10 onBeforeAdd
8.47.3.11 onBeforeContextMenu
8.47.3.12 onBeforeDelete
8.47.3.13 onBeforeDrag
8.47.3.14 onBeforeDragIn
8.47.3.15 onBeforeDrop
8.47.3.16 onBeforeDropOut
8.47.3.17 onBeforeLoad
8.47.3.18 onBeforeRender
8.47.3.19 onBeforeSelect
8.47.3.20 onBeforeSort
8.47.3.21 onBindRequest
8.47.3.22 onDataRequest
8.47.3.23 onDataUpdate
8.47.3.24 onDestruct
8.47.3.25 onDragOut
8.47.3.26 onItemClick
8.47.3.27 onItemDblClick
8.47.3.28 onItemRender
8.47.3.29 onKeyPress
8.47.3.30 onLoadError
8.47.3.31 onLongTouch
8.47.3.32 onMouseMove
8.47.3.33 onMouseMoving
8.47.3.34 onMouseOut
8.47.3.35 onPaste
8.47.3.36 onSelectChange
8.47.3.37 onSwipeX
8.47.3.38 onSwipeY
8.47.3.39 onTimedKeyPress
8.47.3.40 onTouchEnd
8.47.3.41 onTouchMove
8.47.3.42 onTouchStart
8.47.3.43 onValidationError
8.47.3.44 onValidationSuccess
8.47.3.45 onViewResize

8.47.4 Others
8.47.4.1 $drag
8.47.4.2 $dragHTML
8.47.4.3 $dragIn
8.47.4.4 $dragMark
8.47.4.5 $dragOut
8.47.4.6 $drop
8.47.4.7 $dropAllow
8.47.4.8 $getSize
8.47.4.9 $height
8.47.4.10 $scope
8.47.4.11 $setSize
8.47.4.12 $skin
8.47.4.13 $view
8.47.4.14 $width
8.47.4.15 config
8.47.4.16 name
8.47.4.17 on_click
8.47.4.18 on_context
8.47.4.19 on_dblclick
8.47.4.20 on_mouse_move
8.47.4.21 type
8.47.4.22 types
8.48 ui.menu
8.48.1 Methods
8.48.1.1 add
8.48.1.2 addCss
8.48.1.3 adjust
8.48.1.4 attachEvent
8.48.1.5 bind
8.48.1.6 blockEvent
8.48.1.7 callEvent
8.48.1.8 clearAll
8.48.1.9 clearCss
8.48.1.10 clearValidation
8.48.1.11 copy
8.48.1.12 count
8.48.1.13 customize
8.48.1.14 define
8.48.1.15 destructor
8.48.1.16 detachEvent
8.48.1.17 disable
8.48.1.18 disableItem
8.48.1.19 enable
8.48.1.20 enableItem
8.48.1.21 exists
8.48.1.22 filter
8.48.1.23 getChildViews
8.48.1.24 getFirstId
8.48.1.25 getFormView
8.48.1.26 getIdByIndex
8.48.1.27 getIndexById
8.48.1.28 getItem
8.48.1.29 getItemNode
8.48.1.30 getLastId
8.48.1.31 getMenu
8.48.1.32 getMenuItem
8.48.1.33 getNextId

8.48.1.34 getNode
8.48.1.35 getPage
8.48.1.36 getPager
8.48.1.37 getParentView
8.48.1.38 getPrevId
8.48.1.39 getScrollState
8.48.1.40 getSelectedId
8.48.1.41 getSelectedItem
8.48.1.42 getSubMenu
8.48.1.43 getTopMenu
8.48.1.44 getTopParentView
8.48.1.45 getVisibleCount
8.48.1.46 hasCss
8.48.1.47 hasEvent
8.48.1.48 hide
8.48.1.49 hideItem
8.48.1.50 isEnabled
8.48.1.51 isSelected
8.48.1.52 isVisible
8.48.1.53 load
8.48.1.54 loadNext
8.48.1.55 locate
8.48.1.56 mapEvent
8.48.1.57 move
8.48.1.58 moveBottom
8.48.1.59 moveDown
8.48.1.60 moveSelection
8.48.1.61 moveTop
8.48.1.62 moveUp
8.48.1.63 parse
8.48.1.64 refresh
8.48.1.65 remove
8.48.1.66 removeCss
8.48.1.67 render
8.48.1.68 resize
8.48.1.69 scrollTo
8.48.1.70 select
8.48.1.71 selectAll
8.48.1.72 serialize
8.48.1.73 setPage
8.48.1.74 show
8.48.1.75 showItem
8.48.1.76 sort
8.48.1.77 sync
8.48.1.78 unbind
8.48.1.79 unblockEvent
8.48.1.80 unselect
8.48.1.81 unselectAll
8.48.1.82 updateItem
8.48.1.83 validate
8.48.2 Properties
8.48.2.1 animate
8.48.2.2 autoheight
8.48.2.3 autowidth
8.48.2.4 borderless
8.48.2.5 click
8.48.2.6 clipboard
8.48.2.7 container

8.48.2.8 css
8.48.2.9 data
8.48.2.10 dataFeed
8.48.2.11 datathrottle
8.48.2.12 datatype
8.48.2.13 disabled
8.48.2.14 drag
8.48.2.15 dragscroll
8.48.2.16 externalData
8.48.2.17 gravity
8.48.2.18 height
8.48.2.19 hidden
8.48.2.20 id
8.48.2.21 layout
8.48.2.22 maxHeight
8.48.2.23 maxWidth
8.48.2.24 minHeight
8.48.2.25 minWidth
8.48.2.26 mouseEventDelay
8.48.2.27 multiselect
8.48.2.28 navigation
8.48.2.29 on
8.48.2.30 onClick
8.48.2.31 onContext
8.48.2.32 onDblClick
8.48.2.33 onMouseMove
8.48.2.34 pager
8.48.2.35 ready
8.48.2.36 removeMissed
8.48.2.37 rules
8.48.2.38 save
8.48.2.39 scheme
8.48.2.40 scroll
8.48.2.41 scrollSpeed
8.48.2.42 select
8.48.2.43 subMenuPos
8.48.2.44 submenu
8.48.2.45 template
8.48.2.46 templateCopy
8.48.2.47 tooltip
8.48.2.48 type
8.48.2.49 url
8.48.2.50 width
8.48.2.51 xCount
8.48.2.52 yCount
8.48.3 Events
8.48.3.1 onAfterAdd
8.48.3.2 onAfterContextMenu
8.48.3.3 onAfterDelete
8.48.3.4 onAfterDrop
8.48.3.5 onAfterLoad
8.48.3.6 onAfterRender
8.48.3.7 onAfterScroll
8.48.3.8 onAfterSelect
8.48.3.9 onAfterSort
8.48.3.10 onBeforeAdd
8.48.3.11 onBeforeContextMenu
8.48.3.12 onBeforeDelete

8.48.3.13 onBeforeDrag
8.48.3.14 onBeforeDragIn
8.48.3.15 onBeforeDrop
8.48.3.16 onBeforeDropOut
8.48.3.17 onBeforeLoad
8.48.3.18 onBeforeRender
8.48.3.19 onBeforeSelect
8.48.3.20 onBeforeSort
8.48.3.21 onBindRequest
8.48.3.22 onDataRequest
8.48.3.23 onDataUpdate
8.48.3.24 onDestruct
8.48.3.25 onDragOut
8.48.3.26 onItemClick
8.48.3.27 onItemDblClick
8.48.3.28 onItemRender
8.48.3.29 onKeyPress
8.48.3.30 onLoadError
8.48.3.31 onLongTouch
8.48.3.32 onMenuItemClick
8.48.3.33 onMouseMove
8.48.3.34 onMouseMoving
8.48.3.35 onMouseOut
8.48.3.36 onPaste
8.48.3.37 onSelectChange
8.48.3.38 onSwipeX
8.48.3.39 onSwipeY
8.48.3.40 onTimedKeyPress
8.48.3.41 onTouchEnd
8.48.3.42 onTouchMove
8.48.3.43 onTouchStart
8.48.3.44 onValidationError
8.48.3.45 onValidationSuccess
8.48.3.46 onViewResize
8.48.4 Others
8.48.4.1 $drag
8.48.4.2 $dragHTML
8.48.4.3 $dragIn
8.48.4.4 $dragMark
8.48.4.5 $dragOut
8.48.4.6 $drop
8.48.4.7 $dropAllow
8.48.4.8 $getSize
8.48.4.9 $height
8.48.4.10 $scope
8.48.4.11 $setSize
8.48.4.12 $skin
8.48.4.13 $view
8.48.4.14 $width
8.48.4.15 config
8.48.4.16 name
8.48.4.17 on_click
8.48.4.18 on_context
8.48.4.19 on_dblclick
8.48.4.20 on_mouse_move
8.48.4.21 type
8.48.4.22 types
8.49 ui.multicombo

8.49.1 Methods
8.49.1.1 adjust
8.49.1.2 attachEvent
8.49.1.3 bind
8.49.1.4 blockEvent
8.49.1.5 blur
8.49.1.6 callEvent
8.49.1.7 define
8.49.1.8 destructor
8.49.1.9 detachEvent
8.49.1.10 disable
8.49.1.11 enable
8.49.1.12 focus
8.49.1.13 getChildViews
8.49.1.14 getFormView
8.49.1.15 getInputNode
8.49.1.16 getNode
8.49.1.17 getParentView
8.49.1.18 getPopup
8.49.1.19 getText
8.49.1.20 getTopParentView
8.49.1.21 getValue
8.49.1.22 hasEvent
8.49.1.23 hide
8.49.1.24 isEnabled
8.49.1.25 isVisible
8.49.1.26 mapEvent
8.49.1.27 refresh
8.49.1.28 render
8.49.1.29 resize
8.49.1.30 setValue
8.49.1.31 show
8.49.1.32 sync
8.49.1.33 unbind
8.49.1.34 unblockEvent
8.49.1.35 validate
8.49.2 Properties
8.49.2.1 align
8.49.2.2 animate
8.49.2.3 attributes
8.49.2.4 borderless
8.49.2.5 bottomLabel
8.49.2.6 bottomPadding
8.49.2.7 click
8.49.2.8 container
8.49.2.9 content
8.49.2.10 css
8.49.2.11 disabled
8.49.2.12 gravity
8.49.2.13 height
8.49.2.14 hidden
8.49.2.15 hotkey
8.49.2.16 icon
8.49.2.17 iconWidth
8.49.2.18 id
8.49.2.19 inputHeight
8.49.2.20 inputPadding
8.49.2.21 inputWidth

8.49.2.22 invalidMessage
8.49.2.23 label
8.49.2.24 labelPosition
8.49.2.25 labelWidth
8.49.2.26 master
8.49.2.27 maxHeight
8.49.2.28 maxWidth
8.49.2.29 minHeight
8.49.2.30 minWidth
8.49.2.31 name
8.49.2.32 on
8.49.2.33 optionWidth
8.49.2.34 options
8.49.2.35 placeholder
8.49.2.36 popup
8.49.2.37 popupWidth
8.49.2.38 readonly
8.49.2.39 relatedAction
8.49.2.40 relatedView
8.49.2.41 required
8.49.2.42 separator
8.49.2.43 suggest
8.49.2.44 tabFocus
8.49.2.45 template
8.49.2.46 text
8.49.2.47 tooltip
8.49.2.48 type
8.49.2.49 validate
8.49.2.50 validateEvent
8.49.2.51 value
8.49.2.52 width
8.49.3 Events
8.49.3.1 onAfterRender
8.49.3.2 onAfterScroll
8.49.3.3 onBeforeRender
8.49.3.4 onBindRequest
8.49.3.5 onChange
8.49.3.6 onDestruct
8.49.3.7 onItemClick
8.49.3.8 onKeyPress
8.49.3.9 onLongTouch
8.49.3.10 onSwipeX
8.49.3.11 onSwipeY
8.49.3.12 onTimedKeyPress
8.49.3.13 onTouchEnd
8.49.3.14 onTouchMove
8.49.3.15 onTouchStart
8.49.3.16 onViewResize
8.49.4 Others
8.49.4.1 $cssName
8.49.4.2 $getSize
8.49.4.3 $getValue
8.49.4.4 $height
8.49.4.5 $render
8.49.4.6 $renderIcon
8.49.4.7 $renderInput
8.49.4.8 $renderLabel
8.49.4.9 $scope

8.49.4.10 $setSize
8.49.4.11 $setValue
8.49.4.12 $skin
8.49.4.13 $view
8.49.4.14 $width
8.49.4.15 config
8.49.4.16 name
8.49.4.17 on_click
8.49.4.18 touchable
8.50 ui.multiselect
8.50.1 Methods
8.50.1.1 adjust
8.50.1.2 attachEvent
8.50.1.3 bind
8.50.1.4 blockEvent
8.50.1.5 blur
8.50.1.6 callEvent
8.50.1.7 define
8.50.1.8 destructor
8.50.1.9 detachEvent
8.50.1.10 disable
8.50.1.11 enable
8.50.1.12 focus
8.50.1.13 getChildViews
8.50.1.14 getFormView
8.50.1.15 getInputNode
8.50.1.16 getNode
8.50.1.17 getParentView
8.50.1.18 getPopup
8.50.1.19 getText
8.50.1.20 getTopParentView
8.50.1.21 getValue
8.50.1.22 hasEvent
8.50.1.23 hide
8.50.1.24 isEnabled
8.50.1.25 isVisible
8.50.1.26 mapEvent
8.50.1.27 refresh
8.50.1.28 render
8.50.1.29 resize
8.50.1.30 setValue
8.50.1.31 show
8.50.1.32 sync
8.50.1.33 unbind
8.50.1.34 unblockEvent
8.50.1.35 validate
8.50.2 Properties
8.50.2.1 align
8.50.2.2 animate
8.50.2.3 attributes
8.50.2.4 borderless
8.50.2.5 bottomLabel
8.50.2.6 bottomPadding
8.50.2.7 click
8.50.2.8 container
8.50.2.9 content
8.50.2.10 css
8.50.2.11 disabled

8.50.2.12 gravity
8.50.2.13 height
8.50.2.14 hidden
8.50.2.15 hotkey
8.50.2.16 icon
8.50.2.17 id
8.50.2.18 inputHeight
8.50.2.19 inputPadding
8.50.2.20 inputWidth
8.50.2.21 invalidMessage
8.50.2.22 label
8.50.2.23 labelPosition
8.50.2.24 labelWidth
8.50.2.25 master
8.50.2.26 maxHeight
8.50.2.27 maxWidth
8.50.2.28 minHeight
8.50.2.29 minWidth
8.50.2.30 name
8.50.2.31 on
8.50.2.32 optionWidth
8.50.2.33 options
8.50.2.34 placeholder
8.50.2.35 popup
8.50.2.36 popupWidth
8.50.2.37 readonly
8.50.2.38 relatedAction
8.50.2.39 relatedView
8.50.2.40 required
8.50.2.41 separator
8.50.2.42 suggest
8.50.2.43 tabFocus
8.50.2.44 template
8.50.2.45 text
8.50.2.46 tooltip
8.50.2.47 type
8.50.2.48 validate
8.50.2.49 validateEvent
8.50.2.50 value
8.50.2.51 width
8.50.3 Events
8.50.3.1 onAfterRender
8.50.3.2 onAfterScroll
8.50.3.3 onBeforeRender
8.50.3.4 onBindRequest
8.50.3.5 onChange
8.50.3.6 onDestruct
8.50.3.7 onItemClick
8.50.3.8 onKeyPress
8.50.3.9 onLongTouch
8.50.3.10 onSwipeX
8.50.3.11 onSwipeY
8.50.3.12 onTimedKeyPress
8.50.3.13 onTouchEnd
8.50.3.14 onTouchMove
8.50.3.15 onTouchStart
8.50.3.16 onViewResize
8.50.4 Others

8.50.4.1 $cssName
8.50.4.2 $getSize
8.50.4.3 $getValue
8.50.4.4 $height
8.50.4.5 $render
8.50.4.6 $renderIcon
8.50.4.7 $renderInput
8.50.4.8 $renderLabel
8.50.4.9 $scope
8.50.4.10 $setSize
8.50.4.11 $setValue
8.50.4.12 $skin
8.50.4.13 $view
8.50.4.14 $width
8.50.4.15 config
8.50.4.16 name
8.50.4.17 on_click
8.50.4.18 touchable
8.51 ui.multisuggest
8.51.1 Methods
8.51.1.1 adjust
8.51.1.2 attachEvent
8.51.1.3 bind
8.51.1.4 blockEvent
8.51.1.5 callEvent
8.51.1.6 close
8.51.1.7 define
8.51.1.8 destructor
8.51.1.9 detachEvent
8.51.1.10 disable
8.51.1.11 enable
8.51.1.12 getBody
8.51.1.13 getButton
8.51.1.14 getChildViews
8.51.1.15 getFormView
8.51.1.16 getHead
8.51.1.17 getItemText
8.51.1.18 getList
8.51.1.19 getMasterValue
8.51.1.20 getNode
8.51.1.21 getParentView
8.51.1.22 getSuggestion
8.51.1.23 getTopParentView
8.51.1.24 getValue
8.51.1.25 hasEvent
8.51.1.26 hide
8.51.1.27 isEnabled
8.51.1.28 isVisible
8.51.1.29 linkInput
8.51.1.30 mapEvent
8.51.1.31 resize
8.51.1.32 resizeChildren
8.51.1.33 setMasterValue
8.51.1.34 setPosition
8.51.1.35 setValue
8.51.1.36 show
8.51.1.37 unbind
8.51.1.38 unblockEvent

8.51.2 Properties
8.51.2.1 animate
8.51.2.2 autofit
8.51.2.3 autofocus
8.51.2.4 body
8.51.2.5 borderless
8.51.2.6 button
8.51.2.7 buttonText
8.51.2.8 container
8.51.2.9 css
8.51.2.10 disabled
8.51.2.11 filter
8.51.2.12 fitMaster
8.51.2.13 gravity
8.51.2.14 head
8.51.2.15 headHeight
8.51.2.16 height
8.51.2.17 hidden
8.51.2.18 id
8.51.2.19 input
8.51.2.20 keyPressTimeout
8.51.2.21 left
8.51.2.22 master
8.51.2.23 maxHeight
8.51.2.24 maxWidth
8.51.2.25 minHeight
8.51.2.26 minWidth
8.51.2.27 modal
8.51.2.28 move
8.51.2.29 on
8.51.2.30 padding
8.51.2.31 position
8.51.2.32 relative
8.51.2.33 separator
8.51.2.34 template
8.51.2.35 textValue
8.51.2.36 top
8.51.2.37 type
8.51.2.38 width
8.51.2.39 zIndex
8.51.3 Events
8.51.3.1 onAfterScroll
8.51.3.2 onBeforeShow
8.51.3.3 onBindRequest
8.51.3.4 onDestruct
8.51.3.5 onHide
8.51.3.6 onKeyPress
8.51.3.7 onLongTouch
8.51.3.8 onShow
8.51.3.9 onSwipeX
8.51.3.10 onSwipeY
8.51.3.11 onTimedKeyPress
8.51.3.12 onTouchEnd
8.51.3.13 onTouchMove
8.51.3.14 onTouchStart
8.51.3.15 onValueSuggest
8.51.3.16 onViewMove
8.51.3.17 onViewMoveEnd

8.51.3.18 onViewResize
8.51.4 Others
8.51.4.1 $getSize
8.51.4.2 $height
8.51.4.3 $scope
8.51.4.4 $setSize
8.51.4.5 $skin
8.51.4.6 $view
8.51.4.7 $width
8.51.4.8 config
8.51.4.9 name
8.52 ui.multitext
8.52.1 Methods
8.52.1.1 addSection
8.52.1.2 adjust
8.52.1.3 attachEvent
8.52.1.4 bind
8.52.1.5 blockEvent
8.52.1.6 blur
8.52.1.7 callEvent
8.52.1.8 define
8.52.1.9 destructor
8.52.1.10 detachEvent
8.52.1.11 disable
8.52.1.12 enable
8.52.1.13 focus
8.52.1.14 getChildViews
8.52.1.15 getFormView
8.52.1.16 getInputNode
8.52.1.17 getNode
8.52.1.18 getParentView
8.52.1.19 getTopParentView
8.52.1.20 getValue
8.52.1.21 getValueHere
8.52.1.22 hasEvent
8.52.1.23 hide
8.52.1.24 isEnabled
8.52.1.25 isVisible
8.52.1.26 mapEvent
8.52.1.27 refresh
8.52.1.28 removeSection
8.52.1.29 render
8.52.1.30 resize
8.52.1.31 setValue
8.52.1.32 setValueHere
8.52.1.33 show
8.52.1.34 sync
8.52.1.35 unbind
8.52.1.36 unblockEvent
8.52.1.37 validate
8.52.2 Properties
8.52.2.1 align
8.52.2.2 animate
8.52.2.3 attributes
8.52.2.4 borderless
8.52.2.5 bottomLabel
8.52.2.6 bottomPadding
8.52.2.7 click

8.52.2.8 container
8.52.2.9 content
8.52.2.10 css
8.52.2.11 disabled
8.52.2.12 gravity
8.52.2.13 height
8.52.2.14 hidden
8.52.2.15 hotkey
8.52.2.16 icon
8.52.2.17 iconWidth
8.52.2.18 id
8.52.2.19 inputHeight
8.52.2.20 inputPadding
8.52.2.21 inputWidth
8.52.2.22 invalidMessage
8.52.2.23 label
8.52.2.24 labelPosition
8.52.2.25 labelWidth
8.52.2.26 maxHeight
8.52.2.27 maxWidth
8.52.2.28 minHeight
8.52.2.29 minWidth
8.52.2.30 name
8.52.2.31 on
8.52.2.32 placeholder
8.52.2.33 popup
8.52.2.34 readonly
8.52.2.35 relatedAction
8.52.2.36 relatedView
8.52.2.37 required
8.52.2.38 separator
8.52.2.39 suggest
8.52.2.40 tabFocus
8.52.2.41 template
8.52.2.42 tooltip
8.52.2.43 type
8.52.2.44 validate
8.52.2.45 validateEvent
8.52.2.46 value
8.52.2.47 width
8.52.3 Events
8.52.3.1 onAfterRender
8.52.3.2 onAfterScroll
8.52.3.3 onBeforeRender
8.52.3.4 onBindRequest
8.52.3.5 onChange
8.52.3.6 onDestruct
8.52.3.7 onItemClick
8.52.3.8 onKeyPress
8.52.3.9 onLongTouch
8.52.3.10 onSwipeX
8.52.3.11 onSwipeY
8.52.3.12 onTimedKeyPress
8.52.3.13 onTouchEnd
8.52.3.14 onTouchMove
8.52.3.15 onTouchStart
8.52.3.16 onViewResize
8.52.4 Others

8.52.4.1 $cssName
8.52.4.2 $getSize
8.52.4.3 $getValue
8.52.4.4 $height
8.52.4.5 $render
8.52.4.6 $renderIcon
8.52.4.7 $renderInput
8.52.4.8 $renderLabel
8.52.4.9 $scope
8.52.4.10 $setSize
8.52.4.11 $setValue
8.52.4.12 $skin
8.52.4.13 $view
8.52.4.14 $width
8.52.4.15 config
8.52.4.16 name
8.52.4.17 on_click
8.52.4.18 touchable
8.53 ui.multiview
8.53.1 Methods
8.53.1.1 addView
8.53.1.2 adjust
8.53.1.3 attachEvent
8.53.1.4 back
8.53.1.5 bind
8.53.1.6 blockEvent
8.53.1.7 callEvent
8.53.1.8 define
8.53.1.9 destructor
8.53.1.10 detachEvent
8.53.1.11 disable
8.53.1.12 enable
8.53.1.13 getActiveId
8.53.1.14 getChildViews
8.53.1.15 getFormView
8.53.1.16 getNode
8.53.1.17 getParentView
8.53.1.18 getTopParentView
8.53.1.19 getValue
8.53.1.20 hasEvent
8.53.1.21 hide
8.53.1.22 index
8.53.1.23 isEnabled
8.53.1.24 isVisible
8.53.1.25 mapEvent
8.53.1.26 reconstruct
8.53.1.27 removeView
8.53.1.28 resize
8.53.1.29 resizeChildren
8.53.1.30 setValue
8.53.1.31 show
8.53.1.32 showBatch
8.53.1.33 unbind
8.53.1.34 unblockEvent
8.53.2 Properties
8.53.2.1 animate
8.53.2.2 borderless
8.53.2.3 cells

8.53.2.4 cols
8.53.2.5 container
8.53.2.6 css
8.53.2.7 disabled
8.53.2.8 fitBiggest
8.53.2.9 gravity
8.53.2.10 height
8.53.2.11 hidden
8.53.2.12 id
8.53.2.13 keepViews
8.53.2.14 maxHeight
8.53.2.15 maxWidth
8.53.2.16 minHeight
8.53.2.17 minWidth
8.53.2.18 on
8.53.2.19 responsive
8.53.2.20 rows
8.53.2.21 visibleBatch
8.53.2.22 width
8.53.3 Events
8.53.3.1 onBeforeBack
8.53.3.2 onBindRequest
8.53.3.3 onDestruct
8.53.3.4 onViewChange
8.53.4 Others
8.53.4.1 $getSize
8.53.4.2 $height
8.53.4.3 $setSize
8.53.4.4 $skin
8.53.4.5 $view
8.53.4.6 $width
8.53.4.7 config
8.53.4.8 name
8.54 ui.organogram
8.54.1 Methods
8.54.1.1 add
8.54.1.2 addCss
8.54.1.3 adjust
8.54.1.4 attachEvent
8.54.1.5 bind
8.54.1.6 blockEvent
8.54.1.7 callEvent
8.54.1.8 clearAll
8.54.1.9 clearCss
8.54.1.10 close
8.54.1.11 closeAll
8.54.1.12 count
8.54.1.13 customize
8.54.1.14 define
8.54.1.15 destructor
8.54.1.16 detachEvent
8.54.1.17 disable
8.54.1.18 enable
8.54.1.19 exists
8.54.1.20 filter
8.54.1.21 getBranchIndex
8.54.1.22 getChildViews
8.54.1.23 getFirstChildId

8.54.1.24 getFirstId
8.54.1.25 getFormView
8.54.1.26 getIdByIndex
8.54.1.27 getIndexById
8.54.1.28 getItem
8.54.1.29 getItemNode
8.54.1.30 getLastId
8.54.1.31 getNextId
8.54.1.32 getNextSiblingId
8.54.1.33 getNode
8.54.1.34 getOpenItems
8.54.1.35 getParentId
8.54.1.36 getParentView
8.54.1.37 getPrevId
8.54.1.38 getPrevSiblingId
8.54.1.39 getScrollState
8.54.1.40 getSelectedId
8.54.1.41 getSelectedItem
8.54.1.42 getState
8.54.1.43 getTopParentView
8.54.1.44 group
8.54.1.45 hasCss
8.54.1.46 hasEvent
8.54.1.47 hide
8.54.1.48 isBranch
8.54.1.49 isBranchOpen
8.54.1.50 isEnabled
8.54.1.51 isSelected
8.54.1.52 isVisible
8.54.1.53 load
8.54.1.54 loadBranch
8.54.1.55 loadNext
8.54.1.56 locate
8.54.1.57 mapEvent
8.54.1.58 open
8.54.1.59 openAll
8.54.1.60 parse
8.54.1.61 refresh
8.54.1.62 remove
8.54.1.63 removeCss
8.54.1.64 render
8.54.1.65 resize
8.54.1.66 scrollTo
8.54.1.67 select
8.54.1.68 selectAll
8.54.1.69 serialize
8.54.1.70 setState
8.54.1.71 show
8.54.1.72 showItem
8.54.1.73 sort
8.54.1.74 sync
8.54.1.75 unbind
8.54.1.76 unblockEvent
8.54.1.77 ungroup
8.54.1.78 unselect
8.54.1.79 unselectAll
8.54.1.80 updateItem
8.54.2 Properties

8.54.2.1 animate
8.54.2.2 autoheight
8.54.2.3 autowidth
8.54.2.4 borderless
8.54.2.5 click
8.54.2.6 container
8.54.2.7 css
8.54.2.8 data
8.54.2.9 dataFeed
8.54.2.10 datathrottle
8.54.2.11 datatype
8.54.2.12 disabled
8.54.2.13 filterMode
8.54.2.14 gravity
8.54.2.15 height
8.54.2.16 hidden
8.54.2.17 id
8.54.2.18 maxHeight
8.54.2.19 maxWidth
8.54.2.20 minHeight
8.54.2.21 minWidth
8.54.2.22 mouseEventDelay
8.54.2.23 multiselect
8.54.2.24 on
8.54.2.25 onClick
8.54.2.26 onContext
8.54.2.27 onDblClick
8.54.2.28 onMouseMove
8.54.2.29 ready
8.54.2.30 removeMissed
8.54.2.31 save
8.54.2.32 scheme
8.54.2.33 scroll
8.54.2.34 scrollSpeed
8.54.2.35 select
8.54.2.36 template
8.54.2.37 threeState
8.54.2.38 tooltip
8.54.2.39 type
8.54.2.40 url
8.54.2.41 width
8.54.3 Events
8.54.3.1 onAfterAdd
8.54.3.2 onAfterClose
8.54.3.3 onAfterContextMenu
8.54.3.4 onAfterDelete
8.54.3.5 onAfterLoad
8.54.3.6 onAfterOpen
8.54.3.7 onAfterRender
8.54.3.8 onAfterScroll
8.54.3.9 onAfterSelect
8.54.3.10 onAfterSort
8.54.3.11 onBeforeAdd
8.54.3.12 onBeforeClose
8.54.3.13 onBeforeContextMenu
8.54.3.14 onBeforeDelete
8.54.3.15 onBeforeLoad
8.54.3.16 onBeforeOpen

8.54.3.17 onBeforeRender
8.54.3.18 onBeforeSelect
8.54.3.19 onBeforeSort
8.54.3.20 onBindRequest
8.54.3.21 onDataRequest
8.54.3.22 onDataUpdate
8.54.3.23 onDestruct
8.54.3.24 onItemCheck
8.54.3.25 onItemClick
8.54.3.26 onItemDblClick
8.54.3.27 onItemRender
8.54.3.28 onKeyPress
8.54.3.29 onLoadError
8.54.3.30 onLongTouch
8.54.3.31 onMouseMove
8.54.3.32 onMouseMoving
8.54.3.33 onMouseOut
8.54.3.34 onSelectChange
8.54.3.35 onSwipeX
8.54.3.36 onSwipeY
8.54.3.37 onTimedKeyPress
8.54.3.38 onTouchEnd
8.54.3.39 onTouchMove
8.54.3.40 onTouchStart
8.54.3.41 onViewResize
8.54.4 Others
8.54.4.1 $getSize
8.54.4.2 $height
8.54.4.3 $scope
8.54.4.4 $setSize
8.54.4.5 $skin
8.54.4.6 $view
8.54.4.7 $width
8.54.4.8 config
8.54.4.9 name
8.54.4.10 on_click
8.54.4.11 on_context
8.54.4.12 on_dblclick
8.54.4.13 on_mouse_move
8.54.4.14 type
8.54.4.15 types
8.55 ui.pager
8.55.1 Methods
8.55.1.1 adjust
8.55.1.2 attachEvent
8.55.1.3 bind
8.55.1.4 blockEvent
8.55.1.5 callEvent
8.55.1.6 clone
8.55.1.7 customize
8.55.1.8 define
8.55.1.9 destructor
8.55.1.10 detachEvent
8.55.1.11 disable
8.55.1.12 enable
8.55.1.13 getChildViews
8.55.1.14 getFormView
8.55.1.15 getNode

8.55.1.16 getParentView
8.55.1.17 getTopParentView
8.55.1.18 hasEvent
8.55.1.19 hide
8.55.1.20 isEnabled
8.55.1.21 isVisible
8.55.1.22 mapEvent
8.55.1.23 refresh
8.55.1.24 render
8.55.1.25 resize
8.55.1.26 select
8.55.1.27 show
8.55.1.28 sync
8.55.1.29 unbind
8.55.1.30 unblockEvent
8.55.2 Properties
8.55.2.1 animate
8.55.2.2 apiOnly
8.55.2.3 borderless
8.55.2.4 container
8.55.2.5 content
8.55.2.6 count
8.55.2.7 css
8.55.2.8 disabled
8.55.2.9 gravity
8.55.2.10 group
8.55.2.11 height
8.55.2.12 hidden
8.55.2.13 id
8.55.2.14 limit
8.55.2.15 maxHeight
8.55.2.16 maxWidth
8.55.2.17 minHeight
8.55.2.18 minWidth
8.55.2.19 mouseEventDelay
8.55.2.20 on
8.55.2.21 onClick
8.55.2.22 onContext
8.55.2.23 onDblClick
8.55.2.24 onMouseMove
8.55.2.25 page
8.55.2.26 size
8.55.2.27 template
8.55.2.28 width
8.55.3 Events
8.55.3.1 onAfterContextMenu
8.55.3.2 onAfterRender
8.55.3.3 onAfterScroll
8.55.3.4 onBeforeContextMenu
8.55.3.5 onBeforeRender
8.55.3.6 onBindRequest
8.55.3.7 onDestruct
8.55.3.8 onItemClick
8.55.3.9 onItemDblClick
8.55.3.10 onKeyPress
8.55.3.11 onLongTouch
8.55.3.12 onMouseMove
8.55.3.13 onMouseMoving

8.55.3.14 onMouseOut
8.55.3.15 onSwipeX
8.55.3.16 onSwipeY
8.55.3.17 onTimedKeyPress
8.55.3.18 onTouchEnd
8.55.3.19 onTouchMove
8.55.3.20 onTouchStart
8.55.3.21 onViewResize
8.55.4 Others
8.55.4.1 $getSize
8.55.4.2 $height
8.55.4.3 $scope
8.55.4.4 $setSize
8.55.4.5 $skin
8.55.4.6 $view
8.55.4.7 $width
8.55.4.8 config
8.55.4.9 name
8.55.4.10 on_click
8.55.4.11 on_context
8.55.4.12 on_dblclick
8.55.4.13 on_mouse_move
8.55.4.14 type
8.56 ui.popup
8.56.1 Methods
8.56.1.1 adjust
8.56.1.2 attachEvent
8.56.1.3 bind
8.56.1.4 blockEvent
8.56.1.5 callEvent
8.56.1.6 close
8.56.1.7 define
8.56.1.8 destructor
8.56.1.9 detachEvent
8.56.1.10 disable
8.56.1.11 enable
8.56.1.12 getBody
8.56.1.13 getChildViews
8.56.1.14 getFormView
8.56.1.15 getHead
8.56.1.16 getNode
8.56.1.17 getParentView
8.56.1.18 getTopParentView
8.56.1.19 hasEvent
8.56.1.20 hide
8.56.1.21 isEnabled
8.56.1.22 isVisible
8.56.1.23 mapEvent
8.56.1.24 resize
8.56.1.25 resizeChildren
8.56.1.26 setPosition
8.56.1.27 show
8.56.1.28 unbind
8.56.1.29 unblockEvent
8.56.2 Properties
8.56.2.1 animate
8.56.2.2 autofit
8.56.2.3 autofocus

8.56.2.4 body
8.56.2.5 borderless
8.56.2.6 container
8.56.2.7 css
8.56.2.8 disabled
8.56.2.9 gravity
8.56.2.10 head
8.56.2.11 headHeight
8.56.2.12 height
8.56.2.13 hidden
8.56.2.14 id
8.56.2.15 left
8.56.2.16 maxHeight
8.56.2.17 maxWidth
8.56.2.18 minHeight
8.56.2.19 minWidth
8.56.2.20 modal
8.56.2.21 move
8.56.2.22 on
8.56.2.23 padding
8.56.2.24 position
8.56.2.25 relative
8.56.2.26 top
8.56.2.27 width
8.56.2.28 zIndex
8.56.3 Events
8.56.3.1 onAfterScroll
8.56.3.2 onBeforeShow
8.56.3.3 onBindRequest
8.56.3.4 onDestruct
8.56.3.5 onHide
8.56.3.6 onKeyPress
8.56.3.7 onLongTouch
8.56.3.8 onShow
8.56.3.9 onSwipeX
8.56.3.10 onSwipeY
8.56.3.11 onTimedKeyPress
8.56.3.12 onTouchEnd
8.56.3.13 onTouchMove
8.56.3.14 onTouchStart
8.56.3.15 onViewMove
8.56.3.16 onViewMoveEnd
8.56.3.17 onViewResize
8.56.4 Others
8.56.4.1 $getSize
8.56.4.2 $height
8.56.4.3 $scope
8.56.4.4 $setSize
8.56.4.5 $skin
8.56.4.6 $view
8.56.4.7 $width
8.56.4.8 config
8.56.4.9 name
8.57 ui.portlet
8.57.1 Methods
8.57.1.1 addView
8.57.1.2 adjust
8.57.1.3 attachEvent

8.57.1.4 bind
8.57.1.5 blockEvent
8.57.1.6 callEvent
8.57.1.7 define
8.57.1.8 destructor
8.57.1.9 detachEvent
8.57.1.10 disable
8.57.1.11 enable
8.57.1.12 getChildViews
8.57.1.13 getFormView
8.57.1.14 getNode
8.57.1.15 getParentView
8.57.1.16 getTopParentView
8.57.1.17 hasEvent
8.57.1.18 hide
8.57.1.19 index
8.57.1.20 isEnabled
8.57.1.21 isVisible
8.57.1.22 mapEvent
8.57.1.23 markDropArea
8.57.1.24 movePortlet
8.57.1.25 reconstruct
8.57.1.26 removeView
8.57.1.27 resize
8.57.1.28 resizeChildren
8.57.1.29 show
8.57.1.30 showBatch
8.57.1.31 unbind
8.57.1.32 unblockEvent
8.57.2 Properties
8.57.2.1 animate
8.57.2.2 body
8.57.2.3 borderless
8.57.2.4 cols
8.57.2.5 container
8.57.2.6 css
8.57.2.7 disabled
8.57.2.8 gravity
8.57.2.9 height
8.57.2.10 hidden
8.57.2.11 id
8.57.2.12 isolate
8.57.2.13 layoutType
8.57.2.14 margin
8.57.2.15 maxHeight
8.57.2.16 maxWidth
8.57.2.17 minHeight
8.57.2.18 minWidth
8.57.2.19 on
8.57.2.20 padding
8.57.2.21 paddingX
8.57.2.22 paddingY
8.57.2.23 responsive
8.57.2.24 rows
8.57.2.25 type
8.57.2.26 visibleBatch
8.57.2.27 width
8.57.3 Events

8.57.3.1 onBindRequest
8.57.3.2 onDestruct
8.57.4 Others
8.57.4.1 $drag
8.57.4.2 $getSize
8.57.4.3 $height
8.57.4.4 $setSize
8.57.4.5 $skin
8.57.4.6 $view
8.57.4.7 $width
8.57.4.8 config
8.57.4.9 name
8.58 ui.property
8.58.1 Methods
8.58.1.1 adjust
8.58.1.2 attachEvent
8.58.1.3 bind
8.58.1.4 blockEvent
8.58.1.5 callEvent
8.58.1.6 customize
8.58.1.7 define
8.58.1.8 destructor
8.58.1.9 detachEvent
8.58.1.10 disable
8.58.1.11 edit
8.58.1.12 editCancel
8.58.1.13 editNext
8.58.1.14 editStop
8.58.1.15 enable
8.58.1.16 focusEditor
8.58.1.17 getChildViews
8.58.1.18 getEditState
8.58.1.19 getEditor
8.58.1.20 getEditorValue
8.58.1.21 getFormView
8.58.1.22 getItem
8.58.1.23 getItemNode
8.58.1.24 getNode
8.58.1.25 getParentView
8.58.1.26 getScrollState
8.58.1.27 getTopParentView
8.58.1.28 getValues
8.58.1.29 hasEvent
8.58.1.30 hide
8.58.1.31 isEnabled
8.58.1.32 isVisible
8.58.1.33 load
8.58.1.34 locate
8.58.1.35 mapEvent
8.58.1.36 parse
8.58.1.37 refresh
8.58.1.38 registerType
8.58.1.39 render
8.58.1.40 resize
8.58.1.41 scrollTo
8.58.1.42 setValues
8.58.1.43 show
8.58.1.44 showItem

8.58.1.45 sync
8.58.1.46 unbind
8.58.1.47 unblockEvent
8.58.1.48 updateItem
8.58.1.49 validateEditor
8.58.2 Properties
8.58.2.1 animate
8.58.2.2 autoheight
8.58.2.3 borderless
8.58.2.4 container
8.58.2.5 content
8.58.2.6 css
8.58.2.7 data
8.58.2.8 dataFeed
8.58.2.9 datatype
8.58.2.10 disabled
8.58.2.11 editValue
8.58.2.12 editable
8.58.2.13 editaction
8.58.2.14 elements
8.58.2.15 form
8.58.2.16 gravity
8.58.2.17 height
8.58.2.18 hidden
8.58.2.19 id
8.58.2.20 maxHeight
8.58.2.21 maxWidth
8.58.2.22 minHeight
8.58.2.23 minWidth
8.58.2.24 mouseEventDelay
8.58.2.25 nameWidth
8.58.2.26 on
8.58.2.27 onClick
8.58.2.28 onContext
8.58.2.29 onDblClick
8.58.2.30 onMouseMove
8.58.2.31 scroll
8.58.2.32 scrollSpeed
8.58.2.33 template
8.58.2.34 url
8.58.2.35 width
8.58.3 Events
8.58.3.1 onAfterContextMenu
8.58.3.2 onAfterEditStart
8.58.3.3 onAfterEditStop
8.58.3.4 onAfterLoad
8.58.3.5 onAfterRender
8.58.3.6 onAfterScroll
8.58.3.7 onBeforeContextMenu
8.58.3.8 onBeforeEditStart
8.58.3.9 onBeforeEditStop
8.58.3.10 onBeforeLoad
8.58.3.11 onBeforeRender
8.58.3.12 onBindRequest
8.58.3.13 onDestruct
8.58.3.14 onItemClick
8.58.3.15 onItemDblClick
8.58.3.16 onKeyPress

8.58.3.17 onLiveEdit
8.58.3.18 onLoadError
8.58.3.19 onLongTouch
8.58.3.20 onMouseMove
8.58.3.21 onMouseMoving
8.58.3.22 onMouseOut
8.58.3.23 onSwipeX
8.58.3.24 onSwipeY
8.58.3.25 onTimedKeyPress
8.58.3.26 onTouchEnd
8.58.3.27 onTouchMove
8.58.3.28 onTouchStart
8.58.3.29 onViewResize
8.58.4 Others
8.58.4.1 $getSize
8.58.4.2 $height
8.58.4.3 $scope
8.58.4.4 $setSize
8.58.4.5 $skin
8.58.4.6 $view
8.58.4.7 $width
8.58.4.8 config
8.58.4.9 name
8.58.4.10 on_click
8.58.4.11 on_context
8.58.4.12 on_dblclick
8.58.4.13 on_edit
8.58.4.14 on_mouse_move
8.58.4.15 on_render
8.58.4.16 type
8.59 ui.radio
8.59.1 Methods
8.59.1.1 adjust
8.59.1.2 attachEvent
8.59.1.3 bind
8.59.1.4 blockEvent
8.59.1.5 blur
8.59.1.6 callEvent
8.59.1.7 define
8.59.1.8 destructor
8.59.1.9 detachEvent
8.59.1.10 disable
8.59.1.11 enable
8.59.1.12 focus
8.59.1.13 getChildViews
8.59.1.14 getFormView
8.59.1.15 getInputNode
8.59.1.16 getNode
8.59.1.17 getParentView
8.59.1.18 getTopParentView
8.59.1.19 getValue
8.59.1.20 hasEvent
8.59.1.21 hide
8.59.1.22 isEnabled
8.59.1.23 isVisible
8.59.1.24 mapEvent
8.59.1.25 refresh
8.59.1.26 render

8.59.1.27 resize
8.59.1.28 setValue
8.59.1.29 show
8.59.1.30 sync
8.59.1.31 unbind
8.59.1.32 unblockEvent
8.59.1.33 validate
8.59.2 Properties
8.59.2.1 align
8.59.2.2 animate
8.59.2.3 attributes
8.59.2.4 borderless
8.59.2.5 bottomLabel
8.59.2.6 bottomPadding
8.59.2.7 click
8.59.2.8 container
8.59.2.9 content
8.59.2.10 css
8.59.2.11 customRadio
8.59.2.12 disabled
8.59.2.13 gravity
8.59.2.14 height
8.59.2.15 hidden
8.59.2.16 hotkey
8.59.2.17 id
8.59.2.18 inputHeight
8.59.2.19 inputPadding
8.59.2.20 inputWidth
8.59.2.21 invalidMessage
8.59.2.22 label
8.59.2.23 labelPosition
8.59.2.24 labelWidth
8.59.2.25 maxHeight
8.59.2.26 maxWidth
8.59.2.27 minHeight
8.59.2.28 minWidth
8.59.2.29 name
8.59.2.30 on
8.59.2.31 optionHeight
8.59.2.32 options
8.59.2.33 placeholder
8.59.2.34 popup
8.59.2.35 readonly
8.59.2.36 relatedAction
8.59.2.37 relatedView
8.59.2.38 required
8.59.2.39 suggest
8.59.2.40 tabFocus
8.59.2.41 template
8.59.2.42 tooltip
8.59.2.43 type
8.59.2.44 validate
8.59.2.45 validateEvent
8.59.2.46 value
8.59.2.47 vertical
8.59.2.48 width
8.59.3 Events
8.59.3.1 onAfterRender

8.59.3.2 onAfterScroll
8.59.3.3 onBeforeRender
8.59.3.4 onBindRequest
8.59.3.5 onChange
8.59.3.6 onDestruct
8.59.3.7 onItemClick
8.59.3.8 onKeyPress
8.59.3.9 onLongTouch
8.59.3.10 onSwipeX
8.59.3.11 onSwipeY
8.59.3.12 onTimedKeyPress
8.59.3.13 onTouchEnd
8.59.3.14 onTouchMove
8.59.3.15 onTouchStart
8.59.3.16 onViewResize
8.59.4 Others
8.59.4.1 $getSize
8.59.4.2 $getValue
8.59.4.3 $height
8.59.4.4 $render
8.59.4.5 $renderIcon
8.59.4.6 $renderInput
8.59.4.7 $renderLabel
8.59.4.8 $scope
8.59.4.9 $setSize
8.59.4.10 $setValue
8.59.4.11 $skin
8.59.4.12 $view
8.59.4.13 $width
8.59.4.14 config
8.59.4.15 name
8.59.4.16 on_click
8.59.4.17 touchable
8.60 ui.rangeslider
8.60.1 Methods
8.60.1.1 adjust
8.60.1.2 attachEvent
8.60.1.3 bind
8.60.1.4 blockEvent
8.60.1.5 blur
8.60.1.6 callEvent
8.60.1.7 define
8.60.1.8 destructor
8.60.1.9 detachEvent
8.60.1.10 disable
8.60.1.11 enable
8.60.1.12 focus
8.60.1.13 getChildViews
8.60.1.14 getFormView
8.60.1.15 getInputNode
8.60.1.16 getNode
8.60.1.17 getParentView
8.60.1.18 getTopParentView
8.60.1.19 getValue
8.60.1.20 hasEvent
8.60.1.21 hide
8.60.1.22 isEnabled
8.60.1.23 isVisible

8.60.1.24 mapEvent
8.60.1.25 refresh
8.60.1.26 render
8.60.1.27 resize
8.60.1.28 setValue
8.60.1.29 show
8.60.1.30 sync
8.60.1.31 unbind
8.60.1.32 unblockEvent
8.60.1.33 validate

8.60.2 Properties
8.60.2.1 align
8.60.2.2 animate
8.60.2.3 attributes
8.60.2.4 borderless
8.60.2.5 bottomLabel
8.60.2.6 bottomPadding
8.60.2.7 click
8.60.2.8 container
8.60.2.9 content
8.60.2.10 css
8.60.2.11 disabled
8.60.2.12 gravity
8.60.2.13 height
8.60.2.14 hidden
8.60.2.15 hotkey
8.60.2.16 id
8.60.2.17 inputHeight
8.60.2.18 inputPadding
8.60.2.19 inputWidth
8.60.2.20 invalidMessage
8.60.2.21 label
8.60.2.22 labelPosition
8.60.2.23 labelWidth
8.60.2.24 max
8.60.2.25 maxHeight
8.60.2.26 maxWidth
8.60.2.27 min
8.60.2.28 minHeight
8.60.2.29 minWidth
8.60.2.30 name
8.60.2.31 on
8.60.2.32 placeholder
8.60.2.33 popup
8.60.2.34 readonly
8.60.2.35 relatedAction
8.60.2.36 relatedView
8.60.2.37 required
8.60.2.38 separator
8.60.2.39 step
8.60.2.40 stringResult
8.60.2.41 suggest
8.60.2.42 tabFocus
8.60.2.43 template
8.60.2.44 title
8.60.2.45 tooltip
8.60.2.46 type
8.60.2.47 validate

8.60.2.48 validateEvent
8.60.2.49 value
8.60.2.50 width
8.60.3 Events
8.60.3.1 onAfterRender
8.60.3.2 onAfterScroll
8.60.3.3 onBeforeRender
8.60.3.4 onBindRequest
8.60.3.5 onChange
8.60.3.6 onDestruct
8.60.3.7 onItemClick
8.60.3.8 onKeyPress
8.60.3.9 onLongTouch
8.60.3.10 onSliderDrag
8.60.3.11 onSwipeX
8.60.3.12 onSwipeY
8.60.3.13 onTimedKeyPress
8.60.3.14 onTouchEnd
8.60.3.15 onTouchMove
8.60.3.16 onTouchStart
8.60.3.17 onViewResize
8.60.4 Others
8.60.4.1 $cssName
8.60.4.2 $getSize
8.60.4.3 $getValue
8.60.4.4 $height
8.60.4.5 $render
8.60.4.6 $renderIcon
8.60.4.7 $renderInput
8.60.4.8 $renderLabel
8.60.4.9 $scope
8.60.4.10 $setSize
8.60.4.11 $setValue
8.60.4.12 $skin
8.60.4.13 $touchCapture
8.60.4.14 $view
8.60.4.15 $width
8.60.4.16 config
8.60.4.17 name
8.60.4.18 on_click
8.60.4.19 touchable
8.61 ui.resizer
8.61.1 Methods
8.61.1.1 adjust
8.61.1.2 bind
8.61.1.3 define
8.61.1.4 destructor
8.61.1.5 disable
8.61.1.6 enable
8.61.1.7 getChildViews
8.61.1.8 getFormView
8.61.1.9 getNode
8.61.1.10 getParentView
8.61.1.11 getTopParentView
8.61.1.12 hide
8.61.1.13 isEnabled
8.61.1.14 isVisible
8.61.1.15 resize

8.61.1.16 show
8.61.1.17 unbind
8.61.2 Properties
8.61.2.1 animate
8.61.2.2 borderless
8.61.2.3 container
8.61.2.4 css
8.61.2.5 disabled
8.61.2.6 gravity
8.61.2.7 height
8.61.2.8 hidden
8.61.2.9 id
8.61.2.10 maxHeight
8.61.2.11 maxWidth
8.61.2.12 minHeight
8.61.2.13 minWidth
8.61.2.14 mouseEventDelay
8.61.2.15 onClick
8.61.2.16 onContext
8.61.2.17 onDblClick
8.61.2.18 onMouseMove
8.61.2.19 width
8.61.3 Events
8.61.3.1 onAfterContextMenu
8.61.3.2 onAfterScroll
8.61.3.3 onBeforeContextMenu
8.61.3.4 onBindRequest
8.61.3.5 onDestruct
8.61.3.6 onItemClick
8.61.3.7 onItemDblClick
8.61.3.8 onKeyPress
8.61.3.9 onLongTouch
8.61.3.10 onMouseMove
8.61.3.11 onMouseMoving
8.61.3.12 onMouseOut
8.61.3.13 onSwipeX
8.61.3.14 onSwipeY
8.61.3.15 onTimedKeyPress
8.61.3.16 onTouchEnd
8.61.3.17 onTouchMove
8.61.3.18 onTouchStart
8.61.3.19 onViewResize
8.61.4 Others
8.61.4.1 $getSize
8.61.4.2 $height
8.61.4.3 $scope
8.61.4.4 $setSize
8.61.4.5 $skin
8.61.4.6 $view
8.61.4.7 $width
8.61.4.8 config
8.61.4.9 name
8.61.4.10 on_click
8.61.4.11 on_context
8.61.4.12 on_dblclick
8.61.4.13 on_mouse_move
8.62 ui.richselect
8.62.1 Methods

8.62.1.1 adjust
8.62.1.2 attachEvent
8.62.1.3 bind
8.62.1.4 blockEvent
8.62.1.5 blur
8.62.1.6 callEvent
8.62.1.7 define
8.62.1.8 destructor
8.62.1.9 detachEvent
8.62.1.10 disable
8.62.1.11 enable
8.62.1.12 focus
8.62.1.13 getChildViews
8.62.1.14 getFormView
8.62.1.15 getInputNode
8.62.1.16 getNode
8.62.1.17 getParentView
8.62.1.18 getPopup
8.62.1.19 getText
8.62.1.20 getTopParentView
8.62.1.21 getValue
8.62.1.22 hasEvent
8.62.1.23 hide
8.62.1.24 isEnabled
8.62.1.25 isVisible
8.62.1.26 mapEvent
8.62.1.27 refresh
8.62.1.28 render
8.62.1.29 resize
8.62.1.30 setValue
8.62.1.31 show
8.62.1.32 sync
8.62.1.33 unbind
8.62.1.34 unblockEvent
8.62.1.35 validate
8.62.2 Properties
8.62.2.1 align
8.62.2.2 animate
8.62.2.3 attributes
8.62.2.4 borderless
8.62.2.5 bottomLabel
8.62.2.6 bottomPadding
8.62.2.7 click
8.62.2.8 container
8.62.2.9 content
8.62.2.10 css
8.62.2.11 disabled
8.62.2.12 gravity
8.62.2.13 height
8.62.2.14 hidden
8.62.2.15 hotkey
8.62.2.16 icon
8.62.2.17 id
8.62.2.18 inputHeight
8.62.2.19 inputPadding
8.62.2.20 inputWidth
8.62.2.21 invalidMessage
8.62.2.22 label

8.62.2.23 labelPosition
8.62.2.24 labelWidth
8.62.2.25 maxHeight
8.62.2.26 maxWidth
8.62.2.27 minHeight
8.62.2.28 minWidth
8.62.2.29 name
8.62.2.30 on
8.62.2.31 options
8.62.2.32 placeholder
8.62.2.33 popup
8.62.2.34 popupWidth
8.62.2.35 readonly
8.62.2.36 relatedAction
8.62.2.37 relatedView
8.62.2.38 required
8.62.2.39 suggest
8.62.2.40 tabFocus
8.62.2.41 template
8.62.2.42 text
8.62.2.43 tooltip
8.62.2.44 type
8.62.2.45 validate
8.62.2.46 validateEvent
8.62.2.47 value
8.62.2.48 width
8.62.3 Events
8.62.3.1 onAfterRender
8.62.3.2 onAfterScroll
8.62.3.3 onBeforeRender
8.62.3.4 onBindRequest
8.62.3.5 onChange
8.62.3.6 onDestruct
8.62.3.7 onItemClick
8.62.3.8 onKeyPress
8.62.3.9 onLongTouch
8.62.3.10 onSwipeX
8.62.3.11 onSwipeY
8.62.3.12 onTimedKeyPress
8.62.3.13 onTouchEnd
8.62.3.14 onTouchMove
8.62.3.15 onTouchStart
8.62.3.16 onViewResize
8.62.4 Others
8.62.4.1 $getSize
8.62.4.2 $getValue
8.62.4.3 $height
8.62.4.4 $render
8.62.4.5 $renderIcon
8.62.4.6 $renderInput
8.62.4.7 $renderLabel
8.62.4.8 $scope
8.62.4.9 $setSize
8.62.4.10 $setValue
8.62.4.11 $skin
8.62.4.12 $view
8.62.4.13 $width
8.62.4.14 config

8.62.4.15 name
8.62.4.16 on_click
8.62.4.17 touchable
8.63 ui.scrollview
8.63.1 Methods
8.63.1.1 adjust
8.63.1.2 bind
8.63.1.3 define
8.63.1.4 destructor
8.63.1.5 disable
8.63.1.6 enable
8.63.1.7 getBody
8.63.1.8 getChildViews
8.63.1.9 getFormView
8.63.1.10 getNode
8.63.1.11 getParentView
8.63.1.12 getScrollState
8.63.1.13 getTopParentView
8.63.1.14 hide
8.63.1.15 isEnabled
8.63.1.16 isVisible
8.63.1.17 resize
8.63.1.18 resizeChildren
8.63.1.19 scrollTo
8.63.1.20 show
8.63.1.21 showView
8.63.1.22 unbind
8.63.2 Properties
8.63.2.1 animate
8.63.2.2 body
8.63.2.3 borderless
8.63.2.4 container
8.63.2.5 css
8.63.2.6 disabled
8.63.2.7 gravity
8.63.2.8 height
8.63.2.9 hidden
8.63.2.10 id
8.63.2.11 maxHeight
8.63.2.12 maxWidth
8.63.2.13 minHeight
8.63.2.14 minWidth
8.63.2.15 scroll
8.63.2.16 scrollSpeed
8.63.2.17 width
8.63.3 Events
8.63.3.1 onAfterScroll
8.63.3.2 onBindRequest
8.63.3.3 onDestruct
8.63.3.4 onKeyPress
8.63.3.5 onLongTouch
8.63.3.6 onSwipeX
8.63.3.7 onSwipeY
8.63.3.8 onTimedKeyPress
8.63.3.9 onTouchEnd
8.63.3.10 onTouchMove
8.63.3.11 onTouchStart
8.63.3.12 onViewResize

8.63.4 Others
8.63.4.1 $getSize
8.63.4.2 $height
8.63.4.3 $scope
8.63.4.4 $setSize
8.63.4.5 $skin
8.63.4.6 $view
8.63.4.7 $width
8.63.4.8 config
8.63.4.9 name
8.64 ui.search
8.64.1 Methods
8.64.1.1 adjust
8.64.1.2 attachEvent
8.64.1.3 bind
8.64.1.4 blockEvent
8.64.1.5 blur
8.64.1.6 callEvent
8.64.1.7 define
8.64.1.8 destructor
8.64.1.9 detachEvent
8.64.1.10 disable
8.64.1.11 enable
8.64.1.12 focus
8.64.1.13 getChildViews
8.64.1.14 getFormView
8.64.1.15 getInputNode
8.64.1.16 getNode
8.64.1.17 getParentView
8.64.1.18 getTopParentView
8.64.1.19 getValue
8.64.1.20 hasEvent
8.64.1.21 hide
8.64.1.22 isEnabled
8.64.1.23 isVisible
8.64.1.24 mapEvent
8.64.1.25 refresh
8.64.1.26 render
8.64.1.27 resize
8.64.1.28 setValue
8.64.1.29 show
8.64.1.30 sync
8.64.1.31 unbind
8.64.1.32 unblockEvent
8.64.1.33 validate
8.64.2 Properties
8.64.2.1 align
8.64.2.2 animate
8.64.2.3 attributes
8.64.2.4 borderless
8.64.2.5 bottomLabel
8.64.2.6 bottomPadding
8.64.2.7 click
8.64.2.8 container
8.64.2.9 content
8.64.2.10 css
8.64.2.11 disabled
8.64.2.12 gravity

8.64.2.13 height
8.64.2.14 hidden
8.64.2.15 hotkey
8.64.2.16 icon
8.64.2.17 id
8.64.2.18 inputHeight
8.64.2.19 inputPadding
8.64.2.20 inputWidth
8.64.2.21 invalidMessage
8.64.2.22 label
8.64.2.23 labelPosition
8.64.2.24 labelWidth
8.64.2.25 maxHeight
8.64.2.26 maxWidth
8.64.2.27 minHeight
8.64.2.28 minWidth
8.64.2.29 name
8.64.2.30 on
8.64.2.31 placeholder
8.64.2.32 popup
8.64.2.33 readonly
8.64.2.34 relatedAction
8.64.2.35 relatedView
8.64.2.36 required
8.64.2.37 suggest
8.64.2.38 tabFocus
8.64.2.39 template
8.64.2.40 tooltip
8.64.2.41 type
8.64.2.42 validate
8.64.2.43 validateEvent
8.64.2.44 value
8.64.2.45 width
8.64.3 Events
8.64.3.1 onAfterRender
8.64.3.2 onAfterScroll
8.64.3.3 onBeforeRender
8.64.3.4 onBindRequest
8.64.3.5 onChange
8.64.3.6 onDestruct
8.64.3.7 onItemClick
8.64.3.8 onKeyPress
8.64.3.9 onLongTouch
8.64.3.10 onSearchIconClick
8.64.3.11 onSwipeX
8.64.3.12 onSwipeY
8.64.3.13 onTimedKeyPress
8.64.3.14 onTouchEnd
8.64.3.15 onTouchMove
8.64.3.16 onTouchStart
8.64.3.17 onViewResize
8.64.4 Others
8.64.4.1 $getSize
8.64.4.2 $getValue
8.64.4.3 $height
8.64.4.4 $render
8.64.4.5 $renderIcon
8.64.4.6 $renderInput

8.64.4.7 $renderLabel
8.64.4.8 $scope
8.64.4.9 $setSize
8.64.4.10 $setValue
8.64.4.11 $skin
8.64.4.12 $view
8.64.4.13 $width
8.64.4.14 config
8.64.4.15 name
8.64.4.16 on_click
8.64.4.17 touchable

8.65 ui.segmented

8.65.1 Methods

8.65.1.1 addOption
8.65.1.2 adjust
8.65.1.3 attachEvent
8.65.1.4 bind
8.65.1.5 blockEvent
8.65.1.6 blur
8.65.1.7 callEvent
8.65.1.8 define
8.65.1.9 destructor
8.65.1.10 detachEvent
8.65.1.11 disable
8.65.1.12 enable
8.65.1.13 focus
8.65.1.14 getChildViews
8.65.1.15 getFormView
8.65.1.16 getInputNode
8.65.1.17 getNode
8.65.1.18 getParentView
8.65.1.19 getTopParentView
8.65.1.20 getValue
8.65.1.21 hasEvent
8.65.1.22 hide
8.65.1.23 isEnabled
8.65.1.24 isVisible
8.65.1.25 mapEvent
8.65.1.26 optionIndex
8.65.1.27 refresh
8.65.1.28 removeOption
8.65.1.29 render
8.65.1.30 resize
8.65.1.31 setValue
8.65.1.32 show
8.65.1.33 sync
8.65.1.34 unbind
8.65.1.35 unblockEvent
8.65.1.36 validate

8.65.2 Properties

8.65.2.1 align
8.65.2.2 animate
8.65.2.3 attributes
8.65.2.4 borderless
8.65.2.5 bottomLabel
8.65.2.6 bottomPadding
8.65.2.7 click
8.65.2.8 container

8.65.2.9 content
8.65.2.10 css
8.65.2.11 disabled
8.65.2.12 gravity
8.65.2.13 height
8.65.2.14 hidden
8.65.2.15 hotkey
8.65.2.16 id
8.65.2.17 inputHeight
8.65.2.18 inputPadding
8.65.2.19 inputWidth
8.65.2.20 invalidMessage
8.65.2.21 label
8.65.2.22 labelPosition
8.65.2.23 labelWidth
8.65.2.24 maxHeight
8.65.2.25 maxWidth
8.65.2.26 minHeight
8.65.2.27 minWidth
8.65.2.28 multiview
8.65.2.29 name
8.65.2.30 on
8.65.2.31 options
8.65.2.32 placeholder
8.65.2.33 popup
8.65.2.34 readonly
8.65.2.35 relatedAction
8.65.2.36 relatedView
8.65.2.37 required
8.65.2.38 suggest
8.65.2.39 tabFocus
8.65.2.40 template
8.65.2.41 tooltip
8.65.2.42 type
8.65.2.43 validate
8.65.2.44 validateEvent
8.65.2.45 value
8.65.2.46 width
8.65.3 Events
8.65.3.1 onAfterRender
8.65.3.2 onAfterScroll
8.65.3.3 onAfterTabClick
8.65.3.4 onBeforeRender
8.65.3.5 onBeforeTabClick
8.65.3.6 onBindRequest
8.65.3.7 onChange
8.65.3.8 onDestruct
8.65.3.9 onItemClick
8.65.3.10 onKeyPress
8.65.3.11 onLongTouch
8.65.3.12 onOptionRemove
8.65.3.13 onSwipeX
8.65.3.14 onSwipeY
8.65.3.15 onTimedKeyPress
8.65.3.16 onTouchEnd
8.65.3.17 onTouchMove
8.65.3.18 onTouchStart
8.65.3.19 onViewResize

8.65.4 Others
8.65.4.1 $getSize
8.65.4.2 $getValue
8.65.4.3 $height
8.65.4.4 $render
8.65.4.5 $renderIcon
8.65.4.6 $renderInput
8.65.4.7 $renderLabel
8.65.4.8 $scope
8.65.4.9 $setSize
8.65.4.10 $setValue
8.65.4.11 $skin
8.65.4.12 $view
8.65.4.13 $width
8.65.4.14 config
8.65.4.15 name
8.65.4.16 on_click
8.65.4.17 touchable
8.66 ui.select
8.66.1 Methods
8.66.1.1 adjust
8.66.1.2 attachEvent
8.66.1.3 bind
8.66.1.4 blockEvent
8.66.1.5 blur
8.66.1.6 callEvent
8.66.1.7 define
8.66.1.8 destructor
8.66.1.9 detachEvent
8.66.1.10 disable
8.66.1.11 enable
8.66.1.12 focus
8.66.1.13 getChildViews
8.66.1.14 getFormView
8.66.1.15 getInputNode
8.66.1.16 getNode
8.66.1.17 getParentView
8.66.1.18 getTopParentView
8.66.1.19 getValue
8.66.1.20 hasEvent
8.66.1.21 hide
8.66.1.22 isEnabled
8.66.1.23 isVisible
8.66.1.24 mapEvent
8.66.1.25 refresh
8.66.1.26 render
8.66.1.27 resize
8.66.1.28 setValue
8.66.1.29 show
8.66.1.30 sync
8.66.1.31 unbind
8.66.1.32 unblockEvent
8.66.1.33 validate
8.66.2 Properties
8.66.2.1 align
8.66.2.2 animate
8.66.2.3 attributes
8.66.2.4 borderless

8.66.2.5 bottomLabel
8.66.2.6 bottomPadding
8.66.2.7 click
8.66.2.8 container
8.66.2.9 content
8.66.2.10 css
8.66.2.11 disabled
8.66.2.12 gravity
8.66.2.13 height
8.66.2.14 hidden
8.66.2.15 hotkey
8.66.2.16 id
8.66.2.17 inputHeight
8.66.2.18 inputPadding
8.66.2.19 inputWidth
8.66.2.20 invalidMessage
8.66.2.21 label
8.66.2.22 labelPosition
8.66.2.23 labelWidth
8.66.2.24 maxHeight
8.66.2.25 maxWidth
8.66.2.26 minHeight
8.66.2.27 minWidth
8.66.2.28 name
8.66.2.29 on
8.66.2.30 options
8.66.2.31 placeholder
8.66.2.32 popup
8.66.2.33 readonly
8.66.2.34 relatedAction
8.66.2.35 relatedView
8.66.2.36 required
8.66.2.37 suggest
8.66.2.38 tabFocus
8.66.2.39 template
8.66.2.40 tooltip
8.66.2.41 type
8.66.2.42 validate
8.66.2.43 validateEvent
8.66.2.44 value
8.66.2.45 width
8.66.3 Events
8.66.3.1 onAfterRender
8.66.3.2 onAfterScroll
8.66.3.3 onBeforeRender
8.66.3.4 onBindRequest
8.66.3.5 onChange
8.66.3.6 onDestruct
8.66.3.7 onItemClick
8.66.3.8 onKeyPress
8.66.3.9 onLongTouch
8.66.3.10 onSwipeX
8.66.3.11 onSwipeY
8.66.3.12 onTimedKeyPress
8.66.3.13 onTouchEnd
8.66.3.14 onTouchMove
8.66.3.15 onTouchStart
8.66.3.16 onViewResize

8.66.4 Others
8.66.4.1 $getSize
8.66.4.2 $getValue
8.66.4.3 $height
8.66.4.4 $render
8.66.4.5 $renderIcon
8.66.4.6 $renderInput
8.66.4.7 $renderLabel
8.66.4.8 $scope
8.66.4.9 $setSize
8.66.4.10 $setValue
8.66.4.11 $skin
8.66.4.12 $view
8.66.4.13 $width
8.66.4.14 config
8.66.4.15 name
8.66.4.16 on_click
8.66.4.17 touchable
8.67 ui.slider
8.67.1 Methods
8.67.1.1 adjust
8.67.1.2 attachEvent
8.67.1.3 bind
8.67.1.4 blockEvent
8.67.1.5 blur
8.67.1.6 callEvent
8.67.1.7 define
8.67.1.8 destructor
8.67.1.9 detachEvent
8.67.1.10 disable
8.67.1.11 enable
8.67.1.12 focus
8.67.1.13 getChildViews
8.67.1.14 getFormView
8.67.1.15 getInputNode
8.67.1.16 getNode
8.67.1.17 getParentView
8.67.1.18 getTopParentView
8.67.1.19 getValue
8.67.1.20 hasEvent
8.67.1.21 hide
8.67.1.22 isEnabled
8.67.1.23 isVisible
8.67.1.24 mapEvent
8.67.1.25 refresh
8.67.1.26 render
8.67.1.27 resize
8.67.1.28 setValue
8.67.1.29 show
8.67.1.30 sync
8.67.1.31 unbind
8.67.1.32 unblockEvent
8.67.1.33 validate
8.67.2 Properties
8.67.2.1 align
8.67.2.2 animate
8.67.2.3 attributes
8.67.2.4 borderless

8.67.2.5 bottomLabel
8.67.2.6 bottomPadding
8.67.2.7 click
8.67.2.8 container
8.67.2.9 content
8.67.2.10 css
8.67.2.11 disabled
8.67.2.12 gravity
8.67.2.13 height
8.67.2.14 hidden
8.67.2.15 hotkey
8.67.2.16 id
8.67.2.17 inputHeight
8.67.2.18 inputPadding
8.67.2.19 inputWidth
8.67.2.20 invalidMessage
8.67.2.21 label
8.67.2.22 labelPosition
8.67.2.23 labelWidth
8.67.2.24 max
8.67.2.25 maxHeight
8.67.2.26 maxWidth
8.67.2.27 min
8.67.2.28 minHeight
8.67.2.29 minWidth
8.67.2.30 name
8.67.2.31 on
8.67.2.32 placeholder
8.67.2.33 popup
8.67.2.34 readonly
8.67.2.35 relatedAction
8.67.2.36 relatedView
8.67.2.37 required
8.67.2.38 step
8.67.2.39 suggest
8.67.2.40 tabFocus
8.67.2.41 template
8.67.2.42 title
8.67.2.43 tooltip
8.67.2.44 type
8.67.2.45 validate
8.67.2.46 validateEvent
8.67.2.47 value
8.67.2.48 width
8.67.3 Events
8.67.3.1 onAfterRender
8.67.3.2 onAfterScroll
8.67.3.3 onBeforeRender
8.67.3.4 onBindRequest
8.67.3.5 onChange
8.67.3.6 onDestruct
8.67.3.7 onItemClick
8.67.3.8 onKeyPress
8.67.3.9 onLongTouch
8.67.3.10 onSliderDrag
8.67.3.11 onSwipeX
8.67.3.12 onSwipeY
8.67.3.13 onTimedKeyPress

8.67.3.14 onTouchEnd
8.67.3.15 onTouchMove
8.67.3.16 onTouchStart
8.67.3.17 onViewResize
8.67.4 Others
8.67.4.1 $getSize
8.67.4.2 $getValue
8.67.4.3 $height
8.67.4.4 $render
8.67.4.5 $renderIcon
8.67.4.6 $renderInput
8.67.4.7 $renderLabel
8.67.4.8 $scope
8.67.4.9 $setSize
8.67.4.10 $setValue
8.67.4.11 $skin
8.67.4.12 $touchCapture
8.67.4.13 $view
8.67.4.14 $width
8.67.4.15 config
8.67.4.16 name
8.67.4.17 on_click
8.67.4.18 touchable
8.68 ui.spacer
8.68.1 Methods
8.68.1.1 adjust
8.68.1.2 bind
8.68.1.3 define
8.68.1.4 destructor
8.68.1.5 disable
8.68.1.6 enable
8.68.1.7 getChildViews
8.68.1.8 getFormView
8.68.1.9 getNode
8.68.1.10 getParentView
8.68.1.11 getTopParentView
8.68.1.12 hide
8.68.1.13 isEnabled
8.68.1.14 isVisible
8.68.1.15 resize
8.68.1.16 show
8.68.1.17 unbind
8.68.2 Properties
8.68.2.1 animate
8.68.2.2 borderless
8.68.2.3 container
8.68.2.4 css
8.68.2.5 disabled
8.68.2.6 gravity
8.68.2.7 height
8.68.2.8 hidden
8.68.2.9 id
8.68.2.10 maxHeight
8.68.2.11 maxWidth
8.68.2.12 minHeight
8.68.2.13 minWidth
8.68.2.14 width
8.68.3 Events

8.68.3.1 onAfterScroll
8.68.3.2 onBindRequest
8.68.3.3 onDestruct
8.68.3.4 onKeyPress
8.68.3.5 onLongTouch
8.68.3.6 onSwipeX
8.68.3.7 onSwipeY
8.68.3.8 onTimedKeyPress
8.68.3.9 onTouchEnd
8.68.3.10 onTouchMove
8.68.3.11 onTouchStart
8.68.3.12 onViewResize
8.68.4 Others
8.68.4.1 $getSize
8.68.4.2 $height
8.68.4.3 $scope
8.68.4.4 $setSize
8.68.4.5 $skin
8.68.4.6 $view
8.68.4.7 $width
8.68.4.8 config
8.68.4.9 name
8.69 ui.submenu
8.69.1 Methods
8.69.1.1 add
8.69.1.2 addCss
8.69.1.3 adjust
8.69.1.4 attachEvent
8.69.1.5 bind
8.69.1.6 blockEvent
8.69.1.7 callEvent
8.69.1.8 clearAll
8.69.1.9 clearCss
8.69.1.10 clearValidation
8.69.1.11 close
8.69.1.12 copy
8.69.1.13 count
8.69.1.14 customize
8.69.1.15 define
8.69.1.16 destructor
8.69.1.17 detachEvent
8.69.1.18 disable
8.69.1.19 disableItem
8.69.1.20 enable
8.69.1.21 enableItem
8.69.1.22 exists
8.69.1.23 filter
8.69.1.24 getBody
8.69.1.25 getChildViews
8.69.1.26 getFirstId
8.69.1.27 getFormView
8.69.1.28 getHead
8.69.1.29 getIdByIndex
8.69.1.30 getIndexById
8.69.1.31 getItem
8.69.1.32 getItemNode
8.69.1.33 getLastId
8.69.1.34 getMenu

8.69.1.35 getMenuItem
8.69.1.36 getNextId
8.69.1.37 getNode
8.69.1.38 getPage
8.69.1.39 getPager
8.69.1.40 getParentView
8.69.1.41 getPrevId
8.69.1.42 getScrollState
8.69.1.43 getSelectedId
8.69.1.44 getSelectedItem
8.69.1.45 getSubMenu
8.69.1.46 getTopMenu
8.69.1.47 getTopParentView
8.69.1.48 getVisibleCount
8.69.1.49 hasCss
8.69.1.50 hasEvent
8.69.1.51 hide
8.69.1.52 hideItem
8.69.1.53 isEnabled
8.69.1.54 isSelected
8.69.1.55 isVisible
8.69.1.56 load
8.69.1.57 loadNext
8.69.1.58 locate
8.69.1.59 mapEvent
8.69.1.60 move
8.69.1.61 moveBottom
8.69.1.62 moveDown
8.69.1.63 moveSelection
8.69.1.64 moveTop
8.69.1.65 moveUp
8.69.1.66 parse
8.69.1.67 refresh
8.69.1.68 remove
8.69.1.69 removeCss
8.69.1.70 render
8.69.1.71 resize
8.69.1.72 resizeChildren
8.69.1.73 scrollTo
8.69.1.74 select
8.69.1.75 selectAll
8.69.1.76 serialize
8.69.1.77 setPage
8.69.1.78 setPosition
8.69.1.79 show
8.69.1.80 showItem
8.69.1.81 sort
8.69.1.82 sync
8.69.1.83 unbind
8.69.1.84 unblockEvent
8.69.1.85 unselect
8.69.1.86 unselectAll
8.69.1.87 updateItem
8.69.1.88 validate
8.69.2 Properties
8.69.2.1 animate
8.69.2.2 autofit
8.69.2.3 autofocus

8.69.2.4 autoheight
8.69.2.5 autowidth
8.69.2.6 body
8.69.2.7 borderless
8.69.2.8 click
8.69.2.9 clipboard
8.69.2.10 container
8.69.2.11 css
8.69.2.12 data
8.69.2.13 dataFeed
8.69.2.14 datathrottle
8.69.2.15 datatype
8.69.2.16 disabled
8.69.2.17 drag
8.69.2.18 dragscroll
8.69.2.19 externalData
8.69.2.20 gravity
8.69.2.21 head
8.69.2.22 headHeight
8.69.2.23 height
8.69.2.24 hidden
8.69.2.25 id
8.69.2.26 layout
8.69.2.27 left
8.69.2.28 maxHeight
8.69.2.29 maxWidth
8.69.2.30 minHeight
8.69.2.31 minWidth
8.69.2.32 modal
8.69.2.33 mouseEventDelay
8.69.2.34 move
8.69.2.35 multiselect
8.69.2.36 navigation
8.69.2.37 on
8.69.2.38 onClick
8.69.2.39 onContext
8.69.2.40 onDblClick
8.69.2.41 onMouseMove
8.69.2.42 padding
8.69.2.43 pager
8.69.2.44 position
8.69.2.45 ready
8.69.2.46 relative
8.69.2.47 removeMissed
8.69.2.48 rules
8.69.2.49 save
8.69.2.50 scheme
8.69.2.51 scroll
8.69.2.52 scrollSpeed
8.69.2.53 select
8.69.2.54 subMenuPos
8.69.2.55 submenu
8.69.2.56 template
8.69.2.57 templateCopy
8.69.2.58 tooltip
8.69.2.59 top
8.69.2.60 type
8.69.2.61 url

8.69.2.62 width
8.69.2.63 xCount
8.69.2.64 yCount
8.69.2.65 zIndex
8.69.3 Events
8.69.3.1 onAfterAdd
8.69.3.2 onAfterContextMenu
8.69.3.3 onAfterDelete
8.69.3.4 onAfterDrop
8.69.3.5 onAfterLoad
8.69.3.6 onAfterRender
8.69.3.7 onAfterScroll
8.69.3.8 onAfterSelect
8.69.3.9 onAfterSort
8.69.3.10 onBeforeAdd
8.69.3.11 onBeforeContextMenu
8.69.3.12 onBeforeDelete
8.69.3.13 onBeforeDrag
8.69.3.14 onBeforeDragIn
8.69.3.15 onBeforeDrop
8.69.3.16 onBeforeDropOut
8.69.3.17 onBeforeLoad
8.69.3.18 onBeforeRender
8.69.3.19 onBeforeSelect
8.69.3.20 onBeforeShow
8.69.3.21 onBeforeSort
8.69.3.22 onBindRequest
8.69.3.23 onDataRequest
8.69.3.24 onDataUpdate
8.69.3.25 onDestruct
8.69.3.26 onDragOut
8.69.3.27 onHide
8.69.3.28 onItemClick
8.69.3.29 onItemDblClick
8.69.3.30 onItemRender
8.69.3.31 onKeyPress
8.69.3.32 onLoadError
8.69.3.33 onLongTouch
8.69.3.34 onMenuItemClick
8.69.3.35 onMouseMove
8.69.3.36 onMouseMoving
8.69.3.37 onMouseOut
8.69.3.38 onPaste
8.69.3.39 onSelectChange
8.69.3.40 onShow
8.69.3.41 onSwipeX
8.69.3.42 onSwipeY
8.69.3.43 onTimedKeyPress
8.69.3.44 onTouchEnd
8.69.3.45 onTouchMove
8.69.3.46 onTouchStart
8.69.3.47 onValidationError
8.69.3.48 onValidationSuccess
8.69.3.49 onViewMove
8.69.3.50 onViewMoveEnd
8.69.3.51 onViewResize
8.69.4 Others
8.69.4.1 $drag

8.69.4.2 $dragHTML
8.69.4.3 $dragIn
8.69.4.4 $dragMark
8.69.4.5 $dragOut
8.69.4.6 $drop
8.69.4.7 $dropAllow
8.69.4.8 $getSize
8.69.4.9 $height
8.69.4.10 $scope
8.69.4.11 $setSize
8.69.4.12 $skin
8.69.4.13 $view
8.69.4.14 $width
8.69.4.15 config
8.69.4.16 name
8.69.4.17 on_click
8.69.4.18 on_context
8.69.4.19 on_dblclick
8.69.4.20 on_mouse_move
8.69.4.21 type
8.69.4.22 types
8.70 ui.suggest
8.70.1 Methods
8.70.1.1 adjust
8.70.1.2 attachEvent
8.70.1.3 bind
8.70.1.4 blockEvent
8.70.1.5 callEvent
8.70.1.6 close
8.70.1.7 define
8.70.1.8 destructor
8.70.1.9 detachEvent
8.70.1.10 disable
8.70.1.11 enable
8.70.1.12 getBody
8.70.1.13 getChildViews
8.70.1.14 getFormView
8.70.1.15 getHead
8.70.1.16 getItemText
8.70.1.17 getList
8.70.1.18 getMasterValue
8.70.1.19 getNode
8.70.1.20 getParentView
8.70.1.21 getSuggestion
8.70.1.22 getTopParentView
8.70.1.23 getValue
8.70.1.24 hasEvent
8.70.1.25 hide
8.70.1.26 isEnabled
8.70.1.27 isVisible
8.70.1.28 linkInput
8.70.1.29 mapEvent
8.70.1.30 resize
8.70.1.31 resizeChildren
8.70.1.32 setMasterValue
8.70.1.33 setPosition
8.70.1.34 setValue
8.70.1.35 show

8.70.1.36 unbind
8.70.1.37 unblockEvent
8.70.2 Properties
8.70.2.1 animate
8.70.2.2 autofit
8.70.2.3 autofocus
8.70.2.4 body
8.70.2.5 borderless
8.70.2.6 container
8.70.2.7 css
8.70.2.8 disabled
8.70.2.9 filter
8.70.2.10 fitMaster
8.70.2.11 gravity
8.70.2.12 head
8.70.2.13 headHeight
8.70.2.14 height
8.70.2.15 hidden
8.70.2.16 id
8.70.2.17 input
8.70.2.18 keyPressTimeout
8.70.2.19 left
8.70.2.20 master
8.70.2.21 maxHeight
8.70.2.22 maxWidth
8.70.2.23 minHeight
8.70.2.24 minWidth
8.70.2.25 modal
8.70.2.26 move
8.70.2.27 on
8.70.2.28 padding
8.70.2.29 position
8.70.2.30 relative
8.70.2.31 template
8.70.2.32 textValue
8.70.2.33 top
8.70.2.34 type
8.70.2.35 width
8.70.2.36 zIndex
8.70.3 Events
8.70.3.1 onAfterScroll
8.70.3.2 onBeforeShow
8.70.3.3 onBindRequest
8.70.3.4 onDestruct
8.70.3.5 onHide
8.70.3.6 onKeyPress
8.70.3.7 onLongTouch
8.70.3.8 onShow
8.70.3.9 onSwipeX
8.70.3.10 onSwipeY
8.70.3.11 onTimedKeyPress
8.70.3.12 onTouchEnd
8.70.3.13 onTouchMove
8.70.3.14 onTouchStart
8.70.3.15 onValueSuggest
8.70.3.16 onViewMove
8.70.3.17 onViewMoveEnd
8.70.3.18 onViewResize

8.70.4 Others
8.70.4.1 $getSize
8.70.4.2 $height
8.70.4.3 $scope
8.70.4.4 $setSize
8.70.4.5 $skin
8.70.4.6 $view
8.70.4.7 $width
8.70.4.8 config
8.70.4.9 name
8.71 ui.tabbar
8.71.1 Methods
8.71.1.1 addOption
8.71.1.2 adjust
8.71.1.3 attachEvent
8.71.1.4 bind
8.71.1.5 blockEvent
8.71.1.6 blur
8.71.1.7 callEvent
8.71.1.8 define
8.71.1.9 destructor
8.71.1.10 detachEvent
8.71.1.11 disable
8.71.1.12 enable
8.71.1.13 focus
8.71.1.14 getChildViews
8.71.1.15 getFormView
8.71.1.16 getInputNode
8.71.1.17 getNode
8.71.1.18 getParentView
8.71.1.19 getPopup
8.71.1.20 getTopParentView
8.71.1.21 getValue
8.71.1.22 hasEvent
8.71.1.23 hide
8.71.1.24 isEnabled
8.71.1.25 isVisible
8.71.1.26 mapEvent
8.71.1.27 optionIndex
8.71.1.28 refresh
8.71.1.29 removeOption
8.71.1.30 render
8.71.1.31 resize
8.71.1.32 setValue
8.71.1.33 show
8.71.1.34 sync
8.71.1.35 unbind
8.71.1.36 unblockEvent
8.71.1.37 validate
8.71.2 Properties
8.71.2.1 align
8.71.2.2 animate
8.71.2.3 attributes
8.71.2.4 borderless
8.71.2.5 bottomLabel
8.71.2.6 bottomOffset
8.71.2.7 bottomPadding
8.71.2.8 click

8.71.2.9 container
8.71.2.10 content
8.71.2.11 css
8.71.2.12 disabled
8.71.2.13 gravity
8.71.2.14 height
8.71.2.15 hidden
8.71.2.16 hotkey
8.71.2.17 id
8.71.2.18 inputHeight
8.71.2.19 inputPadding
8.71.2.20 inputWidth
8.71.2.21 invalidMessage
8.71.2.22 label
8.71.2.23 labelPosition
8.71.2.24 labelWidth
8.71.2.25 maxHeight
8.71.2.26 maxWidth
8.71.2.27 minHeight
8.71.2.28 minWidth
8.71.2.29 moreTemplate
8.71.2.30 multiview
8.71.2.31 name
8.71.2.32 on
8.71.2.33 options
8.71.2.34 placeholder
8.71.2.35 popup
8.71.2.36 popupTemplate
8.71.2.37 popupWidth
8.71.2.38 readonly
8.71.2.39 relatedAction
8.71.2.40 relatedView
8.71.2.41 required
8.71.2.42 suggest
8.71.2.43 tabFocus
8.71.2.44 tabMargin
8.71.2.45 tabMinWidth
8.71.2.46 tabMoreWidth
8.71.2.47 tabOffset
8.71.2.48 tabbarPopup
8.71.2.49 template
8.71.2.50 tooltip
8.71.2.51 topOffset
8.71.2.52 type
8.71.2.53 validate
8.71.2.54 validateEvent
8.71.2.55 value
8.71.2.56 width
8.71.2.57 yCount
8.71.3 Events
8.71.3.1 onAfterRender
8.71.3.2 onAfterScroll
8.71.3.3 onAfterTabClick
8.71.3.4 onBeforeRender
8.71.3.5 onBeforeTabClick
8.71.3.6 onBeforeTabClose
8.71.3.7 onBindRequest
8.71.3.8 onChange

8.71.3.9 onDestruct
8.71.3.10 onItemClick
8.71.3.11 onKeyPress
8.71.3.12 onLongTouch
8.71.3.13 onOptionRemove
8.71.3.14 onSwipeX
8.71.3.15 onSwipeY
8.71.3.16 onTimedKeyPress
8.71.3.17 onTouchEnd
8.71.3.18 onTouchMove
8.71.3.19 onTouchStart
8.71.3.20 onViewResize
8.71.4 Others
8.71.4.1 $getSize
8.71.4.2 $getValue
8.71.4.3 $height
8.71.4.4 $render
8.71.4.5 $renderIcon
8.71.4.6 $renderInput
8.71.4.7 $renderLabel
8.71.4.8 $scope
8.71.4.9 $setSize
8.71.4.10 $setValue
8.71.4.11 $skin
8.71.4.12 $view
8.71.4.13 $width
8.71.4.14 config
8.71.4.15 name
8.71.4.16 on_click
8.71.4.17 touchable
8.72 ui.tabview
8.72.1 Methods
8.72.1.1 addView
8.72.1.2 adjust
8.72.1.3 attachEvent
8.72.1.4 bind
8.72.1.5 blockEvent
8.72.1.6 callEvent
8.72.1.7 define
8.72.1.8 destructor
8.72.1.9 detachEvent
8.72.1.10 disable
8.72.1.11 enable
8.72.1.12 getChildViews
8.72.1.13 getFormView
8.72.1.14 getMultiview
8.72.1.15 getNode
8.72.1.16 getParentView
8.72.1.17 getTabbar
8.72.1.18 getTopParentView
8.72.1.19 getValue
8.72.1.20 hasEvent
8.72.1.21 hide
8.72.1.22 index
8.72.1.23 isEnabled
8.72.1.24 isVisible
8.72.1.25 mapEvent
8.72.1.26 reconstruct

8.72.1.27 removeView
8.72.1.28 resize
8.72.1.29 resizeChildren
8.72.1.30 setValue
8.72.1.31 show
8.72.1.32 showBatch
8.72.1.33 unbind
8.72.1.34 unblockEvent
8.72.2 Properties
8.72.2.1 animate
8.72.2.2 borderless
8.72.2.3 cells
8.72.2.4 cols
8.72.2.5 container
8.72.2.6 css
8.72.2.7 disabled
8.72.2.8 gravity
8.72.2.9 height
8.72.2.10 hidden
8.72.2.11 id
8.72.2.12 isolate
8.72.2.13 margin
8.72.2.14 maxHeight
8.72.2.15 maxWidth
8.72.2.16 minHeight
8.72.2.17 minWidth
8.72.2.18 multiview
8.72.2.19 on
8.72.2.20 padding
8.72.2.21 paddingX
8.72.2.22 paddingY
8.72.2.23 responsive
8.72.2.24 rows
8.72.2.25 tabbar
8.72.2.26 type
8.72.2.27 visibleBatch
8.72.2.28 width
8.72.3 Events
8.72.3.1 onBindRequest
8.72.3.2 onDestruct
8.72.4 Others
8.72.4.1 $getSize
8.72.4.2 $height
8.72.4.3 $setSize
8.72.4.4 $skin
8.72.4.5 $view
8.72.4.6 $width
8.72.4.7 config
8.72.4.8 name
8.73 ui.template
8.73.1 Methods
8.73.1.1 adjust
8.73.1.2 attachEvent
8.73.1.3 bind
8.73.1.4 blockEvent
8.73.1.5 callEvent
8.73.1.6 define
8.73.1.7 destructor

8.73.1.8 detachEvent
8.73.1.9 disable
8.73.1.10 enable
8.73.1.11 getChildViews
8.73.1.12 getFormView
8.73.1.13 getNode
8.73.1.14 getParentView
8.73.1.15 getScrollState
8.73.1.16 getTopParentView
8.73.1.17 hasEvent
8.73.1.18 hide
8.73.1.19 isEnabled
8.73.1.20 isVisible
8.73.1.21 load
8.73.1.22 mapEvent
8.73.1.23 parse
8.73.1.24 refresh
8.73.1.25 render
8.73.1.26 resize
8.73.1.27 scrollTo
8.73.1.28 setContent
8.73.1.29 setHTML
8.73.1.30 setValues
8.73.1.31 show
8.73.1.32 sync
8.73.1.33 unbind
8.73.1.34 unblockEvent
8.73.2 Properties
8.73.2.1 animate
8.73.2.2 autoheight
8.73.2.3 borderless
8.73.2.4 container
8.73.2.5 content
8.73.2.6 css
8.73.2.7 data
8.73.2.8 dataFeed
8.73.2.9 datatype
8.73.2.10 disabled
8.73.2.11 gravity
8.73.2.12 height
8.73.2.13 hidden
8.73.2.14 id
8.73.2.15 maxHeight
8.73.2.16 maxWidth
8.73.2.17 minHeight
8.73.2.18 minWidth
8.73.2.19 on
8.73.2.20 onClick
8.73.2.21 scroll
8.73.2.22 scrollSpeed
8.73.2.23 src
8.73.2.24 template
8.73.2.25 type
8.73.2.26 url
8.73.2.27 width
8.73.3 Events
8.73.3.1 onAfterLoad
8.73.3.2 onAfterRender

8.73.3.3 onAfterScroll
8.73.3.4 onBeforeLoad
8.73.3.5 onBeforeRender
8.73.3.6 onBindRequest
8.73.3.7 onDestruct
8.73.3.8 onKeyPress
8.73.3.9 onLoadError
8.73.3.10 onLongTouch
8.73.3.11 onSwipeX
8.73.3.12 onSwipeY
8.73.3.13 onTimedKeyPress
8.73.3.14 onTouchEnd
8.73.3.15 onTouchMove
8.73.3.16 onTouchStart
8.73.3.17 onViewResize
8.73.4 Others
8.73.4.1 $getSize
8.73.4.2 $height
8.73.4.3 $scope
8.73.4.4 $setSize
8.73.4.5 $skin
8.73.4.6 $view
8.73.4.7 $width
8.73.4.8 config
8.73.4.9 name
8.74 ui.text
8.74.1 Methods
8.74.1.1 adjust
8.74.1.2 attachEvent
8.74.1.3 bind
8.74.1.4 blockEvent
8.74.1.5 blur
8.74.1.6 callEvent
8.74.1.7 define
8.74.1.8 destructor
8.74.1.9 detachEvent
8.74.1.10 disable
8.74.1.11 enable
8.74.1.12 focus
8.74.1.13 getChildViews
8.74.1.14 getFormView
8.74.1.15 getInputNode
8.74.1.16 getNode
8.74.1.17 getParentView
8.74.1.18 getTopParentView
8.74.1.19 getValue
8.74.1.20 hasEvent
8.74.1.21 hide
8.74.1.22 isEnabled
8.74.1.23 isVisible
8.74.1.24 mapEvent
8.74.1.25 refresh
8.74.1.26 render
8.74.1.27 resize
8.74.1.28 setValue
8.74.1.29 show
8.74.1.30 sync
8.74.1.31 unbind

8.74.1.32 unblockEvent
8.74.1.33 validate
8.74.2 Properties
8.74.2.1 align
8.74.2.2 animate
8.74.2.3 attributes
8.74.2.4 borderless
8.74.2.5 bottomLabel
8.74.2.6 bottomPadding
8.74.2.7 click
8.74.2.8 container
8.74.2.9 content
8.74.2.10 css
8.74.2.11 disabled
8.74.2.12 gravity
8.74.2.13 height
8.74.2.14 hidden
8.74.2.15 hotkey
8.74.2.16 id
8.74.2.17 inputHeight
8.74.2.18 inputPadding
8.74.2.19 inputWidth
8.74.2.20 invalidMessage
8.74.2.21 label
8.74.2.22 labelPosition
8.74.2.23 labelWidth
8.74.2.24 maxHeight
8.74.2.25 maxWidth
8.74.2.26 minHeight
8.74.2.27 minWidth
8.74.2.28 name
8.74.2.29 on
8.74.2.30 placeholder
8.74.2.31 popup
8.74.2.32 readonly
8.74.2.33 relatedAction
8.74.2.34 relatedView
8.74.2.35 required
8.74.2.36 suggest
8.74.2.37 tabFocus
8.74.2.38 template
8.74.2.39 tooltip
8.74.2.40 type
8.74.2.41 validate
8.74.2.42 validateEvent
8.74.2.43 value
8.74.2.44 width
8.74.3 Events
8.74.3.1 onAfterRender
8.74.3.2 onAfterScroll
8.74.3.3 onBeforeRender
8.74.3.4 onBindRequest
8.74.3.5 onChange
8.74.3.6 onDestruct
8.74.3.7 onItemClick
8.74.3.8 onKeyPress
8.74.3.9 onLongTouch
8.74.3.10 onSwipeX

8.74.3.11 onSwipeY
8.74.3.12 onTimedKeyPress
8.74.3.13 onTouchEnd
8.74.3.14 onTouchMove
8.74.3.15 onTouchStart
8.74.3.16 onViewResize
8.74.4 Others
8.74.4.1 $getSize
8.74.4.2 $getValue
8.74.4.3 $height
8.74.4.4 $render
8.74.4.5 $renderIcon
8.74.4.6 $renderInput
8.74.4.7 $renderLabel
8.74.4.8 $scope
8.74.4.9 $setSize
8.74.4.10 $setValue
8.74.4.11 $skin
8.74.4.12 $view
8.74.4.13 $width
8.74.4.14 config
8.74.4.15 name
8.74.4.16 on_click
8.74.4.17 touchable
8.75 ui.textarea
8.75.1 Methods
8.75.1.1 adjust
8.75.1.2 attachEvent
8.75.1.3 bind
8.75.1.4 blockEvent
8.75.1.5 blur
8.75.1.6 callEvent
8.75.1.7 define
8.75.1.8 destructor
8.75.1.9 detachEvent
8.75.1.10 disable
8.75.1.11 enable
8.75.1.12 focus
8.75.1.13 getChildViews
8.75.1.14 getFormView
8.75.1.15 getInputNode
8.75.1.16 getNode
8.75.1.17 getParentView
8.75.1.18 getTopParentView
8.75.1.19 getValue
8.75.1.20 hasEvent
8.75.1.21 hide
8.75.1.22 isEnabled
8.75.1.23 isVisible
8.75.1.24 mapEvent
8.75.1.25 refresh
8.75.1.26 render
8.75.1.27 resize
8.75.1.28 setValue
8.75.1.29 show
8.75.1.30 sync
8.75.1.31 unbind
8.75.1.32 unblockEvent

8.75.1.33 validate
8.75.2 Properties
8.75.2.1 align
8.75.2.2 animate
8.75.2.3 attributes
8.75.2.4 borderless
8.75.2.5 bottomLabel
8.75.2.6 bottomPadding
8.75.2.7 click
8.75.2.8 container
8.75.2.9 content
8.75.2.10 css
8.75.2.11 disabled
8.75.2.12 gravity
8.75.2.13 height
8.75.2.14 hidden
8.75.2.15 hotkey
8.75.2.16 id
8.75.2.17 inputHeight
8.75.2.18 inputPadding
8.75.2.19 inputWidth
8.75.2.20 invalidMessage
8.75.2.21 label
8.75.2.22 labelPosition
8.75.2.23 labelWidth
8.75.2.24 maxHeight
8.75.2.25 maxWidth
8.75.2.26 minHeight
8.75.2.27 minWidth
8.75.2.28 name
8.75.2.29 on
8.75.2.30 placeholder
8.75.2.31 popup
8.75.2.32 readonly
8.75.2.33 relatedAction
8.75.2.34 relatedView
8.75.2.35 required
8.75.2.36 suggest
8.75.2.37 tabFocus
8.75.2.38 template
8.75.2.39 tooltip
8.75.2.40 type
8.75.2.41 validate
8.75.2.42 validateEvent
8.75.2.43 value
8.75.2.44 width
8.75.3 Events
8.75.3.1 onAfterRender
8.75.3.2 onAfterScroll
8.75.3.3 onBeforeRender
8.75.3.4 onBindRequest
8.75.3.5 onChange
8.75.3.6 onDestruct
8.75.3.7 onItemClick
8.75.3.8 onKeyPress
8.75.3.9 onLongTouch
8.75.3.10 onSwipeX
8.75.3.11 onSwipeY

8.75.3.12 onTimedKeyPress
8.75.3.13 onTouchEnd
8.75.3.14 onTouchMove
8.75.3.15 onTouchStart
8.75.3.16 onViewResize
8.75.4 Others
8.75.4.1 $getSize
8.75.4.2 $getValue
8.75.4.3 $height
8.75.4.4 $render
8.75.4.5 $renderIcon
8.75.4.6 $renderInput
8.75.4.7 $renderLabel
8.75.4.8 $scope
8.75.4.9 $setSize
8.75.4.10 $setValue
8.75.4.11 $skin
8.75.4.12 $view
8.75.4.13 $width
8.75.4.14 config
8.75.4.15 name
8.75.4.16 on_click
8.75.4.17 touchable
8.76 ui.toggle
8.76.1 Methods
8.76.1.1 adjust
8.76.1.2 attachEvent
8.76.1.3 bind
8.76.1.4 blockEvent
8.76.1.5 blur
8.76.1.6 callEvent
8.76.1.7 define
8.76.1.8 destructor
8.76.1.9 detachEvent
8.76.1.10 disable
8.76.1.11 enable
8.76.1.12 focus
8.76.1.13 getChildViews
8.76.1.14 getFormView
8.76.1.15 getInputNode
8.76.1.16 getNode
8.76.1.17 getParentView
8.76.1.18 getTopParentView
8.76.1.19 getValue
8.76.1.20 hasEvent
8.76.1.21 hide
8.76.1.22 isEnabled
8.76.1.23 isVisible
8.76.1.24 mapEvent
8.76.1.25 refresh
8.76.1.26 render
8.76.1.27 resize
8.76.1.28 setValue
8.76.1.29 show
8.76.1.30 sync
8.76.1.31 toggle
8.76.1.32 unbind
8.76.1.33 unblockEvent

8.76.2 Properties
8.76.2.1 align
8.76.2.2 animate
8.76.2.3 borderless
8.76.2.4 click
8.76.2.5 container
8.76.2.6 content
8.76.2.7 css
8.76.2.8 disabled
8.76.2.9 gravity
8.76.2.10 height
8.76.2.11 hidden
8.76.2.12 hotkey
8.76.2.13 id
8.76.2.14 inputHeight
8.76.2.15 inputWidth
8.76.2.16 label
8.76.2.17 labelPosition
8.76.2.18 maxHeight
8.76.2.19 maxWidth
8.76.2.20 minHeight
8.76.2.21 minWidth
8.76.2.22 name
8.76.2.23 on
8.76.2.24 placeholder
8.76.2.25 popup
8.76.2.26 tabFocus
8.76.2.27 template
8.76.2.28 tooltip
8.76.2.29 type
8.76.2.30 value
8.76.2.31 width
8.76.3 Events
8.76.3.1 onAfterRender
8.76.3.2 onAfterScroll
8.76.3.3 onBeforeRender
8.76.3.4 onBindRequest
8.76.3.5 onChange
8.76.3.6 onDestruct
8.76.3.7 onItemClick
8.76.3.8 onKeyPress
8.76.3.9 onLongTouch
8.76.3.10 onSwipeX
8.76.3.11 onSwipeY
8.76.3.12 onTimedKeyPress
8.76.3.13 onTouchEnd
8.76.3.14 onTouchMove
8.76.3.15 onTouchStart
8.76.3.16 onViewResize
8.76.4 Others
8.76.4.1 $getSize
8.76.4.2 $getValue
8.76.4.3 $height
8.76.4.4 $scope
8.76.4.5 $setSize
8.76.4.6 $setValue
8.76.4.7 $skin
8.76.4.8 $view

8.76.4.9 $width
8.76.4.10 config
8.76.4.11 name
8.76.4.12 on_click
8.76.4.13 touchable
8.77 ui.toolbar
8.77.1 Methods
8.77.1.1 addView
8.77.1.2 adjust
8.77.1.3 attachEvent
8.77.1.4 bind
8.77.1.5 blockEvent
8.77.1.6 callEvent
8.77.1.7 clear
8.77.1.8 clearValidation
8.77.1.9 define
8.77.1.10 destructor
8.77.1.11 detachEvent
8.77.1.12 disable
8.77.1.13 enable
8.77.1.14 focus
8.77.1.15 getChildViews
8.77.1.16 getCleanValues
8.77.1.17 getDirtyValues
8.77.1.18 getFormView
8.77.1.19 getNode
8.77.1.20 getParentView
8.77.1.21 getScrollState
8.77.1.22 getTopParentView
8.77.1.23 getValues
8.77.1.24 hasEvent
8.77.1.25 hide
8.77.1.26 index
8.77.1.27 isDirty
8.77.1.28 isEnabled
8.77.1.29 isVisible
8.77.1.30 load
8.77.1.31 mapEvent
8.77.1.32 parse
8.77.1.33 reconstruct
8.77.1.34 refresh
8.77.1.35 removeView
8.77.1.36 render
8.77.1.37 resize
8.77.1.38 resizeChildren
8.77.1.39 scrollTo
8.77.1.40 setDirty
8.77.1.41 setValues
8.77.1.42 show
8.77.1.43 showBatch
8.77.1.44 unbind
8.77.1.45 unblockEvent
8.77.1.46 validate
8.77.2 Properties
8.77.2.1 animate
8.77.2.2 borderless
8.77.2.3 cols
8.77.2.4 container

8.77.2.5 css
8.77.2.6 data
8.77.2.7 dataFeed
8.77.2.8 datatype
8.77.2.9 disabled
8.77.2.10 elements
8.77.2.11 elementsConfig
8.77.2.12 gravity
8.77.2.13 height
8.77.2.14 hidden
8.77.2.15 id
8.77.2.16 isolate
8.77.2.17 margin
8.77.2.18 maxHeight
8.77.2.19 maxWidth
8.77.2.20 minHeight
8.77.2.21 minWidth
8.77.2.22 on
8.77.2.23 padding
8.77.2.24 paddingX
8.77.2.25 paddingY
8.77.2.26 responsive
8.77.2.27 rows
8.77.2.28 rules
8.77.2.29 scroll
8.77.2.30 scrollSpeed
8.77.2.31 type
8.77.2.32 url
8.77.2.33 visibleBatch
8.77.2.34 width
8.77.3 Events
8.77.3.1 onAfterLoad
8.77.3.2 onAfterValidation
8.77.3.3 onBeforeLoad
8.77.3.4 onBeforeValidate
8.77.3.5 onBindRequest
8.77.3.6 onChange
8.77.3.7 onDestruct
8.77.3.8 onLoadError
8.77.3.9 onValidationError
8.77.3.10 onValidationSuccess
8.77.3.11 onValues
8.77.4 Others
8.77.4.1 $getSize
8.77.4.2 $height
8.77.4.3 $setSize
8.77.4.4 $skin
8.77.4.5 $view
8.77.4.6 $width
8.77.4.7 config
8.77.4.8 name
8.78 ui.tooltip
8.78.1 Methods
8.78.1.1 adjust
8.78.1.2 attachEvent
8.78.1.3 bind
8.78.1.4 blockEvent
8.78.1.5 callEvent

8.78.1.6 customize
8.78.1.7 define
8.78.1.8 destructor
8.78.1.9 detachEvent
8.78.1.10 disable
8.78.1.11 enable
8.78.1.12 getChildViews
8.78.1.13 getFormView
8.78.1.14 getNode
8.78.1.15 getParentView
8.78.1.16 getTopParentView
8.78.1.17 hasEvent
8.78.1.18 hide
8.78.1.19 isEnabled
8.78.1.20 isVisible
8.78.1.21 mapEvent
8.78.1.22 render
8.78.1.23 resize
8.78.1.24 show
8.78.1.25 sync
8.78.1.26 unbind
8.78.1.27 unblockEvent
8.78.2 Properties
8.78.2.1 animate
8.78.2.2 borderless
8.78.2.3 container
8.78.2.4 content
8.78.2.5 css
8.78.2.6 disabled
8.78.2.7 dx
8.78.2.8 dy
8.78.2.9 gravity
8.78.2.10 height
8.78.2.11 hidden
8.78.2.12 id
8.78.2.13 maxHeight
8.78.2.14 maxWidth
8.78.2.15 minHeight
8.78.2.16 minWidth
8.78.2.17 on
8.78.2.18 template
8.78.2.19 width
8.78.3 Events
8.78.3.1 onAfterRender
8.78.3.2 onAfterScroll
8.78.3.3 onBeforeRender
8.78.3.4 onBindRequest
8.78.3.5 onDestruct
8.78.3.6 onKeyPress
8.78.3.7 onLongTouch
8.78.3.8 onSwipeX
8.78.3.9 onSwipeY
8.78.3.10 onTimedKeyPress
8.78.3.11 onTouchEnd
8.78.3.12 onTouchMove
8.78.3.13 onTouchStart
8.78.3.14 onViewResize
8.78.4 Others

8.78.4.1 $getSize
8.78.4.2 $height
8.78.4.3 $scope
8.78.4.4 $setSize
8.78.4.5 $skin
8.78.4.6 $view
8.78.4.7 $width
8.78.4.8 config
8.78.4.9 name
8.78.4.10 type
8.79 ui.tree
8.79.1 Methods
8.79.1.1 add
8.79.1.2 addCss
8.79.1.3 adjust
8.79.1.4 attachEvent
8.79.1.5 bind
8.79.1.6 blockEvent
8.79.1.7 callEvent
8.79.1.8 checkAll
8.79.1.9 checkItem
8.79.1.10 clearAll
8.79.1.11 clearCss
8.79.1.12 clearValidation
8.79.1.13 close
8.79.1.14 closeAll
8.79.1.15 copy
8.79.1.16 count
8.79.1.17 customize
8.79.1.18 define
8.79.1.19 destructor
8.79.1.20 detachEvent
8.79.1.21 disable
8.79.1.22 enable
8.79.1.23 exists
8.79.1.24 filter
8.79.1.25 getBranchIndex
8.79.1.26 getChecked
8.79.1.27 getChildViews
8.79.1.28 getFirstChildId
8.79.1.29 getFirstId
8.79.1.30 getFormView
8.79.1.31 getIdByIndex
8.79.1.32 getIndexById
8.79.1.33 getItem
8.79.1.34 getItemNode
8.79.1.35 getLastId
8.79.1.36 getNextId
8.79.1.37 getNextSiblingId
8.79.1.38 getNode
8.79.1.39 getOpenItems
8.79.1.40 getParentId
8.79.1.41 getParentView
8.79.1.42 getPrevId
8.79.1.43 getPrevSiblingId
8.79.1.44 getScrollState
8.79.1.45 getSelectedId
8.79.1.46 getSelectedItem

8.79.1.47 getState
8.79.1.48 getTopParentView
8.79.1.49 group
8.79.1.50 hasCss
8.79.1.51 hasEvent
8.79.1.52 hide
8.79.1.53 isBranch
8.79.1.54 isBranchOpen
8.79.1.55 isChecked
8.79.1.56 isEnabled
8.79.1.57 isSelected
8.79.1.58 isVisible
8.79.1.59 load
8.79.1.60 loadBranch
8.79.1.61 loadNext
8.79.1.62 locate
8.79.1.63 mapEvent
8.79.1.64 move
8.79.1.65 moveSelection
8.79.1.66 open
8.79.1.67 openAll
8.79.1.68 parse
8.79.1.69 refresh
8.79.1.70 remove
8.79.1.71 removeCss
8.79.1.72 render
8.79.1.73 resize
8.79.1.74 scrollTo
8.79.1.75 select
8.79.1.76 selectAll
8.79.1.77 serialize
8.79.1.78 setState
8.79.1.79 show
8.79.1.80 showItem
8.79.1.81 sort
8.79.1.82 sync
8.79.1.83 unbind
8.79.1.84 unblockEvent
8.79.1.85 uncheckAll
8.79.1.86 uncheckItem
8.79.1.87 ungroup
8.79.1.88 unselect
8.79.1.89 unselectAll
8.79.1.90 updateItem
8.79.1.91 validate
8.79.2 Properties
8.79.2.1 animate
8.79.2.2 borderless
8.79.2.3 click
8.79.2.4 clipboard
8.79.2.5 container
8.79.2.6 css
8.79.2.7 data
8.79.2.8 dataFeed
8.79.2.9 datathrottle
8.79.2.10 datatype
8.79.2.11 disabled
8.79.2.12 drag

8.79.2.13 dragscroll
8.79.2.14 filterMode
8.79.2.15 gravity
8.79.2.16 height
8.79.2.17 hidden
8.79.2.18 id
8.79.2.19 maxHeight
8.79.2.20 maxWidth
8.79.2.21 minHeight
8.79.2.22 minWidth
8.79.2.23 mouseEventDelay
8.79.2.24 multiselect
8.79.2.25 navigation
8.79.2.26 on
8.79.2.27 onClick
8.79.2.28 onContext
8.79.2.29 onDblClick
8.79.2.30 onMouseMove
8.79.2.31 pager
8.79.2.32 ready
8.79.2.33 removeMissed
8.79.2.34 rules
8.79.2.35 save
8.79.2.36 scheme
8.79.2.37 scroll
8.79.2.38 scrollSpeed
8.79.2.39 select
8.79.2.40 template
8.79.2.41 templateCopy
8.79.2.42 threeState
8.79.2.43 tooltip
8.79.2.44 type
8.79.2.45 url
8.79.2.46 width
8.79.3 Events
8.79.3.1 onAfterAdd
8.79.3.2 onAfterClose
8.79.3.3 onAfterContextMenu
8.79.3.4 onAfterDelete
8.79.3.5 onAfterDrop
8.79.3.6 onAfterLoad
8.79.3.7 onAfterOpen
8.79.3.8 onAfterRender
8.79.3.9 onAfterScroll
8.79.3.10 onAfterSelect
8.79.3.11 onAfterSort
8.79.3.12 onBeforeAdd
8.79.3.13 onBeforeClose
8.79.3.14 onBeforeContextMenu
8.79.3.15 onBeforeDelete
8.79.3.16 onBeforeDrag
8.79.3.17 onBeforeDragIn
8.79.3.18 onBeforeDrop
8.79.3.19 onBeforeDropOut
8.79.3.20 onBeforeLoad
8.79.3.21 onBeforeOpen
8.79.3.22 onBeforeRender
8.79.3.23 onBeforeSelect

8.79.3.24 onBeforeSort
8.79.3.25 onBindRequest
8.79.3.26 onDataRequest
8.79.3.27 onDataUpdate
8.79.3.28 onDestruct
8.79.3.29 onDragOut
8.79.3.30 onItemCheck
8.79.3.31 onItemClick
8.79.3.32 onItemDblClick
8.79.3.33 onItemRender
8.79.3.34 onKeyPress
8.79.3.35 onLoadError
8.79.3.36 onLongTouch
8.79.3.37 onMouseMove
8.79.3.38 onMouseMoving
8.79.3.39 onMouseOut
8.79.3.40 onPartialRender
8.79.3.41 onPaste
8.79.3.42 onSelectChange
8.79.3.43 onSwipeX
8.79.3.44 onSwipeY
8.79.3.45 onTimedKeyPress
8.79.3.46 onTouchEnd
8.79.3.47 onTouchMove
8.79.3.48 onTouchStart
8.79.3.49 onValidationError
8.79.3.50 onValidationSuccess
8.79.3.51 onViewResize
8.79.4 Others
8.79.4.1 $drag
8.79.4.2 $dragHTML
8.79.4.3 $dragIn
8.79.4.4 $dragMark
8.79.4.5 $dragOut
8.79.4.6 $drop
8.79.4.7 $dropAllow
8.79.4.8 $getSize
8.79.4.9 $height
8.79.4.10 $scope
8.79.4.11 $setSize
8.79.4.12 $skin
8.79.4.13 $view
8.79.4.14 $width
8.79.4.15 config
8.79.4.16 name
8.79.4.17 on_click
8.79.4.18 on_context
8.79.4.19 on_dblclick
8.79.4.20 on_mouse_move
8.79.4.21 type
8.79.4.22 types
8.80 ui.treetable
8.80.1 Methods
8.80.1.1 add
8.80.1.2 addCellCss
8.80.1.3 addCss
8.80.1.4 addRowCss
8.80.1.5 adjust

8.80.1.6 adjustColumn
8.80.1.7 adjustRowHeight
8.80.1.8 attachEvent
8.80.1.9 bind
8.80.1.10 blockEvent
8.80.1.11 callEvent
8.80.1.12 checkAll
8.80.1.13 checkItem
8.80.1.14 clearAll
8.80.1.15 clearCss
8.80.1.16 clearValidation
8.80.1.17 close
8.80.1.18 closeAll
8.80.1.19 collectValues
8.80.1.20 columnId
8.80.1.21 copy
8.80.1.22 count
8.80.1.23 define
8.80.1.24 destructor
8.80.1.25 detachEvent
8.80.1.26 disable
8.80.1.27 eachColumn
8.80.1.28 eachRow
8.80.1.29 edit
8.80.1.30 editCancel
8.80.1.31 editCell
8.80.1.32 editColumn
8.80.1.33 editNext
8.80.1.34 editRow
8.80.1.35 editStop
8.80.1.36 enable
8.80.1.37 exists
8.80.1.38 exportToExcel
8.80.1.39 exportToPDF
8.80.1.40 filter
8.80.1.41 filterByAll
8.80.1.42 find
8.80.1.43 focusEditor
8.80.1.44 getBranchIndex
8.80.1.45 getChecked
8.80.1.46 getChildViews
8.80.1.47 getColumnConfig
8.80.1.48 getColumnIndex
8.80.1.49 getEditState
8.80.1.50 getEditor
8.80.1.51 getEditorValue
8.80.1.52 getFilter
8.80.1.53 getFirstChildId
8.80.1.54 getFirstId
8.80.1.55 getFormView
8.80.1.56 getHeaderContent
8.80.1.57 getHeaderNode
8.80.1.58 getIdByIndex
8.80.1.59 getIndexById
8.80.1.60 getItem
8.80.1.61 getItemNode
8.80.1.62 getLastId
8.80.1.63 getNextId

8.80.1.64 getNextSiblingId
8.80.1.65 getNode
8.80.1.66 getOpenItems
8.80.1.67 getPage
8.80.1.68 getPager
8.80.1.69 getParentId
8.80.1.70 getParentView
8.80.1.71 getPrevId
8.80.1.72 getPrevSiblingId
8.80.1.73 getScrollState
8.80.1.74 getSelectedId
8.80.1.75 getSelectedItem
8.80.1.76 getState
8.80.1.77 getText
8.80.1.78 getTopParentView
8.80.1.79 getVisibleCount
8.80.1.80 group
8.80.1.81 hasCss
8.80.1.82 hasEvent
8.80.1.83 hide
8.80.1.84 hideColumn
8.80.1.85 hideOverlay
8.80.1.86 isBranch
8.80.1.87 isBranchOpen
8.80.1.88 isChecked
8.80.1.89 isColumnVisible
8.80.1.90 isEnabled
8.80.1.91 isVisible
8.80.1.92 load
8.80.1.93 loadBranch
8.80.1.94 loadNext
8.80.1.95 locate
8.80.1.96 mapCells
8.80.1.97 mapEvent
8.80.1.98 markSorting
8.80.1.99 move
8.80.1.100 moveBottom
8.80.1.101 moveColumn
8.80.1.102 moveDown
8.80.1.103 moveSelection
8.80.1.104 moveTop
8.80.1.105 moveUp
8.80.1.106 open
8.80.1.107 openAll
8.80.1.108 parse
8.80.1.109 refresh
8.80.1.110 refreshColumns
8.80.1.111 refreshFilter
8.80.1.112 refreshHeaderContent
8.80.1.113 registerFilter
8.80.1.114 remove
8.80.1.115 removeCellCss
8.80.1.116 removeCss
8.80.1.117 removeRowCss
8.80.1.118 render
8.80.1.119 resize
8.80.1.120 scrollTo
8.80.1.121 serialize

8.80.1.122 setColumnWidth
8.80.1.123 setPage
8.80.1.124 setRowHeight
8.80.1.125 setState
8.80.1.126 show
8.80.1.127 showCell
8.80.1.128 showColumn
8.80.1.129 showColumnBatch
8.80.1.130 showItem
8.80.1.131 showItemByIndex
8.80.1.132 showOverlay
8.80.1.133 sort
8.80.1.134 sync
8.80.1.135 unbind
8.80.1.136 unblockEvent
8.80.1.137 uncheckAll
8.80.1.138 uncheckItem
8.80.1.139 ungroup
8.80.1.140 updateItem
8.80.1.141 validate
8.80.1.142 validateEditor
8.80.2 Properties
8.80.2.1 animate
8.80.2.2 autoConfig
8.80.2.3 autoheight
8.80.2.4 autowidth
8.80.2.5 blockselect
8.80.2.6 borderless
8.80.2.7 checkboxRefresh
8.80.2.8 clipboard
8.80.2.9 columnWidth
8.80.2.10 columns
8.80.2.11 container
8.80.2.12 css
8.80.2.13 data
8.80.2.14 dataFeed
8.80.2.15 datafetch
8.80.2.16 datathrottle
8.80.2.17 datatype
8.80.2.18 delimiter
8.80.2.19 disabled
8.80.2.20 drag
8.80.2.21 dragColumn
8.80.2.22 dragscroll
8.80.2.23 editMath
8.80.2.24 editValue
8.80.2.25 editable
8.80.2.26 editaction
8.80.2.27 externalData
8.80.2.28 filterMode
8.80.2.29 fixedRowHeight
8.80.2.30 footer
8.80.2.31 form
8.80.2.32 gravity
8.80.2.33 header
8.80.2.34 headerRowHeight
8.80.2.35 headermenu
8.80.2.36 height

8.80.2.37 hidden
8.80.2.38 hover
8.80.2.39 id
8.80.2.40 leftSplit
8.80.2.41 liveValidation
8.80.2.42 loadahead
8.80.2.43 math
8.80.2.44 maxHeight
8.80.2.45 maxWidth
8.80.2.46 minColumnHeight
8.80.2.47 minColumnWidth
8.80.2.48 minHeight
8.80.2.49 minWidth
8.80.2.50 mouseEventDelay
8.80.2.51 multiselect
8.80.2.52 navigation
8.80.2.53 on
8.80.2.54 onClick
8.80.2.55 onContext
8.80.2.56 onDblClick
8.80.2.57 onMouseMove
8.80.2.58 pager
8.80.2.59 prerender
8.80.2.60 ready
8.80.2.61 removeMissed
8.80.2.62 resizeColumn
8.80.2.63 resizeRow
8.80.2.64 rightSplit
8.80.2.65 rowHeight
8.80.2.66 rowLineHeight
8.80.2.67 rules
8.80.2.68 save
8.80.2.69 scheme
8.80.2.70 scroll
8.80.2.71 scrollAlignY
8.80.2.72 scrollX
8.80.2.73 scrollY
8.80.2.74 select
8.80.2.75 spans
8.80.2.76 threeState
8.80.2.77 tooltip
8.80.2.78 type
8.80.2.79 url
8.80.2.80 width
8.80.2.81 yCount
8.80.3 Events
8.80.3.1 onAfterAdd
8.80.3.2 onAfterClose
8.80.3.3 onAfterColumnDrop
8.80.3.4 onAfterColumnDropOrder
8.80.3.5 onAfterContextMenu
8.80.3.6 onAfterDelete
8.80.3.7 onAfterDrop
8.80.3.8 onAfterEditStart
8.80.3.9 onAfterEditStop
8.80.3.10 onAfterFilter
8.80.3.11 onAfterLoad
8.80.3.12 onAfterOpen

8.80.3.13 onAfterRender
8.80.3.14 onAfterScroll
8.80.3.15 onAfterSelect
8.80.3.16 onAfterSort
8.80.3.17 onAfterUnselect
8.80.3.18 onBeforeAdd
8.80.3.19 onBeforeClose
8.80.3.20 onBeforeColumnDrag
8.80.3.21 onBeforeColumnDrop
8.80.3.22 onBeforeColumnDropOrder
8.80.3.23 onBeforeContextMenu
8.80.3.24 onBeforeDelete
8.80.3.25 onBeforeDrag
8.80.3.26 onBeforeDragIn
8.80.3.27 onBeforeDrop
8.80.3.28 onBeforeDropOut
8.80.3.29 onBeforeEditStart
8.80.3.30 onBeforeEditStop
8.80.3.31 onBeforeFilter
8.80.3.32 onBeforeLoad
8.80.3.33 onBeforeOpen
8.80.3.34 onBeforeRender
8.80.3.35 onBeforeSelect
8.80.3.36 onBeforeSort
8.80.3.37 onBeforeUnSelect
8.80.3.38 onBindRequest
8.80.3.39 onCheck
8.80.3.40 onColumnResize
8.80.3.41 onDataRequest
8.80.3.42 onDataUpdate
8.80.3.43 onDestruct
8.80.3.44 onDragOut
8.80.3.45 onHeaderClick
8.80.3.46 onItemCheck
8.80.3.47 onItemClick
8.80.3.48 onItemDblClick
8.80.3.49 onKeyPress
8.80.3.50 onLiveEdit
8.80.3.51 onLoadError
8.80.3.52 onLongTouch
8.80.3.53 onMouseMove
8.80.3.54 onMouseMoving
8.80.3.55 onMouseOut
8.80.3.56 onResize
8.80.3.57 onRowResize
8.80.3.58 onScrollX
8.80.3.59 onScrollY
8.80.3.60 onSelectChange
8.80.3.61 onStructureLoad
8.80.3.62 onStructureUpdate
8.80.3.63 onSwipeX
8.80.3.64 onSwipeY
8.80.3.65 onTimedKeyPress
8.80.3.66 onTouchEnd
8.80.3.67 onTouchMove
8.80.3.68 onTouchStart
8.80.3.69 onValidationError
8.80.3.70 onValidationSuccess

8.80.3.71 onViewResize
8.80.4 Others
8.80.4.1 $drag
8.80.4.2 $dragHTML
8.80.4.3 $dragIn
8.80.4.4 $dragMark
8.80.4.5 $dragOut
8.80.4.6 $drop
8.80.4.7 $dropAllow
8.80.4.8 $getSize
8.80.4.9 $height
8.80.4.10 $scope
8.80.4.11 $setSize
8.80.4.12 $skin
8.80.4.13 $view
8.80.4.14 $width
8.80.4.15 config
8.80.4.16 headerContent
8.80.4.17 name
8.80.4.18 on_click
8.80.4.19 on_context
8.80.4.20 on_dblclick
8.80.4.21 on_mouse_move
8.80.4.22 waitData
8.81 ui.unitlist
8.81.1 Methods
8.81.1.1 add
8.81.1.2 addCss
8.81.1.3 adjust
8.81.1.4 attachEvent
8.81.1.5 bind
8.81.1.6 blockEvent
8.81.1.7 callEvent
8.81.1.8 clearAll
8.81.1.9 clearCss
8.81.1.10 clearValidation
8.81.1.11 copy
8.81.1.12 count
8.81.1.13 customize
8.81.1.14 define
8.81.1.15 destructor
8.81.1.16 detachEvent
8.81.1.17 disable
8.81.1.18 enable
8.81.1.19 exists
8.81.1.20 filter
8.81.1.21 getChildViews
8.81.1.22 getFirstId
8.81.1.23 getFormView
8.81.1.24 getIdByIndex
8.81.1.25 getIndexById
8.81.1.26 getItem
8.81.1.27 getItemNode
8.81.1.28 getLastId
8.81.1.29 getNextId
8.81.1.30 getNode
8.81.1.31 getPage
8.81.1.32 getPager

8.81.1.33 getParentView
8.81.1.34 getPrevId
8.81.1.35 getScrollState
8.81.1.36 getSelectedId
8.81.1.37 getSelectedItem
8.81.1.38 getTopParentView
8.81.1.39 getUnitList
8.81.1.40 getUnits
8.81.1.41 getVisibleCount
8.81.1.42 hasCss
8.81.1.43 hasEvent
8.81.1.44 hide
8.81.1.45 isEnabled
8.81.1.46 isSelected
8.81.1.47 isVisible
8.81.1.48 load
8.81.1.49 loadNext
8.81.1.50 locate
8.81.1.51 mapEvent
8.81.1.52 move
8.81.1.53 moveBottom
8.81.1.54 moveDown
8.81.1.55 moveSelection
8.81.1.56 moveTop
8.81.1.57 moveUp
8.81.1.58 parse
8.81.1.59 refresh
8.81.1.60 remove
8.81.1.61 removeCss
8.81.1.62 render
8.81.1.63 resize
8.81.1.64 scrollTo
8.81.1.65 select
8.81.1.66 selectAll
8.81.1.67 serialize
8.81.1.68 setPage
8.81.1.69 show
8.81.1.70 showItem
8.81.1.71 sort
8.81.1.72 sync
8.81.1.73 unbind
8.81.1.74 unblockEvent
8.81.1.75 unselect
8.81.1.76 unselectAll
8.81.1.77 updateItem
8.81.1.78 validate
8.81.2 Properties
8.81.2.1 animate
8.81.2.2 autoheight
8.81.2.3 autowidth
8.81.2.4 borderless
8.81.2.5 click
8.81.2.6 clipboard
8.81.2.7 container
8.81.2.8 css
8.81.2.9 data
8.81.2.10 dataFeed
8.81.2.11 datathrottle

8.81.2.12 datatype
8.81.2.13 disabled
8.81.2.14 drag
8.81.2.15 dragscroll
8.81.2.16 externalData
8.81.2.17 gravity
8.81.2.18 height
8.81.2.19 hidden
8.81.2.20 id
8.81.2.21 layout
8.81.2.22 maxHeight
8.81.2.23 maxWidth
8.81.2.24 minHeight
8.81.2.25 minWidth
8.81.2.26 mouseEventDelay
8.81.2.27 multiselect
8.81.2.28 navigation
8.81.2.29 on
8.81.2.30 onClick
8.81.2.31 onContext
8.81.2.32 onDblClick
8.81.2.33 onMouseMove
8.81.2.34 pager
8.81.2.35 ready
8.81.2.36 removeMissed
8.81.2.37 rules
8.81.2.38 save
8.81.2.39 scheme
8.81.2.40 scroll
8.81.2.41 scrollSpeed
8.81.2.42 select
8.81.2.43 sort
8.81.2.44 template
8.81.2.45 templateCopy
8.81.2.46 tooltip
8.81.2.47 type
8.81.2.48 uniteBy
8.81.2.49 url
8.81.2.50 width
8.81.2.51 xCount
8.81.2.52 yCount
8.81.3 Events
8.81.3.1 onAfterAdd
8.81.3.2 onAfterContextMenu
8.81.3.3 onAfterDelete
8.81.3.4 onAfterDrop
8.81.3.5 onAfterLoad
8.81.3.6 onAfterRender
8.81.3.7 onAfterScroll
8.81.3.8 onAfterSelect
8.81.3.9 onAfterSort
8.81.3.10 onBeforeAdd
8.81.3.11 onBeforeContextMenu
8.81.3.12 onBeforeDelete
8.81.3.13 onBeforeDrag
8.81.3.14 onBeforeDragIn
8.81.3.15 onBeforeDrop
8.81.3.16 onBeforeDropOut

8.81.3.17 onBeforeLoad
8.81.3.18 onBeforeRender
8.81.3.19 onBeforeSelect
8.81.3.20 onBeforeSort
8.81.3.21 onBindRequest
8.81.3.22 onDataRequest
8.81.3.23 onDataUpdate
8.81.3.24 onDestruct
8.81.3.25 onDragOut
8.81.3.26 onItemClick
8.81.3.27 onItemDblClick
8.81.3.28 onItemRender
8.81.3.29 onKeyPress
8.81.3.30 onLoadError
8.81.3.31 onLongTouch
8.81.3.32 onMouseMove
8.81.3.33 onMouseMoving
8.81.3.34 onMouseOut
8.81.3.35 onPaste
8.81.3.36 onSelectChange
8.81.3.37 onSwipeX
8.81.3.38 onSwipeY
8.81.3.39 onTimedKeyPress
8.81.3.40 onTouchEnd
8.81.3.41 onTouchMove
8.81.3.42 onTouchStart
8.81.3.43 onValidationError
8.81.3.44 onValidationSuccess
8.81.3.45 onViewResize
8.81.4 Others
8.81.4.1 $drag
8.81.4.2 $dragHTML
8.81.4.3 $dragIn
8.81.4.4 $dragMark
8.81.4.5 $dragOut
8.81.4.6 $drop
8.81.4.7 $dropAllow
8.81.4.8 $getSize
8.81.4.9 $height
8.81.4.10 $scope
8.81.4.11 $setSize
8.81.4.12 $skin
8.81.4.13 $view
8.81.4.14 $width
8.81.4.15 config
8.81.4.16 name
8.81.4.17 on_click
8.81.4.18 on_context
8.81.4.19 on_dblclick
8.81.4.20 on_mouse_move
8.81.4.21 type
8.81.4.22 types
8.82 ui.uploader
8.82.1 Methods
8.82.1.1 addDropZone
8.82.1.2 addFile
8.82.1.3 adjust
8.82.1.4 attachEvent

8.82.1.5 bind
8.82.1.6 blockEvent
8.82.1.7 blur
8.82.1.8 callEvent
8.82.1.9 define
8.82.1.10 destructor
8.82.1.11 detachEvent
8.82.1.12 disable
8.82.1.13 enable
8.82.1.14 fileDialog
8.82.1.15 focus
8.82.1.16 getChildViews
8.82.1.17 getFormView
8.82.1.18 getInputNode
8.82.1.19 getNode
8.82.1.20 getParentView
8.82.1.21 getTopParentView
8.82.1.22 getValue
8.82.1.23 hasEvent
8.82.1.24 hide
8.82.1.25 isEnabled
8.82.1.26 isUploaded
8.82.1.27 isVisible
8.82.1.28 mapEvent
8.82.1.29 refresh
8.82.1.30 render
8.82.1.31 resize
8.82.1.32 send
8.82.1.33 setValue
8.82.1.34 show
8.82.1.35 stopUpload
8.82.1.36 sync
8.82.1.37 unbind
8.82.1.38 unblockEvent
8.82.2 Properties
8.82.2.1 align
8.82.2.2 animate
8.82.2.3 apiOnly
8.82.2.4 autosend
8.82.2.5 borderless
8.82.2.6 click
8.82.2.7 container
8.82.2.8 content
8.82.2.9 css
8.82.2.10 disabled
8.82.2.11 formData
8.82.2.12 getValue
8.82.2.13 gravity
8.82.2.14 height
8.82.2.15 hidden
8.82.2.16 hotkey
8.82.2.17 id
8.82.2.18 inputHeight
8.82.2.19 inputWidth
8.82.2.20 label
8.82.2.21 labelPosition
8.82.2.22 link
8.82.2.23 maxHeight

8.82.2.24 maxWidth
8.82.2.25 minHeight
8.82.2.26 minWidth
8.82.2.27 multiple
8.82.2.28 name
8.82.2.29 on
8.82.2.30 placeholder
8.82.2.31 popup
8.82.2.32 tabFocus
8.82.2.33 template
8.82.2.34 tooltip
8.82.2.35 type
8.82.2.36 value
8.82.2.37 width
8.82.3 Events
8.82.3.1 onAfterFileAdd
8.82.3.2 onAfterRender
8.82.3.3 onAfterScroll
8.82.3.4 onBeforeFileAdd
8.82.3.5 onBeforeRender
8.82.3.6 onBindRequest
8.82.3.7 onChange
8.82.3.8 onDestruct
8.82.3.9 onFileUpload
8.82.3.10 onFileUploadError
8.82.3.11 onItemClick
8.82.3.12 onKeyPress
8.82.3.13 onLongTouch
8.82.3.14 onSwipeX
8.82.3.15 onSwipeY
8.82.3.16 onTimedKeyPress
8.82.3.17 onTouchEnd
8.82.3.18 onTouchMove
8.82.3.19 onTouchStart
8.82.3.20 onUploadComplete
8.82.3.21 onViewResize
8.82.4 Others
8.82.4.1 $cssName
8.82.4.2 $getSize
8.82.4.3 $getValue
8.82.4.4 $height
8.82.4.5 $scope
8.82.4.6 $setSize
8.82.4.7 $setValue
8.82.4.8 $skin
8.82.4.9 $view
8.82.4.10 $width
8.82.4.11 config
8.82.4.12 name
8.82.4.13 on_click
8.82.4.14 touchable
8.83 ui.video
8.83.1 Methods
8.83.1.1 adjust
8.83.1.2 bind
8.83.1.3 define
8.83.1.4 destructor
8.83.1.5 disable

8.83.1.6 enable
8.83.1.7 getChildViews
8.83.1.8 getFormView
8.83.1.9 getNode
8.83.1.10 getParentView
8.83.1.11 getTopParentView
8.83.1.12 getVideo
8.83.1.13 hide
8.83.1.14 isEnabled
8.83.1.15 isVisible
8.83.1.16 resize
8.83.1.17 show
8.83.1.18 unbind
8.83.2 Properties
8.83.2.1 animate
8.83.2.2 borderless
8.83.2.3 container
8.83.2.4 controls
8.83.2.5 css
8.83.2.6 disabled
8.83.2.7 gravity
8.83.2.8 height
8.83.2.9 hidden
8.83.2.10 id
8.83.2.11 maxHeight
8.83.2.12 maxWidth
8.83.2.13 minHeight
8.83.2.14 minWidth
8.83.2.15 src
8.83.2.16 width
8.83.3 Events
8.83.3.1 onAfterScroll
8.83.3.2 onBindRequest
8.83.3.3 onDestruct
8.83.3.4 onKeyPress
8.83.3.5 onLongTouch
8.83.3.6 onSwipeX
8.83.3.7 onSwipeY
8.83.3.8 onTimedKeyPress
8.83.3.9 onTouchEnd
8.83.3.10 onTouchMove
8.83.3.11 onTouchStart
8.83.3.12 onViewResize
8.83.4 Others
8.83.4.1 $getSize
8.83.4.2 $height
8.83.4.3 $scope
8.83.4.4 $setSize
8.83.4.5 $skin
8.83.4.6 $view
8.83.4.7 $width
8.83.4.8 config
8.83.4.9 name
8.84 ui.window
8.84.1 Methods
8.84.1.1 adjust
8.84.1.2 attachEvent
8.84.1.3 bind

8.84.1.4 blockEvent
8.84.1.5 callEvent
8.84.1.6 close
8.84.1.7 define
8.84.1.8 destructor
8.84.1.9 detachEvent
8.84.1.10 disable
8.84.1.11 enable
8.84.1.12 getBody
8.84.1.13 getChildViews
8.84.1.14 getFormView
8.84.1.15 getHead
8.84.1.16 getNode
8.84.1.17 getParentView
8.84.1.18 getTopParentView
8.84.1.19 hasEvent
8.84.1.20 hide
8.84.1.21 isEnabled
8.84.1.22 isVisible
8.84.1.23 mapEvent
8.84.1.24 resize
8.84.1.25 resizeChildren
8.84.1.26 setPosition
8.84.1.27 show
8.84.1.28 unbind
8.84.1.29 unblockEvent
8.84.2 Properties
8.84.2.1 animate
8.84.2.2 autofit
8.84.2.3 autofocus
8.84.2.4 body
8.84.2.5 borderless
8.84.2.6 container
8.84.2.7 css
8.84.2.8 disabled
8.84.2.9 fullscreen
8.84.2.10 gravity
8.84.2.11 head
8.84.2.12 headHeight
8.84.2.13 height
8.84.2.14 hidden
8.84.2.15 id
8.84.2.16 left
8.84.2.17 maxHeight
8.84.2.18 maxWidth
8.84.2.19 minHeight
8.84.2.20 minWidth
8.84.2.21 modal
8.84.2.22 move
8.84.2.23 on
8.84.2.24 padding
8.84.2.25 position
8.84.2.26 relative
8.84.2.27 top
8.84.2.28 width
8.84.2.29 zIndex
8.84.3 Events
8.84.3.1 onAfterScroll

8.84.3.2 onBeforeShow
8.84.3.3 onBindRequest
8.84.3.4 onDestruct
8.84.3.5 onHide
8.84.3.6 onKeyPress
8.84.3.7 onLongTouch
8.84.3.8 onShow
8.84.3.9 onSwipeX
8.84.3.10 onSwipeY
8.84.3.11 onTimedKeyPress
8.84.3.12 onTouchEnd
8.84.3.13 onTouchMove
8.84.3.14 onTouchStart
8.84.3.15 onViewMove
8.84.3.16 onViewMoveEnd
8.84.3.17 onViewResize
8.84.4 Others
8.84.4.1 $getSize
8.84.4.2 $height
8.84.4.3 $scope
8.84.4.4 $setSize
8.84.4.5 $skin
8.84.4.6 $view
8.84.4.7 $width
8.84.4.8 config
8.84.4.9 name
8.85 Mixins
8.85.1 ActiveContent
8.85.1.1 Properties
8.85.1.1.1 activeContent
8.85.2 AtomDataLoader
8.85.2.1 Methods
8.85.2.1.1 load
8.85.2.1.2 parse
8.85.2.2 Properties
8.85.2.2.1 data
8.85.2.2.2 dataFeed
8.85.2.2.3 datatype
8.85.2.2.4 url
8.85.2.3 Events
8.85.2.3.1 onAfterLoad
8.85.2.3.2 onBeforeLoad
8.85.2.3.3 onLoadError
8.85.3 AtomRender
8.85.3.1 Methods
8.85.3.1.1 render
8.85.3.1.2 sync
8.85.3.2 Properties
8.85.3.2.1 content
8.85.3.2.2 template
8.85.3.3 Events
8.85.3.3.1 onAfterRender
8.85.3.3.2 onBeforeRender
8.85.4 AutoTooltip
8.85.4.1 Properties
8.85.4.1.1 tooltip
8.85.5 BaseBind
8.85.5.1 Methods

8.85.5.1.1 bind
8.85.5.1.2 unbind
8.85.5.2 Events
8.85.5.2.1 onBindRequest
8.85.6 BindSource
8.85.6.1 Methods
8.85.6.1.1 addBind
8.85.6.1.2 getBindData
8.85.6.1.3 removeBind
8.85.6.1.4 saveBatch
8.85.6.1.5 setBindData
8.85.6.2 Events
8.85.6.2.1 onBindUpdate
8.85.7 Canvas
8.85.7.1 Methods
8.85.7.1.1 clearCanvas
8.85.7.1.2 getCanvas
8.85.7.1.3 hideCanvas
8.85.7.1.4 renderText
8.85.7.1.5 renderTextAt
8.85.7.1.6 showCanvas
8.85.7.1.7 toggleCanvas
8.85.8 CollectionBind
8.85.8.1 Methods
8.85.8.1.1 getCursor
8.85.8.1.2 refreshCursor
8.85.8.1.3 setCursor
8.85.8.2 Properties
8.85.8.2.1 defaultData
8.85.8.3 Events
8.85.8.3.1 onAfterCursorChange
8.85.8.3.2 onBeforeCursorChange
8.85.9 ContextHelper
8.85.9.1 Methods
8.85.9.1.1 attachTo
8.85.9.1.2 getContext
8.85.9.2 Properties
8.85.9.2.1 body
8.85.9.2.2 hidden
8.85.9.2.3 master
8.85.9.2.4 padding
8.85.9.3 Events
8.85.9.3.1 onBeforeShow
8.85.10 CopyPaste
8.85.10.1 Properties
8.85.10.1.1 clipboard
8.85.10.1.2 templateCopy
8.85.10.2 Events
8.85.10.2.1 onPaste
8.85.11 CustomScroll
8.85.11.1 Methods
8.85.11.1.1 enable
8.85.11.1.2 init
8.85.11.2 Others
8.85.11.2.1 scrollStep
8.85.12 DataCollection
8.85.12.1 Methods
8.85.12.1.1 add

8.85.12.1.2 addBind
8.85.12.1.3 attachEvent
8.85.12.1.4 bind
8.85.12.1.5 blockEvent
8.85.12.1.6 callEvent
8.85.12.1.7 clearAll
8.85.12.1.8 clearValidation
8.85.12.1.9 count
8.85.12.1.10 define
8.85.12.1.11 destructor
8.85.12.1.12 detachEvent
8.85.12.1.13 exists
8.85.12.1.14 filter
8.85.12.1.15 getBindData
8.85.12.1.16 getCursor
8.85.12.1.17 getFirstId
8.85.12.1.18 getIdByIndex
8.85.12.1.19 getIndexById
8.85.12.1.20 getItem
8.85.12.1.21 getLastId
8.85.12.1.22 getNextId
8.85.12.1.23 getPrevId
8.85.12.1.24 hasEvent
8.85.12.1.25 isVisible
8.85.12.1.26 load
8.85.12.1.27 loadNext
8.85.12.1.28 mapEvent
8.85.12.1.29 parse
8.85.12.1.30 refresh
8.85.12.1.31 refreshCursor
8.85.12.1.32 remove
8.85.12.1.33 removeBind
8.85.12.1.34 saveBatch
8.85.12.1.35 serialize
8.85.12.1.36 setBindData
8.85.12.1.37 setCursor
8.85.12.1.38 sort
8.85.12.1.39 sync
8.85.12.1.40 unbind
8.85.12.1.41 unblockEvent
8.85.12.1.42 updateItem
8.85.12.1.43 validate
8.85.12.2 Properties
8.85.12.2.1 data
8.85.12.2.2 dataFeed
8.85.12.2.3 datathrottle
8.85.12.2.4 datatype
8.85.12.2.5 defaultData
8.85.12.2.6 id
8.85.12.2.7 on
8.85.12.2.8 ready
8.85.12.2.9 removeMissed
8.85.12.2.10 rules
8.85.12.2.11 save
8.85.12.2.12 scheme
8.85.12.2.13 url
8.85.12.3 Events
8.85.12.3.1 onAfterAdd

8.85.12.3.2 onAfterCursorChange
8.85.12.3.3 onAfterDelete
8.85.12.3.4 onAfterLoad
8.85.12.3.5 onAfterSort
8.85.12.3.6 onBeforeAdd
8.85.12.3.7 onBeforeCursorChange
8.85.12.3.8 onBeforeDelete
8.85.12.3.9 onBeforeLoad
8.85.12.3.10 onBeforeSort
8.85.12.3.11 onBindRequest
8.85.12.3.12 onBindUpdate
8.85.12.3.13 onDataRequest
8.85.12.3.14 onDataUpdate
8.85.12.3.15 onDestruct
8.85.12.3.16 onLoadError
8.85.12.3.17 onValidationError
8.85.12.3.18 onValidationSuccess
8.85.12.4 Others
8.85.12.4.1 config
8.85.12.4.2 name
8.85.13 DataDriver
8.85.13.1 Others
8.85.13.1.1 csv
8.85.13.1.2 html
8.85.13.1.3 htmltable
8.85.13.1.4 jsarray
8.85.13.1.5 json
8.85.13.1.6 xml
8.85.14 DataLoader
8.85.14.1 Methods
8.85.14.1.1 add
8.85.14.1.2 clearAll
8.85.14.1.3 count
8.85.14.1.4 exists
8.85.14.1.5 filter
8.85.14.1.6 getFirstId
8.85.14.1.7 getIdByIndex
8.85.14.1.8 getIndexById
8.85.14.1.9 getItem
8.85.14.1.10 getLastId
8.85.14.1.11 getNextId
8.85.14.1.12 getPrevId
8.85.14.1.13 load
8.85.14.1.14 loadNext
8.85.14.1.15 parse
8.85.14.1.16 refresh
8.85.14.1.17 remove
8.85.14.1.18 serialize
8.85.14.1.19 sort
8.85.14.1.20 sync
8.85.14.1.21 updateItem
8.85.14.2 Properties
8.85.14.2.1 data
8.85.14.2.2 dataFeed
8.85.14.2.3 datathrottle
8.85.14.2.4 datatype
8.85.14.2.5 ready
8.85.14.2.6 removeMissed

8.85.14.2.7 save
8.85.14.2.8 scheme
8.85.14.2.9 url
8.85.14.3 Events
8.85.14.3.1 onAfterAdd
8.85.14.3.2 onAfterDelete
8.85.14.3.3 onAfterLoad
8.85.14.3.4 onAfterSort
8.85.14.3.5 onBeforeAdd
8.85.14.3.6 onBeforeDelete
8.85.14.3.7 onBeforeLoad
8.85.14.3.8 onBeforeSort
8.85.14.3.9 onDataRequest
8.85.14.3.10 onDataUpdate
8.85.14.3.11 onLoadError
8.85.15 DataMarks
8.85.15.1 Methods
8.85.15.1.1 addCss
8.85.15.1.2 clearCss
8.85.15.1.3 hasCss
8.85.15.1.4 removeCss
8.85.16 DataMove
8.85.16.1 Methods
8.85.16.1.1 copy
8.85.16.1.2 move
8.85.16.1.3 moveBottom
8.85.16.1.4 moveDown
8.85.16.1.5 moveTop
8.85.16.1.6 moveUp
8.85.16.2 Properties
8.85.16.2.1 externalData
8.85.17 DataProcessor
8.85.17.1 Methods
8.85.17.1.1 attachEvent
8.85.17.1.2 attachProgress
8.85.17.1.3 blockEvent
8.85.17.1.4 callEvent
8.85.17.1.5 clearValidation
8.85.17.1.6 define
8.85.17.1.7 detachEvent
8.85.17.1.8 escape
8.85.17.1.9 getItemState
8.85.17.1.10 getState
8.85.17.1.11 hasEvent
8.85.17.1.12 ignore
8.85.17.1.13 mapEvent
8.85.17.1.14 off
8.85.17.1.15 on
8.85.17.1.16 processResult
8.85.17.1.17 reset
8.85.17.1.18 save
8.85.17.1.19 send
8.85.17.1.20 setItemState
8.85.17.1.21 unblockEvent
8.85.17.1.22 validate
8.85.17.2 Properties
8.85.17.2.1 autoupdate
8.85.17.2.2 escape

8.85.17.2.3 id
8.85.17.2.4 master
8.85.17.2.5 mode
8.85.17.2.6 on
8.85.17.2.7 operationName
8.85.17.2.8 rules
8.85.17.2.9 store
8.85.17.2.10 trackMove
8.85.17.2.11 updateFromResponse
8.85.17.2.12 url
8.85.17.3 Events
8.85.17.3.1 onAfter
8.85.17.3.2 onAfterDataSend
8.85.17.3.3 onAfterDelete
8.85.17.3.4 onAfterInsert
8.85.17.3.5 onAfterSave
8.85.17.3.6 onAfterSaveError
8.85.17.3.7 onAfterSync
8.85.17.3.8 onAfterUpdate
8.85.17.3.9 onAfterValidation
8.85.17.3.10 onBefore
8.85.17.3.11 onBeforeDataSend
8.85.17.3.12 onBeforeDelete
8.85.17.3.13 onBeforeInsert
8.85.17.3.14 onBeforeSaveError
8.85.17.3.15 onBeforeSync
8.85.17.3.16 onBeforeUpdate
8.85.17.3.17 onBeforeValidate
8.85.17.3.18 onLoadError
8.85.17.3.19 onValidationError
8.85.17.3.20 onValidationSuccess
8.85.17.4 Others
8.85.17.4.1 config
8.85.17.4.2 name
8.85.18 DataRecord
8.85.18.1 Methods
8.85.18.1.1 attachEvent
8.85.18.1.2 bind
8.85.18.1.3 blockEvent
8.85.18.1.4 callEvent
8.85.18.1.5 define
8.85.18.1.6 detachEvent
8.85.18.1.7 getValues
8.85.18.1.8 hasEvent
8.85.18.1.9 isVisible
8.85.18.1.10 load
8.85.18.1.11 mapEvent
8.85.18.1.12 parse
8.85.18.1.13 refresh
8.85.18.1.14 setValues
8.85.18.1.15 unbind
8.85.18.1.16 unblockEvent
8.85.18.2 Properties
8.85.18.2.1 data
8.85.18.2.2 dataFeed
8.85.18.2.3 datatype
8.85.18.2.4 id
8.85.18.2.5 on

8.85.18.2.6 url
8.85.18.3 Events
8.85.18.3.1 onAfterLoad
8.85.18.3.2 onBeforeLoad
8.85.18.3.3 onBindRequest
8.85.18.3.4 onChange
8.85.18.3.5 onLoadError
8.85.18.4 Others
8.85.18.4.1 config
8.85.18.4.2 name
8.85.19 DataState
8.85.19.1 Methods
8.85.19.1.1 getState
8.85.19.1.2 setState
8.85.20 DataStore
8.85.20.1 Methods
8.85.20.1.1 add
8.85.20.1.2 addMark
8.85.20.1.3 attachEvent
8.85.20.1.4 blockEvent
8.85.20.1.5 callEvent
8.85.20.1.6 changeId
8.85.20.1.7 clearAll
8.85.20.1.8 clearMark
8.85.20.1.9 count
8.85.20.1.10 destructor
8.85.20.1.11 detachEvent
8.85.20.1.12 each
8.85.20.1.13 exists
8.85.20.1.14 filter
8.85.20.1.15 getFirstId
8.85.20.1.16 getIdByIndex
8.85.20.1.17 getIndexById
8.85.20.1.18 getIndexRange
8.85.20.1.19 getItem
8.85.20.1.20 getLastId
8.85.20.1.21 getMark
8.85.20.1.22 getNextId
8.85.20.1.23 getPrevId
8.85.20.1.24 getRange
8.85.20.1.25 hasEvent
8.85.20.1.26 id
8.85.20.1.27 importData
8.85.20.1.28 mapEvent
8.85.20.1.29 move
8.85.20.1.30 provideApi
8.85.20.1.31 refresh
8.85.20.1.32 remove
8.85.20.1.33 removeMark
8.85.20.1.34 scheme
8.85.20.1.35 serialize
8.85.20.1.36 setDriver
8.85.20.1.37 silent
8.85.20.1.38 sort
8.85.20.1.39 sync
8.85.20.1.40 unblockEvent
8.85.20.1.41 unsync
8.85.20.1.42 updateItem

8.85.20.2 Properties
8.85.20.2.1 on
8.85.20.3 Events
8.85.20.3.1 onAfterAdd
8.85.20.3.2 onAfterDelete
8.85.20.3.3 onAfterFilter
8.85.20.3.4 onAfterSort
8.85.20.3.5 onBeforeAdd
8.85.20.3.6 onBeforeDelete
8.85.20.3.7 onBeforeFilter
8.85.20.3.8 onBeforeSort
8.85.20.3.9 onClearAll
8.85.20.3.10 onDataUpdate
8.85.20.3.11 onIdChange
8.85.20.3.12 onParse
8.85.20.3.13 onServerConfig
8.85.20.3.14 onServerOptions
8.85.20.3.15 onStoreLoad
8.85.20.3.16 onStoreUpdated
8.85.20.3.17 onSyncApply
8.85.20.4 Others
8.85.20.4.1 driver
8.85.20.4.2 name
8.85.20.4.3 order
8.85.20.4.4 pull
8.85.21 DataValue
8.85.21.1 Methods
8.85.21.1.1 attachEvent
8.85.21.1.2 bind
8.85.21.1.3 blockEvent
8.85.21.1.4 callEvent
8.85.21.1.5 detachEvent
8.85.21.1.6 getValue
8.85.21.1.7 hasEvent
8.85.21.1.8 isVisible
8.85.21.1.9 mapEvent
8.85.21.1.10 refresh
8.85.21.1.11 setValue
8.85.21.1.12 unbind
8.85.21.1.13 unblockEvent
8.85.21.2 Properties
8.85.21.2.1 id
8.85.21.2.2 on
8.85.21.2.3 value
8.85.21.3 Events
8.85.21.3.1 onBindRequest
8.85.21.3.2 onChange
8.85.21.4 Others
8.85.21.4.1 name
8.85.22 Date
8.85.22.1 Methods
8.85.22.1.1 add
8.85.22.1.2 copy
8.85.22.1.3 datePart
8.85.22.1.4 dateToStr
8.85.22.1.5 dayStart
8.85.22.1.6 equal
8.85.22.1.7 getISOWeek

8.85.22.1.8 getUTCISOWeek
8.85.22.1.9 isHoliday
8.85.22.1.10 monthStart
8.85.22.1.11 strToDate
8.85.22.1.12 timePart
8.85.22.1.13 toFixed
8.85.22.1.14 weekStart
8.85.22.1.15 yearStart
8.85.22.2 Others
8.85.22.2.1 startOnMonday
8.85.23 Destruction
8.85.23.1 Methods
8.85.23.1.1 destructor
8.85.23.2 Events
8.85.23.2.1 onDestruct
8.85.24 DragControl
8.85.24.1 Methods
8.85.24.1.1 addDrag
8.85.24.1.2 addDrop
8.85.24.1.3 createDrag
8.85.24.1.4 destroyDrag
8.85.24.1.5 getContext
8.85.24.1.6 getMaster
8.85.24.1.7 getNode
8.85.24.1.8 sendSignal
8.85.24.2 Others
8.85.24.2.1 $drag
8.85.24.2.2 $dragIn
8.85.24.2.3 $dragOut
8.85.24.2.4 $dragPos
8.85.24.2.5 $drop
8.85.24.2.6 left
8.85.24.2.7 top
8.85.25 DragItem
8.85.25.1 Properties
8.85.25.1.1 drag
8.85.25.1.2 dragscroll
8.85.25.2 Events
8.85.25.2.1 onAfterDrop
8.85.25.2.2 onBeforeDrag
8.85.25.2.3 onBeforeDragIn
8.85.25.2.4 onBeforeDrop
8.85.25.2.5 onBeforeDropOut
8.85.25.2.6 onDragOut
8.85.25.3 Others
8.85.25.3.1 $drag
8.85.25.3.2 $dragHTML
8.85.25.3.3 $dragIn
8.85.25.3.4 $dragMark
8.85.25.3.5 $dragOut
8.85.25.3.6 $drop
8.85.25.3.7 $dropAllow
8.85.26 DragOrder
8.85.26.1 Events
8.85.26.1.1 onAfterDropOrder
8.85.26.1.2 onBeforeDropOrder
8.85.26.2 Others
8.85.26.2.1 $drag

8.85.26.2.2 $dragIn
8.85.26.2.3 $dragPos
8.85.26.2.4 $drop
8.85.27 EditAbility
8.85.27.1 Methods
8.85.27.1.1 edit
8.85.27.1.2 editCancel
8.85.27.1.3 editNext
8.85.27.1.4 editStop
8.85.27.1.5 focusEditor
8.85.27.1.6 getEditState
8.85.27.1.7 getEditor
8.85.27.1.8 getEditorValue
8.85.27.1.9 validateEditor
8.85.27.2 Properties
8.85.27.2.1 editValue
8.85.27.2.2 editable
8.85.27.2.3 editaction
8.85.27.2.4 editor
8.85.27.2.5 form
8.85.27.3 Events
8.85.27.3.1 onAfterEditStart
8.85.27.3.2 onAfterEditStop
8.85.27.3.3 onBeforeEditStart
8.85.27.3.4 onBeforeEditStop
8.85.27.3.5 onLiveEdit
8.85.28 EventSystem
8.85.28.1 Methods
8.85.28.1.1 attachEvent
8.85.28.1.2 blockEvent
8.85.28.1.3 callEvent
8.85.28.1.4 detachEvent
8.85.28.1.5 hasEvent
8.85.28.1.6 mapEvent
8.85.28.1.7 unblockEvent
8.85.28.2 Properties
8.85.28.2.1 on
8.85.29 Group
8.85.29.1 Methods
8.85.29.1.1 group
8.85.29.1.2 ungroup
8.85.30 GroupMethods
8.85.30.1 Methods
8.85.30.1.1 any
8.85.30.1.2 count
8.85.30.1.3 max
8.85.30.1.4 min
8.85.30.1.5 string
8.85.30.1.6 sum
8.85.31 GroupStore
8.85.31.1 Methods
8.85.31.1.1 group
8.85.31.1.2 ungroup
8.85.31.2 Events
8.85.31.2.1 onGroupCreated
8.85.31.2.2 onGroupFooter
8.85.31.2.3 onStoreUpdated
8.85.32 HtmlMap

8.85.32.1 Methods
8.85.32.1.1 addPoly
8.85.32.1.2 addRect
8.85.32.1.3 addSector
8.85.32.1.4 render
8.85.33 IdSpace
8.85.33.1 Methods
8.85.33.1.1 innerId
8.85.33.1.2 ui
8.85.33.2 Others
8.85.33.2.1 $$
8.85.34 KeysNavigation
8.85.34.1 Methods
8.85.34.1.1 moveSelection
8.85.34.2 Properties
8.85.34.2.1 navigation
8.85.35 MapCollection
8.85.36 Modality
8.85.36.1 Properties
8.85.36.1.1 zIndex
8.85.37 MouseEvents
8.85.37.1 Properties
8.85.37.1.1 mouseEventDelay
8.85.37.1.2 onClick
8.85.37.1.3 onContext
8.85.37.1.4 onDblClick
8.85.37.1.5 onMouseMove
8.85.37.2 Events
8.85.37.2.1 onAfterContextMenu
8.85.37.2.2 onBeforeContextMenu
8.85.37.2.3 onItemClick
8.85.37.2.4 onItemDblClick
8.85.37.2.5 onMouseMove
8.85.37.2.6 onMouseMoving
8.85.37.2.7 onMouseOut
8.85.37.3 Others
8.85.37.3.1 on_click
8.85.37.3.2 on_context
8.85.37.3.3 on_dblclick
8.85.37.3.4 on_mouse_move
8.85.38 Movable
8.85.38.1 Properties
8.85.38.1.1 move
8.85.38.2 Events
8.85.38.2.1 onViewMove
8.85.38.2.2 onViewMoveEnd
8.85.39 NavigationButtons
8.85.39.1 Properties
8.85.39.1.1 navigation
8.85.40 Number
8.85.40.1 Methods
8.85.40.1.1 format
8.85.40.1.2 numToStr
8.85.41 OverlayBox
8.85.41.1 Methods
8.85.41.1.1 hideOverlay
8.85.41.1.2 showOverlay
8.85.42 PagingAbility

8.85.42.1 Methods
8.85.42.1.1 getPage
8.85.42.1.2 getPager
8.85.42.1.3 setPage
8.85.42.2 Properties
8.85.42.2.1 pager
8.85.43 PowerArray
8.85.43.1 Methods
8.85.43.1.1 each
8.85.43.1.2 filter
8.85.43.1.3 find
8.85.43.1.4 insertAt
8.85.43.1.5 map
8.85.43.1.6 remove
8.85.43.1.7 removeAt
8.85.44 ProgressBar
8.85.44.1 Methods
8.85.44.1.1 hideProgress
8.85.44.1.2 showProgress
8.85.45 RecordBind
8.85.46 RenderStack
8.85.46.1 Methods
8.85.46.1.1 customize
8.85.46.1.2 getItemNode
8.85.46.1.3 locate
8.85.46.1.4 render
8.85.46.1.5 showItem
8.85.46.2 Properties
8.85.46.2.1 click
8.85.46.2.2 template
8.85.46.2.3 type
8.85.46.3 Events
8.85.46.3.1 onAfterRender
8.85.46.3.2 onBeforeRender
8.85.46.3.3 onItemRender
8.85.46.4 Others
8.85.46.4.1 type
8.85.46.4.2 types
8.85.47 Scrollable
8.85.47.1 Methods
8.85.47.1.1 getScrollState
8.85.47.1.2 scrollTo
8.85.47.2 Properties
8.85.47.2.1 scroll
8.85.47.2.2 scrollSpeed
8.85.48 SelectionModel
8.85.48.1 Methods
8.85.48.1.1 getSelectedId
8.85.48.1.2 getSelectedItem
8.85.48.1.3 isSelected
8.85.48.1.4 select
8.85.48.1.5 selectAll
8.85.48.1.6 unselect
8.85.48.1.7 unselectAll
8.85.48.2 Properties
8.85.48.2.1 multiselect
8.85.48.2.2 select
8.85.48.3 Events

8.85.48.3.1 onAfterSelect
8.85.48.3.2 onBeforeSelect
8.85.48.3.3 onSelectChange
8.85.49 Settings
8.85.49.1 Methods
8.85.49.1.1 define
8.85.49.2 Properties
8.85.49.2.1 id
8.85.49.3 Others
8.85.49.3.1 config
8.85.49.3.2 name
8.85.50 SingleRender
8.85.50.1 Methods
8.85.50.1.1 customize
8.85.50.1.2 render
8.85.50.1.3 sync
8.85.50.2 Properties
8.85.50.2.1 content
8.85.50.2.2 template
8.85.50.3 Events
8.85.50.3.1 onAfterRender
8.85.50.3.2 onBeforeRender
8.85.50.4 Others
8.85.50.4.1 type
8.85.51 TablePaste
8.85.51.1 Properties
8.85.51.1.1 clipboard
8.85.51.1.2 delimiter
8.85.52 Touch
8.85.52.1 Methods
8.85.52.1.1 disable
8.85.52.1.2 enable
8.85.52.1.3 limit
8.85.52.1.4 scrollTo
8.85.52.2 Others
8.85.52.2.1 config
8.85.53 TreeAPI
8.85.53.1 Methods
8.85.53.1.1 close
8.85.53.1.2 closeAll
8.85.53.1.3 getOpenItems
8.85.53.1.4 getState
8.85.53.1.5 isBranchOpen
8.85.53.1.6 open
8.85.53.1.7 openAll
8.85.53.1.8 setState
8.85.53.2 Properties
8.85.53.2.1 threeState
8.85.53.3 Events
8.85.53.3.1 onAfterClose
8.85.53.3.2 onAfterOpen
8.85.53.3.3 onBeforeClose
8.85.53.3.4 onBeforeOpen
8.85.53.3.5 onItemCheck
8.85.54 TreeClick
8.85.54.1 Methods
8.85.54.1.1 webix_tree_checkbox
8.85.54.1.2 webix_tree_close

8.85.54.1.3 webix_tree_open
8.85.55 TreeCollection
8.85.55.1 Methods
8.85.55.1.1 add
8.85.55.1.2 addBind
8.85.55.1.3 attachEvent
8.85.55.1.4 bind
8.85.55.1.5 blockEvent
8.85.55.1.6 callEvent
8.85.55.1.7 clearAll
8.85.55.1.8 clearValidation
8.85.55.1.9 count
8.85.55.1.10 define
8.85.55.1.11 destructor
8.85.55.1.12 detachEvent
8.85.55.1.13 exists
8.85.55.1.14 filter
8.85.55.1.15 getBindData
8.85.55.1.16 getBranchIndex
8.85.55.1.17 getCursor
8.85.55.1.18 getFirstChildId
8.85.55.1.19 getFirstId
8.85.55.1.20 getIdByIndex
8.85.55.1.21 getIndexById
8.85.55.1.22 getItem
8.85.55.1.23 getLastId
8.85.55.1.24 getNextId
8.85.55.1.25 getNextSiblingId
8.85.55.1.26 getParentId
8.85.55.1.27 getPrevId
8.85.55.1.28 getPrevSiblingId
8.85.55.1.29 hasEvent
8.85.55.1.30 isBranch
8.85.55.1.31 isVisible
8.85.55.1.32 load
8.85.55.1.33 loadBranch
8.85.55.1.34 loadNext
8.85.55.1.35 mapEvent
8.85.55.1.36 parse
8.85.55.1.37 refresh
8.85.55.1.38 refreshCursor
8.85.55.1.39 remove
8.85.55.1.40 removeBind
8.85.55.1.41 saveBatch
8.85.55.1.42 serialize
8.85.55.1.43 setBindData
8.85.55.1.44 setCursor
8.85.55.1.45 sort
8.85.55.1.46 sync
8.85.55.1.47 unbind
8.85.55.1.48 unblockEvent
8.85.55.1.49 updateItem
8.85.55.1.50 validate
8.85.55.2 Properties
8.85.55.2.1 data
8.85.55.2.2 dataFeed
8.85.55.2.3 datathrottle
8.85.55.2.4 datatype

8.85.55.2.5 defaultData
8.85.55.2.6 filterMode
8.85.55.2.7 id
8.85.55.2.8 on
8.85.55.2.9 ready
8.85.55.2.10 removeMissed
8.85.55.2.11 rules
8.85.55.2.12 save
8.85.55.2.13 scheme
8.85.55.2.14 url
8.85.55.3 Events
8.85.55.3.1 onAfterAdd
8.85.55.3.2 onAfterCursorChange
8.85.55.3.3 onAfterDelete
8.85.55.3.4 onAfterLoad
8.85.55.3.5 onAfterSort
8.85.55.3.6 onBeforeAdd
8.85.55.3.7 onBeforeCursorChange
8.85.55.3.8 onBeforeDelete
8.85.55.3.9 onBeforeLoad
8.85.55.3.10 onBeforeSort
8.85.55.3.11 onBindRequest
8.85.55.3.12 onBindUpdate
8.85.55.3.13 onDataRequest
8.85.55.3.14 onDataUpdate
8.85.55.3.15 onDestruct
8.85.55.3.16 onLoadError
8.85.55.3.17 onValidationError
8.85.55.3.18 onValidationSuccess
8.85.55.4 Others
8.85.55.4.1 config
8.85.55.4.2 name
8.85.56 TreeDataLoader
8.85.56.1 Methods
8.85.56.1.1 loadBranch
8.85.56.2 Properties
8.85.56.2.1 filterMode
8.85.56.3 Events
8.85.56.3.1 onAfterAdd
8.85.56.3.2 onAfterDelete
8.85.56.3.3 onAfterSort
8.85.56.3.4 onBeforeAdd
8.85.56.3.5 onBeforeDelete
8.85.56.3.6 onBeforeSort
8.85.56.3.7 onDataRequest
8.85.56.3.8 onDataUpdate
8.85.57 TreeDataMove
8.85.57.1 Methods
8.85.57.1.1 copy
8.85.57.1.2 move
8.85.57.2 Others
8.85.57.2.1 $dropAllow
8.85.58 TreeRenderStack
8.85.58.1 Methods
8.85.58.1.1 getItemNode
8.85.58.1.2 getItemNode
8.85.58.2 Events
8.85.58.2.1 onAfterRender

8.85.58.2.2 onBeforeRender
8.85.58.2.3 onItemRender
8.85.58.2.4 onPartialRender
8.85.59 TreeStateCheckbox
8.85.59.1 Methods
8.85.59.1.1 checkAll
8.85.59.1.2 checkItem
8.85.59.1.3 getChecked
8.85.59.1.4 isChecked
8.85.59.1.5 uncheckAll
8.85.59.1.6 uncheckItem
8.85.59.2 Properties
8.85.59.2.1 threeState
8.85.59.3 Events
8.85.59.3.1 onItemCheck
8.85.60 TreeStore
8.85.60.1 Methods
8.85.60.1.1 add
8.85.60.1.2 changeId
8.85.60.1.3 clearAll
8.85.60.1.4 count
8.85.60.1.5 each
8.85.60.1.6 eachChild
8.85.60.1.7 eachOpen
8.85.60.1.8 eachSubItem
8.85.60.1.9 getBranch
8.85.60.1.10 getBranchIndex
8.85.60.1.11 getFirstChildId
8.85.60.1.12 getNextSiblingId
8.85.60.1.13 getParentId
8.85.60.1.14 getPrevSiblingId
8.85.60.1.15 getTopRange
8.85.60.1.16 isBranch
8.85.60.1.17 provideApi
8.85.60.1.18 remove
8.85.60.1.19 serialize
8.85.60.2 Properties
8.85.60.2.1 filterMode
8.85.60.3 Events
8.85.60.3.1 onAfterDelete
8.85.60.3.2 onBeforeDelete
8.85.60.3.3 onStoreUpdated
8.85.60.4 Others
8.85.60.4.1 name
8.85.61 TreeTableClick
8.85.62 TreeTablePaste
8.85.62.1 Methods
8.85.62.1.1 insert
8.85.63 TreeType
8.85.63.1 Methods
8.85.63.1.1 checkbox
8.85.63.1.2 folder
8.85.63.1.3 icon
8.85.63.1.4 space
8.85.64 UIExtension
8.85.65 UIManager
8.85.65.1 Methods
8.85.65.1.1 addHotKey

8.85.65.1.2 canFocus
8.85.65.1.3 destructor
8.85.65.1.4 getFocus
8.85.65.1.5 getNext
8.85.65.1.6 getPrev
8.85.65.1.7 getState
8.85.65.1.8 getTop
8.85.65.1.9 hasFocus
8.85.65.1.10 removeHotKey
8.85.65.1.11 setFocus
8.85.65.1.12 setState
8.85.66 UploadDriver
8.85.66.1 Others
8.85.66.1.1 flash
8.85.66.1.2 html5
8.85.67 ValidateCollection
8.85.67.1 Methods
8.85.67.1.1 clearValidation
8.85.67.1.2 validate
8.85.67.2 Properties
8.85.67.2.1 rules
8.85.67.3 Events
8.85.67.3.1 onValidationError
8.85.67.3.2 onValidationSuccess
8.85.68 ValidateData
8.85.68.1 Methods
8.85.68.1.1 clearValidation
8.85.68.1.2 validate
8.85.68.2 Properties
8.85.68.2.1 rules
8.85.68.3 Events
8.85.68.3.1 onAfterValidation
8.85.68.3.2 onBeforeValidate
8.85.68.3.3 onValidationError
8.85.68.3.4 onValidationSuccess
8.85.69 ValueBind
8.85.70 Values
8.85.70.1 Methods
8.85.70.1.1 clear
8.85.70.1.2 focus
8.85.70.1.3 getCleanValues
8.85.70.1.4 getDirtyValues
8.85.70.1.5 getValues
8.85.70.1.6 isDirty
8.85.70.1.7 setDirty
8.85.70.1.8 setValues
8.85.70.2 Events
8.85.70.2.1 onAfterLoad
8.85.70.2.2 onChange
8.85.70.2.3 onValues
8.85.71 VirtualRenderStack
8.85.71.1 Methods
8.85.71.1.1 getItemNode
8.85.71.1.2 render
8.85.71.1.3 showItem
8.85.71.2 Properties
8.85.71.2.1 datafetch
8.85.71.2.2 loadahead

8.85.71.3 Events
8.85.71.3.1 onAfterRender
8.85.71.3.2 onBeforeRender
8.85.72 ui.baselayout
8.85.72.1 Methods
8.85.72.1.1 addView
8.85.72.1.2 adjust
8.85.72.1.3 attachEvent
8.85.72.1.4 bind
8.85.72.1.5 blockEvent
8.85.72.1.6 callEvent
8.85.72.1.7 define
8.85.72.1.8 destructor
8.85.72.1.9 detachEvent
8.85.72.1.10 disable
8.85.72.1.11 enable
8.85.72.1.12 getChildViews
8.85.72.1.13 getFormView
8.85.72.1.14 getNode
8.85.72.1.15 getParentView
8.85.72.1.16 getTopParentView
8.85.72.1.17 hasEvent
8.85.72.1.18 hide
8.85.72.1.19 index
8.85.72.1.20 isEnabled
8.85.72.1.21 isVisible
8.85.72.1.22 mapEvent
8.85.72.1.23 reconstruct
8.85.72.1.24 removeView
8.85.72.1.25 resize
8.85.72.1.26 resizeChildren
8.85.72.1.27 show
8.85.72.1.28 showBatch
8.85.72.1.29 unbind
8.85.72.1.30 unblockEvent
8.85.72.2 Properties
8.85.72.2.1 animate
8.85.72.2.2 borderless
8.85.72.2.3 cols
8.85.72.2.4 container
8.85.72.2.5 css
8.85.72.2.6 disabled
8.85.72.2.7 gravity
8.85.72.2.8 height
8.85.72.2.9 hidden
8.85.72.2.10 id
8.85.72.2.11 maxHeight
8.85.72.2.12 maxWidth
8.85.72.2.13 minHeight
8.85.72.2.14 minWidth
8.85.72.2.15 on
8.85.72.2.16 responsive
8.85.72.2.17 rows
8.85.72.2.18 visibleBatch
8.85.72.2.19 width
8.85.72.3 Events
8.85.72.3.1 onBindRequest
8.85.72.3.2 onDestruct

8.85.72.4 Others
8.85.72.4.1 $getSize
8.85.72.4.2 $height
8.85.72.4.3 $setSize
8.85.72.4.4 $skin
8.85.72.4.5 $view
8.85.72.4.6 $width
8.85.72.4.7 config
8.85.72.4.8 name
8.85.73 ui.baseview
8.85.73.1 Methods
8.85.73.1.1 adjust
8.85.73.1.2 bind
8.85.73.1.3 define
8.85.73.1.4 destructor
8.85.73.1.5 disable
8.85.73.1.6 enable
8.85.73.1.7 getChildViews
8.85.73.1.8 getFormView
8.85.73.1.9 getNode
8.85.73.1.10 getParentView
8.85.73.1.11 getTopParentView
8.85.73.1.12 hide
8.85.73.1.13 isEnabled
8.85.73.1.14 isVisible
8.85.73.1.15 resize
8.85.73.1.16 show
8.85.73.1.17 unbind
8.85.73.2 Properties
8.85.73.2.1 animate
8.85.73.2.2 borderless
8.85.73.2.3 container
8.85.73.2.4 css
8.85.73.2.5 disabled
8.85.73.2.6 gravity
8.85.73.2.7 height
8.85.73.2.8 hidden
8.85.73.2.9 id
8.85.73.2.10 maxHeight
8.85.73.2.11 maxWidth
8.85.73.2.12 minHeight
8.85.73.2.13 minWidth
8.85.73.2.14 width
8.85.73.3 Events
8.85.73.3.1 onBindRequest
8.85.73.3.2 onDestruct
8.85.73.4 Others
8.85.73.4.1 $getSize
8.85.73.4.2 $height
8.85.73.4.3 $setSize
8.85.73.4.4 $skin
8.85.73.4.5 $view
8.85.73.4.6 $width
8.85.73.4.7 config
8.85.73.4.8 name
8.85.74 ui.proto
8.85.74.1 Methods
8.85.74.1.1 add

8.85.74.1.2 addCss
8.85.74.1.3 adjust
8.85.74.1.4 attachEvent
8.85.74.1.5 bind
8.85.74.1.6 blockEvent
8.85.74.1.7 callEvent
8.85.74.1.8 clearAll
8.85.74.1.9 clearCss
8.85.74.1.10 clearValidation
8.85.74.1.11 count
8.85.74.1.12 customize
8.85.74.1.13 define
8.85.74.1.14 destructor
8.85.74.1.15 detachEvent
8.85.74.1.16 disable
8.85.74.1.17 enable
8.85.74.1.18 exists
8.85.74.1.19 filter
8.85.74.1.20 getChildViews
8.85.74.1.21 getFirstId
8.85.74.1.22 getFormView
8.85.74.1.23 getIdByIndex
8.85.74.1.24 getIndexById
8.85.74.1.25 getItem
8.85.74.1.26 getItemNode
8.85.74.1.27 getLastId
8.85.74.1.28 getNextId
8.85.74.1.29 getNode
8.85.74.1.30 getPage
8.85.74.1.31 getPager
8.85.74.1.32 getParentView
8.85.74.1.33 getPrevId
8.85.74.1.34 getTopParentView
8.85.74.1.35 hasCss
8.85.74.1.36 hasEvent
8.85.74.1.37 hide
8.85.74.1.38 isEnabled
8.85.74.1.39 isVisible
8.85.74.1.40 load
8.85.74.1.41 loadNext
8.85.74.1.42 locate
8.85.74.1.43 mapEvent
8.85.74.1.44 parse
8.85.74.1.45 refresh
8.85.74.1.46 remove
8.85.74.1.47 removeCss
8.85.74.1.48 render
8.85.74.1.49 resize
8.85.74.1.50 serialize
8.85.74.1.51 setPage
8.85.74.1.52 show
8.85.74.1.53 showItem
8.85.74.1.54 sort
8.85.74.1.55 sync
8.85.74.1.56 unbind
8.85.74.1.57 unblockEvent
8.85.74.1.58 updateItem
8.85.74.1.59 validate

8.85.74.2 Properties
8.85.74.2.1 animate
8.85.74.2.2 borderless
8.85.74.2.3 click
8.85.74.2.4 container
8.85.74.2.5 css
8.85.74.2.6 data
8.85.74.2.7 dataFeed
8.85.74.2.8 datathrottle
8.85.74.2.9 datatype
8.85.74.2.10 disabled
8.85.74.2.11 gravity
8.85.74.2.12 height
8.85.74.2.13 hidden
8.85.74.2.14 id
8.85.74.2.15 maxHeight
8.85.74.2.16 maxWidth
8.85.74.2.17 minHeight
8.85.74.2.18 minWidth
8.85.74.2.19 on
8.85.74.2.20 pager
8.85.74.2.21 ready
8.85.74.2.22 removeMissed
8.85.74.2.23 rules
8.85.74.2.24 save
8.85.74.2.25 scheme
8.85.74.2.26 template
8.85.74.2.27 tooltip
8.85.74.2.28 type
8.85.74.2.29 url
8.85.74.2.30 width
8.85.74.3 Events
8.85.74.3.1 onAfterAdd
8.85.74.3.2 onAfterDelete
8.85.74.3.3 onAfterLoad
8.85.74.3.4 onAfterRender
8.85.74.3.5 onAfterScroll
8.85.74.3.6 onAfterSort
8.85.74.3.7 onBeforeAdd
8.85.74.3.8 onBeforeDelete
8.85.74.3.9 onBeforeLoad
8.85.74.3.10 onBeforeRender
8.85.74.3.11 onBeforeSort
8.85.74.3.12 onBindRequest
8.85.74.3.13 onDataRequest
8.85.74.3.14 onDataUpdate
8.85.74.3.15 onDestruct
8.85.74.3.16 onItemRender
8.85.74.3.17 onKeyPress
8.85.74.3.18 onLoadError
8.85.74.3.19 onLongTouch
8.85.74.3.20 onSwipeX
8.85.74.3.21 onSwipeY
8.85.74.3.22 onTimedKeyPress
8.85.74.3.23 onTouchEnd
8.85.74.3.24 onTouchMove
8.85.74.3.25 onTouchStart
8.85.74.3.26 onValidationError

8.85.74.3.27 onValidationSuccess
8.85.74.3.28 onViewResize
8.85.74.4 Others
8.85.74.4.1 $getSize
8.85.74.4.2 $height
8.85.74.4.3 $scope
8.85.74.4.4 $setSize
8.85.74.4.5 $skin
8.85.74.4.6 $view
8.85.74.4.7 $width
8.85.74.4.8 config
8.85.74.4.9 name
8.85.74.4.10 type
8.85.74.4.11 types
8.85.75 ui.resizearea
8.85.75.1 Methods
8.85.75.1.1 attachEvent
8.85.75.1.2 blockEvent
8.85.75.1.3 callEvent
8.85.75.1.4 define
8.85.75.1.5 detachEvent
8.85.75.1.6 hasEvent
8.85.75.1.7 mapEvent
8.85.75.1.8 unblockEvent
8.85.75.2 Properties
8.85.75.2.1 border
8.85.75.2.2 container
8.85.75.2.3 cursor
8.85.75.2.4 dir
8.85.75.2.5 eventPos
8.85.75.2.6 height
8.85.75.2.7 id
8.85.75.2.8 on
8.85.75.2.9 start
8.85.75.2.10 width
8.85.75.3 Events
8.85.75.3.1 onResize
8.85.75.3.2 onResizeEnd
8.85.75.4 Others
8.85.75.4.1 config
8.85.75.4.2 name
8.85.76 ui.view
8.85.76.1 Methods
8.85.76.1.1 adjust
8.85.76.1.2 bind
8.85.76.1.3 define
8.85.76.1.4 destructor
8.85.76.1.5 disable
8.85.76.1.6 enable
8.85.76.1.7 getChildViews
8.85.76.1.8 getFormView
8.85.76.1.9 getNode
8.85.76.1.10 getParentView
8.85.76.1.11 getTopParentView
8.85.76.1.12 hide
8.85.76.1.13 isEnabled
8.85.76.1.14 isVisible
8.85.76.1.15 resize

8.85.76.1.16 show
8.85.76.1.17 unbind
8.85.76.2 Properties
8.85.76.2.1 animate
8.85.76.2.2 borderless
8.85.76.2.3 container
8.85.76.2.4 css
8.85.76.2.5 disabled
8.85.76.2.6 gravity
8.85.76.2.7 height
8.85.76.2.8 hidden
8.85.76.2.9 id
8.85.76.2.10 maxHeight
8.85.76.2.11 maxWidth
8.85.76.2.12 minHeight
8.85.76.2.13 minWidth
8.85.76.2.14 width
8.85.76.3 Events
8.85.76.3.1 onAfterScroll
8.85.76.3.2 onBindRequest
8.85.76.3.3 onDestruct
8.85.76.3.4 onKeyPress
8.85.76.3.5 onLongTouch
8.85.76.3.6 onSwipeX
8.85.76.3.7 onSwipeY
8.85.76.3.8 onTimedKeyPress
8.85.76.3.9 onTouchEnd
8.85.76.3.10 onTouchMove
8.85.76.3.11 onTouchStart
8.85.76.3.12 onViewResize
8.85.76.4 Others
8.85.76.4.1 $getSize
8.85.76.4.2 $height
8.85.76.4.3 $scope
8.85.76.4.4 $setSize
8.85.76.4.5 $skin
8.85.76.4.6 $view
8.85.76.4.7 $width
8.85.76.4.8 config
8.85.76.4.9 name
8.85.77 ui.vscroll
8.85.77.1 Methods
8.85.77.1.1 activeArea
8.85.77.1.2 attachEvent
8.85.77.1.3 blockEvent
8.85.77.1.4 callEvent
8.85.77.1.5 define
8.85.77.1.6 detachEvent
8.85.77.1.7 getScroll
8.85.77.1.8 getSize
8.85.77.1.9 hasEvent
8.85.77.1.10 mapEvent
8.85.77.1.11 scrollTo
8.85.77.1.12 sizeTo
8.85.77.1.13 unblockEvent
8.85.77.2 Properties
8.85.77.2.1 container
8.85.77.2.2 id

8.85.77.2.3 on
8.85.77.2.4 scroll
8.85.77.2.5 scrollHeight
8.85.77.2.6 scrollPos
8.85.77.2.7 scrollSize
8.85.77.2.8 scrollStep
8.85.77.2.9 scrollVisible
8.85.77.2.10 scrollWidth
8.85.77.2.11 zoom
8.85.77.3 Events
8.85.77.3.1 onScroll
8.85.77.4 Others
8.85.77.4.1 config
8.85.77.4.2 name
9 Migration to latest Webix version
9.1 Migration from DHTMLX Touch
9.1.1 Additional Migration Changes

Overview

Webix is a cross-browser, cross-device JavaScript framework for building interactive web applications using JavaScript, CSS, HTML5 techniques. The library delivers over 40 fully customizable components, provides JQuery integration and can work with any server-side platform: PHP, .NET, Java, etc. Flexible layouts, unlimited extensibility, controllable look-and-feel, and intuitive API allow creating apps of any difficulty that will work equally well on desktop, mobile and tablet devices.

Main features

1. Rich UI
   The library supplies over 40 fully customizable components such as form, tree, grid, list etc.

2. Supported data formats: xml, json, jsarray, csv
   Developers can use their favourite formats to specify data. All popular data formats are supported.

3. Intuitive JS OOP
   Simple and light weight code provides easy and understandable OOP concepts.

4. Simple integration with server side
   Some platforms (PHP, .NET, Java) have ready solutions for data loading, saving, exporting.

5. Full API support
   A full range of API with the detailed descriptions is available.

6. Unlimited extensibility
   Webix doesn't limit developers in functionality and provides abundant possibilities in extending existent components and creating new ones.

7. Support for internationalization and localization
   Support for internationalization allows applications be translated and modified into local versions for hundreds of languages.

8. Styling with standard css
   You can use CSS in conjunction with components content to achieve the desired look-and-feel. Css can be applied to the whole layout or to specific components, items etc.

9. Flexible layouts
   Each element in the layout can be configured individually or as a part of layout: width, height, color etc.

10. On-the-fly filtering and sorting
    The library offers rich means of filtering and sorting, starting from built-in tools and ending with fully custom filtering logic.

11. Possibility to show or hide components
    Any component in the layout can be hidden (showed) programmatically.

12. Integration with jQuery

Webix is integrated with jQuery that allows you to follow the main mantra of the jQuery library while developing - running with the least amount of configuration.

13. Dynamically fetching data from the server when the component is scrolled up and down

    While working with large datasets, a developer can load data dynamically to reduce complexity and improve performance. Polling interval and the number of loaded records are controlled.

14. Callbacks for all events: load, sorting, etc.

    Numerous events let you handle any action or proceeding that can happen in the app.

15. Inline editing

    All data containing components can be directly edited on the page, saving users from having to go to a separate page first.

16. Drag-and-drop

    Users have the possibility to drag-and-drop items of the components. The library supports both external and internal dnd and allows customizing 'drag', 'drop' operations.

17. Integration with third party libraries

    Webix can be integrated with various third party libraries: Backbone.js library, Google, Nokia and Yandex maps, Mercury, NicEdit, Tinymce and Code Mirror text editors, etc.

18. Copying to/from clipboard

    By common CTRL+C/CTRL+V keyboard shortcuts data can be copied to some other application for further processing or moved within one or several components.

19. Offline support

    Offline support allows your apps run and be updated offline and don't depend on the state of the network.

20. Data binding

    Data binding is the technique that allows you to bind several components together and keep them in sync, i.e. when the data in some of components changes its value, the remaining components are automatically updated to reflect the change.

21. Variable length and localizable pagination

    One or several pagers can be added to the app to display data page by page. Data will not be retrieved from server side until the related page is opened. The number of rows per page, templates for navigation controls are set up.

Getting Started

Webix UI



Quick Start Guide

This tutorial gives you a detailed instruction how to make your first steps with Webix.



Start Coding with Webix

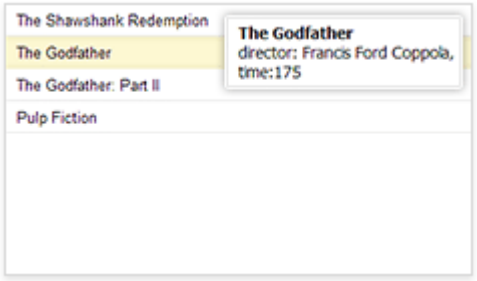
This tutorial introduces basic principles of using the Webix library for web app development.



Creating basic app

This tutorial discusses the main points of creating a simple app with the Webix library:

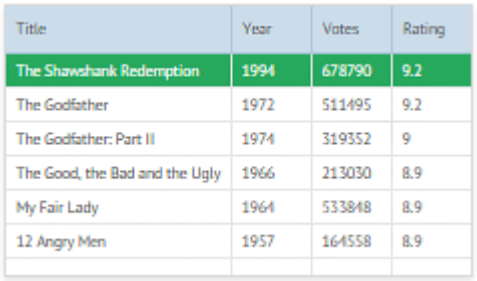
2. Setting Components
3. Interaction Between App Elements
4. Server-Side Integration



Introduction to Events

This tutorial explains the essence of Webix events model and discusses the basics of event processing.

Webix DataTable

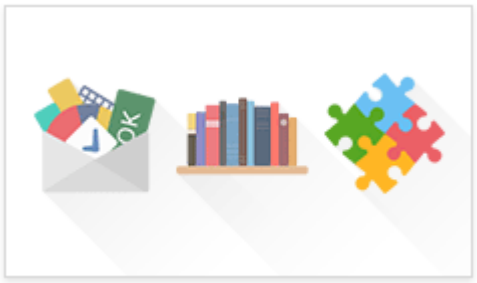


Title	Year	Votes	Rating
The Shawshank Redemption	1994	678790	9.2
The Godfather	1972	511495	9.2
The Godfather: Part II	1974	319352	9
The Good, the Bad and the Ugly	1966	213030	8.9
My Fair Lady	1964	533848	8.9
12 Angry Men	1957	164558	8.9

Quick Start with Datatable

This tutorial teaches how to create a simple app containing DataTable and populate it with data.

Webix Documentation



Webix Docs Guide

This chapter guides you through the contents of Webix documentation and gives a short description of each section.

Quick Start Guide

Fast start

The following steps will take you to your own Webix based app.

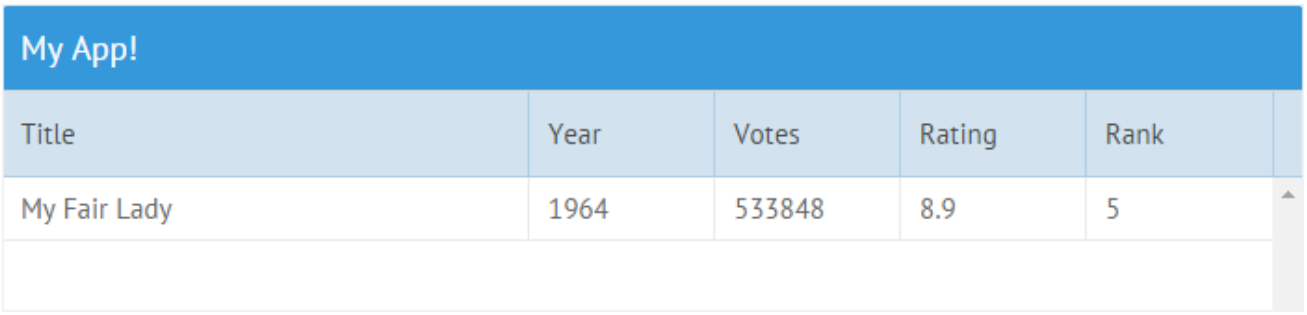
- Create a simple html-page. Specify **HTML5** doctype for it.

```
<!DOCTYPE HTML>
<html>
    <head>
    <link rel="stylesheet" href="http://cdn.webix.com/edge/webix.css"
type="text/css">
    <script src="http://cdn.webix.com/edge/webix.js"
type="text/javascript"></script>
    </head>
    <body>
        <script type="text/javascript" charset="utf-8">
    /* place for UI configuration */
</script>
</body>
</html>
```

- Define the UI configuration

```
webix.ui({
  rows:[
      { view:"template",
        type:"header", template:"My App!" },
      { view:"datatable",
        autoConfig:true,
        data:{
          title:"My Fair Lady", year:1964, votes:533848, rating:8.9,
rank:5
        }
      }
  ]
});
```

And **that is all**. Just open the page in a browser and you have a working mini app.



What's next

- Read a more detailed introduction to the library

or check one of interactive tutorials

- Interactive step-by-step tutorial
- Interactive DataTable tutorial
- Interactive Form tutorial

Beginner

Introduction to Webix

Including library

1 . Create a simple html-page. Specify **HTML5** doctype for it.

2 . Install Webix for your future app. There can be several ways:

Use CDN

Include 2 Webix files (**webix.js** and **webix.css**) directly from Webix CDN

```
<!DOCTYPE HTML>
<html>
    <head>
    <link rel="stylesheet" href="http://cdn.webix.com/edge/webix.css"
type="text/css">
    <script src="http://cdn.webix.com/edge/webix.js"
type="text/javascript"></script>
    </head>
    <body>
        <script type="text/javascript" charset="utf-8">

        // here your app will "take shelter"
</script>
</body>
</html>
```

If you need to get one of the older Webix versions, you shoud include the necessary number of version into the links to the code files:

```
<link rel="stylesheet" href="http://cdn.webix.com/2.2/webix.css"
type="text/css">
<script src="http://cdn.webix.com/2.2/webix.js"
type="text/javascript"></script>
```

Use Bower/NuGet

Install local Webix files via **Nuget** or **Bower** package managers by a single command. You don't need to download anything.

```
NuGet

nuget install Webix
```

```
//If you use Microsoft Visual Studio, execute this from Package Manager Console
install-package Webix
```

```
Bower

bower install webix
```

Download Webix package

Download and unzip the library to the necessary directory. Include **webix.js** and **webix.css** files into the document's head section.

```
<!DOCTYPE HTML>
<html>
    <head>
    <link rel="stylesheet" href="../../codebase/webix.css"
type="text/css">
    <script src="../../codebase/webix.js" type="text/javascript"></script>
    </head>
    <body>
        <script type="text/javascript" charset="utf-8">

 ... // here your app will "take shelter"
</script>
</body>
</html>
```

UI initialization

3 . The apps are to be placed between the *script* tags. The components you create are initialized inside the view.

```
Just a clean page

webix.ui({
  // components
});
```

Here you should be aware of the following:

- to ensure that your code is executed after the page is fully loaded, you can make use of the **webix.ready(function(){ ....})**, which is our alternative to the onDocumentReady event;

```
webix.ready(function(){
  webix.ui({
      ....
  });
```

```
});
```

Next we'll declare base elements, create the app layout and finally nest it with ui components.

4 . At this step we'll specify an HTML container for the app with the following properties:

```
<style>
 #app_here{
   width:1000px; height:400px; margin:20px;
 }

</style>
...
<div id="app_here"></div>
```

Then we'll create a layout structure which will be put into the above created container:

- a layout with 2 rows
- the 2nd row will contain one more layout with 3 columns

```
webix.ui({
  container:"app_here",
  view:"layout", // optional
    rows:[
      {...},
        { view:"layout", // optional
         cols:[
           {...},
           {...},
           {...}
        ]}
    ]
});
```

As the value of the **container** property the id of HTML element should be specified.

Note that you can easily omit the **view:"layout"** lines and start setting rows and columns at once.

5 . Now we can insert some components into the layout view.

We'll put a template that will serve as the app's header into the 1st row.

```
webix.ui({
  container:"app_here",
    rows:[
      { view:"template", type:"header", template:"My App!"},
        { cols:[
           {...},
           {...},
           {...}
```

```
        ]}
    ]
});
```

- you can easily omit the **view:"template"** definition;
- the "header" type of template is used to set headers for layout rows/columns;

6 . It's high time to place components into the 2nd row. The 3 columns of the 2nd row will be occupied by Tree and DataTable components and a resizer line between them:

```
webix.ui({
  container:"app_here",
  rows:[
     { type:"header", template:"My App!" },
     { cols:[
      { view:"tree", id:"mytree", gravity:0.3, select:true, data:tree_data
},
      { view:"resizer" },
      { view:"datatable", id:"mydatatable", autoConfig:true,
data:small_film_set }
    ]}
     ]
});
```

Let's look at the above code in detail:

- one of the key properties of any component and any item is its **ID**, the unique string or number that helps us refer to the object. **ID** should be specified if you are going to work with this component/item later as all functions are applied and events are attached to the component through its ID;
- the **gravity** parameter sets relative size proportions for the elements within the common parent container;
- the **select** property enables item selection in the tree;
- the **data** parameter defines the datasource for the component;

We use JSON in the sample, so the items of the tree_data dataset look like this:

```
[
  { id: "1", type: "folder", value: "Music", css:"folder_music",
data:[..child items..]},
    ...
]
```

The grid_data dataset includes data items in the next mode:

```
[
  { id:1, title:"The Shawshank Redemption", year:1994, votes:678790,
rating:9.2, rank:1},
    ...
```

```
]
```

- the **autoConfig** property creates columns of datatable on the basis of data properties. Headers of columns are based on property names in dataset.

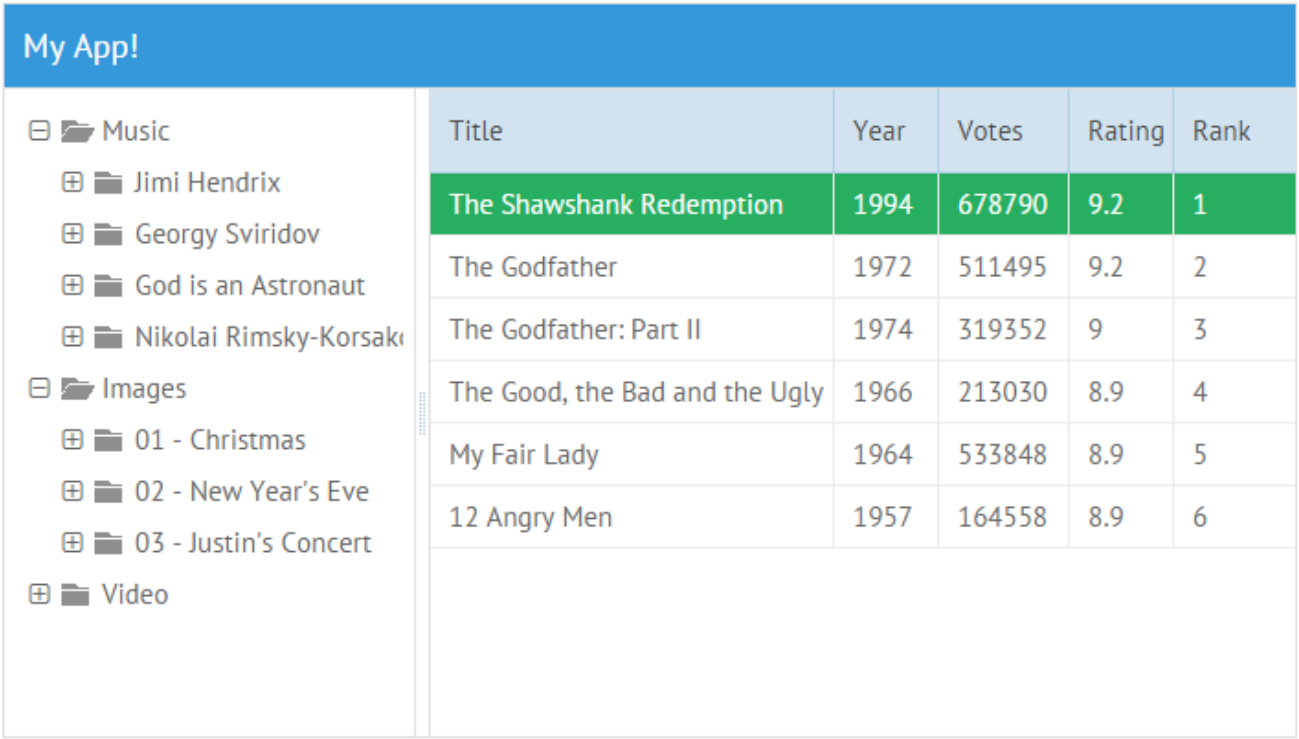
7 . At the last step we'll illustrate the usage of ids. In order to refer to a component the following structure should be used - **$$("id")**.
We will use the Tree method open to open the tree branches and the DataTable method select to select the 1st item of the datatable.

```
webix.ui({
  container:"app_here",
  rows:[
     { type:"header", template:"My App!" },
     { cols:[
      { view:"tree",id:"mytree", gravity:0.3, select:true, data:tree_data
},
      { view:"resizer" },
      { view:"datatable", id:"mydatatable", autoConfig:true,
data:grid_data }
    ]}
  ]
});

$$("mytree").open(1);
$$("mytree").open(2);

$$("mydatatable").select(1);
```

The ready app is presented in the following picture:



You can check the live demo and play with code by the next link http://webix.com/snippet/eadf0190

See also

- Interactive step-by-step tutorial
- Interactive DataTable tutorial
- Interactive Form tutorial

Beginner

Start Coding with Webix

Working with the Webix library you should go through typical development stages that include setting goals and understanding the requirements, choosing components and designing the app to foresee the result and finish it all with the coding itself, which is merely a technical side.

- Setting the goal
- Preparing Workspace
- Designing
- Main principles of coding with the webix
- HTML Containers
- Code Optimization

Setting the goal

At this stage your should determine the app's functionlity and evaluate the means of achieving this goal. You should clearly see the purpose of your future app and try to imagine all possible features it may need to meet the requirements.

Preparing Workspace

At this stage you define all the properties of your app and its components, play with event handlers and apply methods for your app to dance in the hands of users. You can check the state of your app in the realtime just as you've put the last bracket/brace/colon after the code piece.

The lib package you download from the site includes two versions - **debug** and **standard** one. Debug version helps control the app during the development stage while standard script version goes for ready-made products. Both scripts are identical in terms of component building yet **webix_debug.js** version includes a number of handy debugging tools. Read more about the bonuses of the debug version.

All the coding can be done in any text editor that is capable of syntax highlighting (just for convenience sake) and save the code to desired format (js, html, php, xml etc.).

At the initial stage, you need to include Webix library files into your document. It's recommended to use HTML5 doctype.
There are 3 possible ways:

1) Include Webix files from Webix CDN

Yoiu can directly set links to **webix.js** and **webix.css** files located in Webix CDN:

```
<!DOCTYPE HTML>
<html>
    <head>
    <link rel="stylesheet" href="http://cdn.webix.com/edge/webix.css"
type="text/css">
    <script src="http://cdn.webix.com/edge/webix.js"
type="text/javascript"></script>
    </head>
    <body>
        <script type="text/javascript" charset="utf-8">

 ... // here your app will "take shelter"
</script>
</body>
</html>
```

By the way, if you need to get one of the older Webix versions, you can include the necessary number of version into the links to the code files:

```
<link rel="stylesheet" href="http://cdn.webix.com/2.2/webix.css"
type="text/css">
<script src="http://cdn.webix.com/2.2/webix.js"
type="text/javascript"></script>
```

This variant is suitable only to the Webix version distributed under GPLv3.

2) Including Webix files via Package Managers

Webix lib can be installed via special dependency management tools. For now, Webix library is integrated with:

- **NuGet** package manager for the Microsoft development platform;
- **Bower** package manager for the web.

Installing Webix requires by means of these tools requires executing just a line of code inside them (you don't need to download anything):

```
NuGet

nuget install Webix

//If you use Microsoft Visual Studio, execute this from Package Manager
Console
install-package Webix
```

```
Bower

bower install webix
```

This variant is also suitable only to the Webix version distributed under GPLv3.

3) Including Webix Files from Downloaded Package

Compulsory Webix files from the downloaded package are **webix.js** and **webix.css**. They are included via standard script and link attributes. Be attentive to specify the right relative path to the files.

This method allows choosing between debug and production version.

```
<!DOCTYPE HTML>
<html>
    <head>
        <!-- Webix CSS file-->
        <link rel="stylesheet" href="../../codebase/webix.css"
type="text/css">
        <!-- Webix JS file-->
        <script src="../../codebase/webix.js"
type="text/javascript"></script>
    </head>
    <body>
        <script>
 ... // here your app will "take shelter"
</script>
</body>
</html>
```

Designing

Here you think about the app's layout, its look-and-feel, build color scheme and, of course, select all the components for it. You should already decide how the components would be related to each other. Building App Architecture article will help you position components and make you acquainted with layout types the library offers.

Further on, there're a number of data-presenting components and special features like toolbar, menu and calendar that would breathe life into your app and make it dynamically change to implement all the functions you wish it to perform.

The library is supplied with a rich set of samples to show you how the components look and perform in reality. So you won't regret having spent some time studying them.

So, there comes time for creativity, so put on your chinking cap and pass to coding.

Main principles of coding with the Webix

All the components inherit from the base **view** component and are embedded into it to appear in the output. So, the most frequent piece of code you are going to write is

```
webix.ui({
    view:"...",// specifies the component name you want to create
```

```
    id:"...",
    //other parameters
});
```

..or

```
var table = new webix.ui({ //any var name will do
    view:"...",
    id:"...",
    //other parameters
});
```

Working with Component ID

Why is **ID** so important? Because it's the element you can refer to in order to call component methods (including the one that attaches a handler to an inner event):

```
//attaches handler to inner event
$$("component_id").attachEvent("onSomethingHappens", do_something(){...});
//forces repainting
$$("component_id").refresh();
```

Methods can be call directly from the component object:

```
var table = new webix.ui({...});
//or
table.attachEvent("onSomethingHappens", do_something(){...});
table.refresh();
```

However, if you don't specify the ID, it will be automatically generated with the **"$component_name1"** pattern for each instance of a component.

```
Auto Generated ID

webix.ui({
    view:"list",
    //config options
});

$$("$list1").select(5); //$list1 is an auto-generated ID
```

RELATED SAMPLE: Auto-Generated ID

Still, referring to the components by specifying the auto-generated ID is not safe as it may change in case a new instance of one and the same component is added to the application.

Getting Component ID at Runtime

At any moment you can get the component ID (either custom or auto-generated one) through its configuration:

```
var id = table.config.id;
```

and use it for calling methods:

```
$$(id).refresh();
```

HTML Containers

You can set position, style and dimensions of the needed components with the help of javascript only (study Layout and Sizing Components articles) or place the into html containers to take the task to CSS. To achieve this, make up an html-layout, set ID to **div** elements and set this ID as the value of the component's container parameter.

```
<body>
    <div id="dataA" style="width:500px;height:150px;"></div>
    <div id="dataB" style="width:500px;height:300px;"></div>
    <script>
    webix.ui({
        view:"dataview",
        id:"mydataview1",
        container:"dataA",
        ...
    });

    webix.ui({
        view:"dataview",
        id:"mydataview2",
        container:"dataB",
        ...
    });
    </script>
</body>
```

Coding Tips

Components are removed from the application by simply wiping the code in the editor, but they can be as well destructed programmatically with the removal of all HTML elements and event for the component.

```
$$('mydataview1').destructor();
```

In addition, you can programmatically check whether the component with the specified ID was created and exists in the app:

```
if(!$$("my_dataview").exists())// -> returns boolean value
    var dataview = new webix.ui({
        view:"dataview",
```

```
        id:"my_dataview",
        ..config
    });
```

Code Optimization

Sometimes it is recommended to take some parts of an app to separate variables and include them into the **webix.ui()** command. It makes it easy to track mistakes and read the code later on.

```
var simple_config1 = ...;
var simple_config2 = ...;
var simple_config3 = ...;

webix.ui({
    rows:[
        simple_config1,
        simple_config2,
        simple_config3,
    ]
})
```

In case of a really big app with lots of components, controls and functions refer to the Structuring a Big App article to learn all nuances of code optimization.

Read more about HTML templates here.

For further information please study Common Tasks and Event Handling articles.

Debug Version of the Library

Debug version of the Webix library can be found in a **webix_debug.js** file in the **codebase** folder of a downloaded package.

Debug version is recommended for use during development stage for it has a number bonuses that helps you understand what's going on with your app and track bugs during development stage.

Main Features:

- Debug script version inspects function arguments and some component settings and informs about improper configuration like undefined variables, syntax mistakes. etc. instead of some incomprehensible internal errors;
- In case data loading goes wrong, these will be a corresponding message with the information about the error;
- Enables debugging flags usage;
- Enables debugging menu for each component.

Debugging Flags

- **webix.debug = true;** - enables event logging, which is the best way to see what events are there;
- **webix.debug_size = true;** - logs the info about current component sizes;
- **webix.debug_render = true;** - logs the info about how the component are rendered in the application;
- **webix_debug_proto = true;** - logs the info about the way components were created;
- **webix.debug_time = true;** - logs data loading and component rendering time.

Debugging Menu

Debugging menu is activated with a "**Ctrl + right mouse click**" combination on the needed component and offers a lot of debugging options:

- look up into the documentation for this component;
- learn the component's dimensions;
- study the component heirarchy on the page;
- long links to the key features of the component, namely the its JS and HTML object, configuration and data for the component.

Creating Basic Apps

- Setting Components
- Interaction Between App Elements
- Server-Side Integration

Beginner

Setting Components

Here we create a **simple filmset** - a list of films, where you can add elements.

Here you will need:

- a list with film data;
- a form to add new items to the set;
- a toolbar with 'add' and 'delete' to control the app.

First of all, you should create the app layout and then, place ui components and controls there.

Follow the guide step by step:

Step 1.Introduction

This is a clean page:

```
webix.ui({

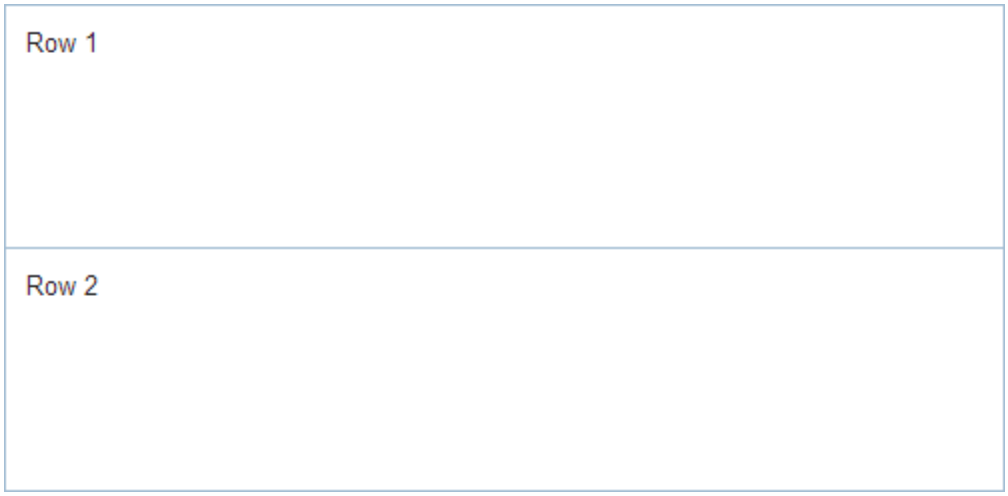
});
```

Step 2. Rows

Divide the page into **two rows** with the upper one being for toolbar. Set the width to the app - 500px.

```
webix.ui({
    width:500,
    rows: [
        { template:"Row 1" }, //1st row
        { template:"Row 2" }  //2nd row
    ]
});
```

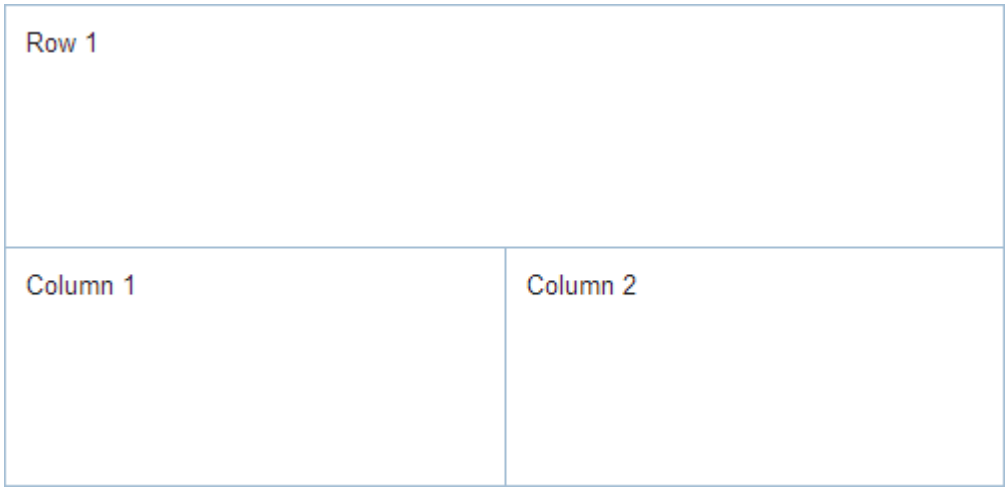
The layout doesn't show empty cells. That's why we temporary put templates into rows to force the layout to display the structure.



Step 3. Columns

Divide the lower row into **two columns** for list and form components.

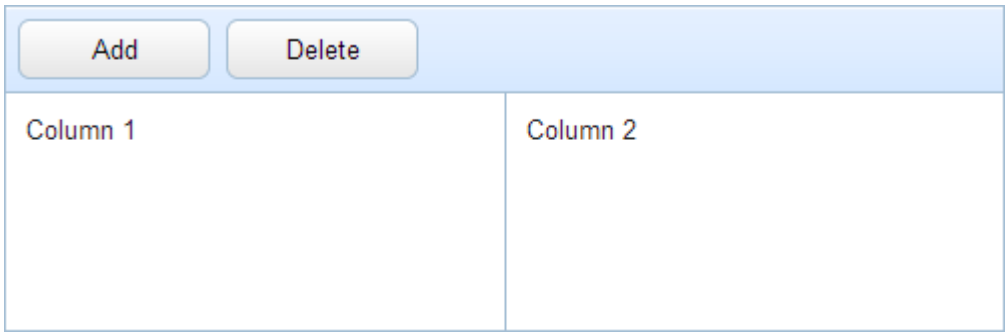
```
webix.ui({
    width:500,
    rows: [
        { template:"Row 1" },
        { cols:[
            { template:"Column 1" },//1st column
            { template:"Column 2" } //2nd column
        ]}
    ]
});
```



Step 4. Toolbar

- Place a toolbar with two buttons into the first row;
- Define text for the buttons with the help of their value property;
- Set the width of the buttons to 100 pixels (with the width property).

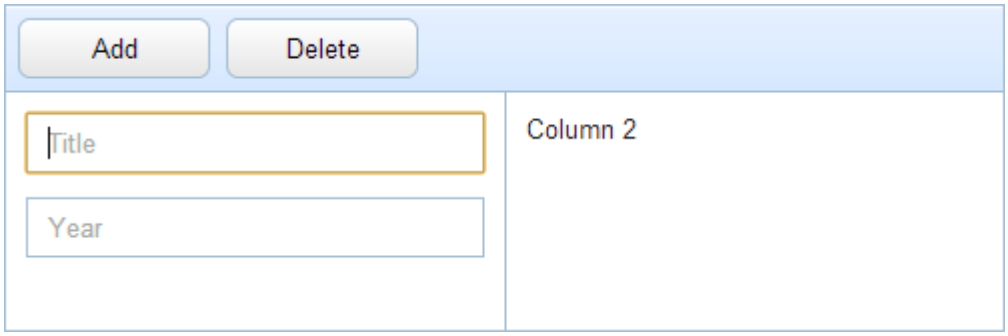
```
webix.ui({
    width:500,
    rows: [
        { view:"toolbar", elements:[
            { view:"button", value:"Add",    width:100 },
            { view:"button", value:"Delete", width:100 }
        ]},
        { cols:[
            { template:"Column 1" },//1st column
            { template:"Column 2" } //2nd column
        ]}
    ]
});
```



Step 5. Form

Place a form into the first column. The form will contain two text inputs stored in the elements array.

```
webix.ui({
    width:500,
    rows: [
        { view:"toolbar", elements:[
            { view:"button", value:"Add",    width:100},
            { view:"button", value:"Delete", width:100 }
        ]},
        { cols:[
            {view:"form", elements:[
                { view:"text", placeholder:"Title"},
                { view:"text", placeholder:"Year"}
            ]},
            { template:"Column 2" } //2nd column
        ]}
    ]
});
```



Step 6. List

- Place a list into the 2nd column;
- Use the data property to populate the list with data in JSON format (there can be other possibilities);
- Notice, how the **title** and **year** properties are reflected in the list template.

```
webix.ui({
    width:500,
    rows: [
        { view:"toolbar", elements:[
        { view:"button", value:"Add",    width:100},
        { view:"button", value:"Delete", width:100 }
        ]},
        { height:120, cols:[
            {view:"form", elements:[
            { view:"text", placeholder:"Title"},
            { view:"text", placeholder:"Year"}
                ]},
            {
                view:"list",
                template:"#title# - #year#", // which data to show
                select:true, //enables selection
                height:400,
                data: [
                    { id:1, title:"The Shawshank Redemption", year:1994},
                    { id:2, title:"The Godfather", year:1972},
                    { id:3, title:"The Godfather: Part II", year:1974},
                    { id:4, title:"The Good, the Bad and the Ugly",
year:1966},
                    { id:5, title:"My Fair Lady", year:1964},
                    { id:6, title:"12 Angry Men", year:1957}
                ]
            }
        ]}
    ]
});
```



RELATED SAMPLE: Creating Basic App. Step 1

Go further:

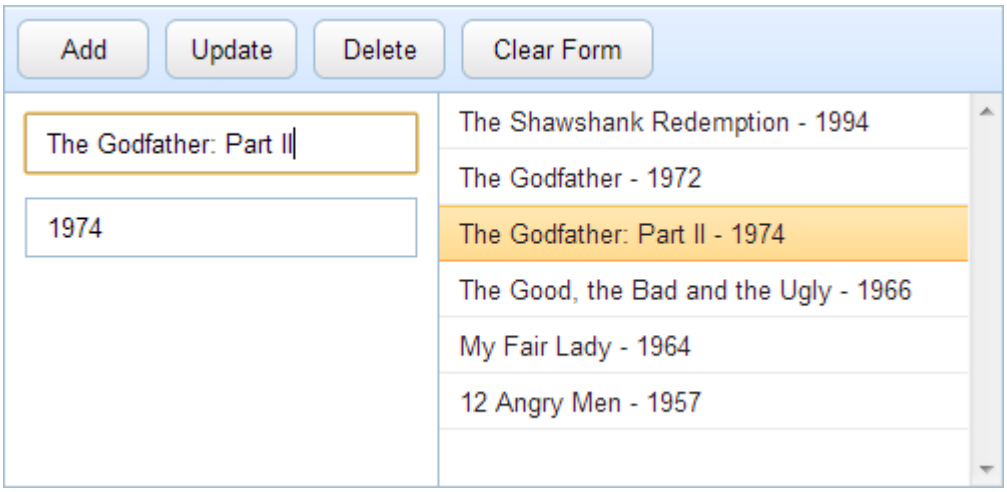
- Interaction between App Elements
- Server-Side Integration

Beginner

Interaction Between App Elements

As soon as you've learned how to create a basic app, let's go on and enable the app's functionality with the help of attached events and functions.

In the previous step we made a form and a list. Now we want to **add items** to the list by filling the form as well as to **delete items** from list.



At the same time, here we'll additionally create an **update** button on toolbar to update existing records in the list and a **clear form** button to delete values from the foorm before inputting new values.

Setting IDs, attaching event handlers

We need to attach functions to the buttons so that they can implement **adding**, **updating** and **deleting** operations stated by their values.

An event handler can be attached to the components in two ways:

- using native attachEvent() method;
- using components properties such as click and on.

All the native methods (including the one that attaches event handlers) are called from the component object that can be received using the following construct:

```
var obj = $$("component_id");
```

And then call any native method, e.g.:

```
$$("component_id").attachEvent(...);
$$("component_id").clear();
```

To attach functions to buttons, you need to specify the event that will trigger their execution anmd the

functions (function names). Here, dedicated functions are called by button **click**.

```
{ view:"button", value:"Add", width:70, click:"add_row"},
{ view:"button", value:"Delete", width:70, click:"delete_row"},
{ view:"button", value:"Update", width:70, click:"update_row"},
{ view:"button", value:"Clear Form", width:70,
click:"$$('myform').clear())"}
```

Then, let's set **IDs** to the needed components. We will work with **form** to **get values** from it as well as with **list** to **add** and **update records** there.

```
{view:"form", id:"myform", elements:[...]},
{view:"list", id:"mylist", template:"..."}
```

Apart from ID, components and controls have another identifier, **name.** It's a read-only component property. In case with form, we need element **names** to derive **form values**.

```
{ view:"text", name:"title", placeholder:"Title"},
{ view:"text", name:"year", placeholder:"Year"}
```

Now, let's describe the functions:

"Delete" button to delete a row

To delete a row from the list, we need to get the selected item and them pass it to the **remove();** function as parameter.

```
function delete_row(){
    var id = $$("mylist").getSelectedId(); //returns the ID of selected
item
    $$("mylist").remove(id);
}
```

"Add" button to add a new record

To add an item, we need to get text values from the form and assign them to list according to its template values.

Form values are derived with the **getValue()** function called from the form using its ID. To get to the necessary value, specify its **name**.

```
function add_row() {
    $$("mylist").add({
        title: $$("myform").getValues().title,
        year: $$("myform").getValues().year,
    })
```

```
}
```

"Update" button to update existing records

Item update is a **two-step** process.

- Firstly, you need to transmit data from selected list item to form fields;
- Secondly, you need to push the changed data into the list so that it replaces old values.

1 . Here we use **setValues();** method to set selected data as values for the form fields. The function fires on list item selection and takes this item ID as argument. List values are set to the fields with specified names:

```
$$("mylist").attachEvent("onAfterSelect", function(id){
    $$("myform").setValues({
        title: $$("mylist").getItem(id).title,
        year: $$("mylist").getItem(id).year
    });
});
```

2 . The core of a custom **update_row()** function is the **getValues()** method that gets values from form fields. Finally, the list is updated with the function of the same name - **update()**

```
function update_row() {
    var sel = $$("mylist").getSelectedId(); //checks whether the item is
selected
    if(!sel) return; // if not, function execution is stopped

    var value1 = $$('myform').getValues().title;
    var value2 = $$('myform').getValues().year;

    var item = $$("mylist").getItem(sel); //selected item object
    item.title = value1;
    item.year = value2;
    $$("mylist").updateItem(sel, item);
}
```

Confirmation window

To avoid accidental deletion of an important item, you need to safeguard user actions with a "confirm" window:

```
function delete_row() {
    var id = $$("mylist").getSelectedId();
    webix.confirm({
        title: "Delete",
        text: "Are you sure you want to delete the selected contact?",
        callback: function(result) {
```

```
            if (result) {
                $$("mylist").remove(id);
            }
        }
    });
}
```

Confirmation window is inited by the webix.confirm constructor. It inlcudes

- **title** (string) - text header of the window;
- **text** (string) - text message to the user;
- **callback** - the function that will be called on button click. The **result** parameter is *true* or *false* depending on your choice (yes or no). If result is positive, the **remove();** function is executed.

Read more about message boxes in the corresponding chapter.

RELATED SAMPLE: Creating Basic App. Step 2

Related articles:

- Event Handling
- Data Update
- HTMLform and Form Treatment

Go to the previous page - Creating a Basic App

Go further to Server-Side Integration

Intermediate

Server-Side Integration

On previous steps we've done good job.

- We've set the layout for the app and placed there UI components. See Basic All Tutorial. Step 1.
- We have enabled interaction between different components within the app. See Basic All Tutorial. Step 2.

Now it's time to learn how to work with data stored in the database and how should we treat **CRUD** (create, read, update, delete) operations so that they not only appear on the page, but also change data in the database.

The **tasks for serverside** are as follows:

- To populate our list with data from the database;
- To ensure that all the changes you make on the client side (within the component) should be reflacted in the database (items should be added, deleted and updated in the DB)

Serverside Integration is described in detail in the related article of documentation while here let's make the following steps:

Loading Data from the Database

Data loading is performed with the help of **ready-made dataConnector**, a script that retrieves data from the necessary database table and renders its data to the component. With library package, you get a wide choice of data connectors to serve the needs of serverside work. There exist .Net, PHP and Java versions.

The connector should be set as data **url** for a component.

```
{
  view: "list", id: "mylist",
  template: "#title# - #year#",
  select: true,
  url: "../common/connector.php",
  datatype: "xml"
}
```

DataConnector enables only **server->client interaction**, it's the component's datasource. Used alone, it doesn't push client-side changes back to the database, which is the responsibility of **DataProcessor**.

DataProcessor Initialization

Webix.DataProcessor is responsible for server-side integration and handles all the CRUD operations.

It should be initialized for the component you'd like to perform server work with:

```
var dp = new webix.DataProcessor({
  master: $$('mylist'),  // specifies the desired component
  url: "../common/connector.php"  // specifies the server API endpoint
});
```

Very often dataProcessor gets the same **url** that is specified as component datasource. At the same time, it can be your own piece of code if you want your own custom logic.

DataProcessor will automatically "translate" all your actions to the server: when you add a record to the list, the new record will appear in the database table, etc.

Component Binding

At last, let's make a step up and **simplify form update**. On previous step we used an "*onAfterSelect*" event to transmit list data into the form and a complicated "*update_row*" function to push edited data back to list. It was for the sake of learning.

At the same time, Webix library has powerful Binding API to enable data synchronization between components. **Bind()** function is used to make one component a datasource for another one on selection basis.

The **update();** function is no longer needed as well since from now on we apply **save();** method to the form to push changes back to the list and the database.

```
webix.ui({
  view: "toolbar",
  cols: [
    {...},
    {
      view: "button",
      id: "update",
      value: "Save",
      width: 100,
      click: "$$('myform').save()"
    }
  ]  // the new function for the button
});

$$("myform").bind($$("mylist")); //component binding
```

Addition and removal of items goes the same way as it was on previous step.

RELATED SAMPLE: Creating Basic App. Step 3

Related Articles:

- Data Binding
- Server-Side Integration. Dataprocessor

Beginner

Introduction to Events

Events are the core of any web application. Why? They help to interact with users and bring interactivity to the page. When the user makes some action on the page, an event occurs. The developer uses this event to detect the action and run the appropriate script. In one word, events is a way to detect user actions and define how to react to them (which functions to run).

Webix events model

Webix supports a big number of cross-browser events to treat its components.

To see the events provided for a certain component, flip to API Reference chapter and click on this component. On the opened page you see all available events.

How to process Webix events?

To process an event, i.e. run a special script when the event triggers on a certain page element, you need to attach event handler to this page element.

To attach an event handler to a Webix component, you should use the attachEvent method as in:

```
$$("componentId").attachEvent("onItemClick", function (id){
    //handler code
});
```

The method takes 2 parameters:

- **name** - (*string*) the event name
- **handler** - (*function*) the handler function

For different events, handler functions take different parameters. The information about event parameters you can find on the page devoted to this event in the API Reference chapter.

Once you call the method, each time the event fires for the calling component the code specified as the second parameter will be executed.

Usage example

Let's consider a simple example: you have a list that displays your favourite films. When you click on the film you want a message box to be shown with the details of this film: the title, director and running time.




```
var films_set = [
  {id:1, title:"The Shawshank Redemption",director:"Frank Darabont",
time:142},
  {id:2, title:"The Godfather",           director:"Francis Ford
Coppola",time:175},
  {id:3, title:"The Godfather: Part II",  director:"Francis Ford
Coppola",time:200},
  {id:4, title:"Pulp Fiction",            director:"Quentin Tarantino",
time:154}
];

var fList = webix.ui({
    view:"list",
    data:films_set,
    template:"#title#"
});

fList.attachEvent("onItemClick", function(id){
    var item = this.getItem(id);
    webix.message("<b>"+item.title+"</b></br>director: "+item.director+
    ", time:"+item.time);
})
```

Related Articles

For more information about Event Handling in Webix, please, study the corresponding article of the documentation.

Webix Docs Guide

To make a great app you need to learn different things but if you're new to Webix you might be wondering where to begin.

Some platforms (Java, .Net, PHP) get ready-to-use solutions for data loading, saving, exporting.

For understanding the basic concepts and getting the desired information in the most convenient way we offer you to follow this road map:

Compatibility - lets you find out whether your development tools satisfy the library requirements.

Getting Started - gives essential notes that will help you to create your first Webix-based application.

- Start Coding with Webix - says about fundamental principles and covers development process from planning to coding.
- Quick Start Guide - links to a short article that will teach you how to easily get a simple application up and running.
- Creating Basic Apps - introduces a detail tutorial that takes a bit more time but cover all nuts-and-bolts of the library

UI Components - describes available components: short overview, initialization, usage, related guides and API.

Guides - the main chapter of the documentation covering developer-oriented tasks in details.

- App architecture - answers general configuration questions: how to specify and size required component, set headers(footers), manipulate visibility, handle events etc.
- UI features - tells you how to fine-tune the appearance of your app and data in it with the help of css classes, templates, special helpers and modes.
- Data operations - covers techniques for performing general operations under data, such as sorting, filtering, copy-pasting and selection.
- Third Party Integration - says which third-party libraries and how can be integrated with Webix components.

Auxiliary Resources - covers advanced techniques implemented by means of helper components. Mostly intended for expert developers to customize the existent functionality.

API Reference - provides reference information you need for coding: all the available methods, parameters and events of the library.

Compatibility

Server-side compatibility

As Webix is a client-side library, it doesn't depend on server backend, so you can use it with any server side technology.
For some platforms (Java, .Net, PHP) - ready to use solution available for data loading, saving, exporting.

Client-side compatibility

Webix-based applications work on PCs and laptops using the following browsers:

- Internet Explorer 8+;
- Google Chrome;
- Apple Safari;
- Mozilla Firefox;
- Opera.

Webix is also intended for mobile use and fully supports the following devices:

- iPad;
- iPhone;
- iPod;
- Android-based devices.

As Webix is compatible with **PhoneGap**, you have access to mobile app stores, such as **Android Market** or **Apple App Store**, for distributing your applications.

What's new

- Version 2.4
- Version 2.3
- Version 2.2
- Version 2.1
- Version 2.0
- Version 1.10
- Version 1.9
- Version 1.8
- Version 1.7
- Version 1.6
- Version 1.5
- Version 1.4
- Version 1.3
- Version 1.2
- Version 1.1
- Version 1.0.2
- Version 1.0.1

Version 2.4

New features

- Portlet widget
- MultiCombo widget
- Range Slider widget
- Advanced validation messages for forms

Updates

- "button" config property
- "multicombo" control and onBeforeShow event
- "static" property for non-animated Multiviews
- event onBeforeEventShow for Scheduler
- getPopup method for combo editor
- ie9 support for progress bar
- invalid message and bottom label for form widget
- metaKey support (mac OS) for multiselection
- methods getCurrentFolder and getSelectedFile for FileManager
- noselect style for views in drag and multi-selection processes
- onPopup event for pivots (fires on popup creation)
- paging for TreeTable
- proxy is released after view destruction
- range slider
- read-only mode for pivot chart and table
- static mode for animation (multiview with iframes)
- webix ajax object to arguments of onBeforeAjax event
- addStyle reused previously created style tag
- headers as 5th argument of onBeforeAjax event
- updated FontAwesome to 4.3.0 version

Fixes

- ActiveContent doesn't survive component repainting in IE
- adding new items in filtered branches of treetable

- buttons inside html form
- clipbuffer styling in IE8
- close all sub-windows on master popup closing
- closing of nested popups ignores windows
- colspans and split mode
- column drag-n-drop on touch devices
- combo accepts full data object as a value
- combo behavior in dyn. loading mode
- correct input width for inputs with no label (richselect-based filters of datatable)
- creating new branches in filtered state
- CSV parser ignores the trailing new-lines
- detach options handler after master component destruction
- disabling headermenu for hidden column
- events on SVG elements
- FileManager upload and Uploader in IE8
- focus() and blur() for buttons
- hidden:false and modal:true in window config
- hot keys and numpad
- IE GET requests
- IE9 doesn't allow setting custom properties on AJAX object
- incorrect rendering of first row in case of paging
- invalid math in newly added rows
- material skin detection
- message width and height
- missed onAfterFilter event for server side filtering - server side filtering with activ...:40 PM
- multitext with suggest
- onBeforeFileAdd in IE9
- onChange event in combo
- regression in data saving handling
- regression in tooltip for icon
- regression in treetable filtering
- resize last column in right-split mode
- responsive attribute doesn't work for clean and line layouts
- richselect filter in IE8
- richselect: setting an empty value clears list selection
- scroll definition for datatable on Touch devices
- scroll position on last row in paging mode
- serialization of data by webix.ajax

- serialization of Date object
- tabFocus doesn't work for icon view
- text positioning in datatable
- unnecessary scroll blocking by datatable
- using select api in datatable with complex id
- validation in dataprocessor breaks insert operations
- validation in hidden form
- value binding and dataFeed
- value formatting
- web skin active button color

Version 2.3

Breaking changes

- Default parse format is changed for datepicker and calendar controls from "*%Y-%m-%d*" to "*%Y-%m-%d %H:%i*". Both of them can be altered within current locale.
- Sizing of layouts changed. In Webix 2.2 and earlier height and width on layout were not including padding and margins. Starting from Webix 2.3 height and width of layout is a full size of layout, including all margings and paddings.

Check migration guide for more details.

New features

- new material skin
- [pro] barcode widget
- [pro] organogram (organization chart) widget
- [pro] multiselect and datepicker filters to Pivot
- [pro] header sorting and value formatting in Pivot
- [pro] processing of equal file names in File Manager
- [pro] file upload for IE9 in File Manager
- onChange event for toolbar and form to capture user changes
- onValues event for toolbar and form
- getMenu method for menu to get current menu/submenu object
- parseTimeFormat is added as **"%H:%i"** by default to parse "time" strings
- badges and icons for list and menus (and views based on them)
- badges for buttons
- placeholder for richselect
- spacer element in menu view
- ability to provide submenu configuration in submenuConfig property of the topmost menu
- ability to trigger onReady in a synchronous way
- "Today" and "Clear" buttons are added to the date editor. They are hidden by default.
- editor object can receive full data item

Updates

- DataCollection has getCursor and setCursor methods by default
- second click on a button with popup property will close the popup
- autoConfig can be reused for datatable, e.g. after clearAll() call
- $touchCapture flag in component configuration for touch limited mode. Used for slider control.
- before sending to server side all dates are serialized
- error handlers of webix.remote calls will receive the text of response
- treestore has ability to serialize all data or filtered one
- scrollable "Month" view in Scheduler
- better default styling, ability to customize layout configuration

Fixes

- fix regression in data loader init
- correct values for onChange event in slider
- using TreeDataLoader instead of DataLoader
- unfiltering to affect not filtered view
- getActiveId() for empty carousel
- correct rendering of buttons after changing visibility state
- removeView() to remove related value from form values
- onChange event in form triggers only on edit operations in form
- readonly mode for datepicker and colorpicker
- onChange event for slider
- using template.escape with null or undefined values
- key navigation in empty combo
- incorrect adding of hidden views in accordion
- using quotas in button's text
- onChange handlers for text and select controls used as ActiveContent
- tooltip for inputs, progress in IE8
- x-scroll in tree
- tree horizontal scroll
- borders in top level (and borderless) space layout
- using removeView() to remove current view in the multiview
- special chars handling in text inputs
- progress bar positioning (scrolling offset for "icon" type only)
- top and bottom positioning
- layout uses fixed height and width without alterations
- recurring editStop() calls
- tree horizontal scroll

- slider fix for Touch in limited mode
- webix.ajax(url) call returns promise
- promiz will not occur if error was catched
- promiz throws an error only in debug version
- fonts for compact skin
- hiding columns with colspan
- clipboard breaks focus in the editor
- dynamic loading on touch devices
- scrolling to the last row index in case of scroll overflow
- IE8 fixes: preventing default actions and mouse event handling
- hiding columns with colspan
- regression in skin mapping
- borders in datatable when Bootstrap used
- clipboard breaks focus in the editor
- drag-n-drop for columns in the datatable, header with colspans
- dynamic loading on touch devices
- fixing promize functionality which can export self to external structure when global "module" is defined
- maximum tab number calculation
- TreeStore: each method and filtered branches
- applying new configuration after config reloading in datatable
- tabs rendering (border-box sizing)
- multiselect ignores non-existing values
- set optionHeight for skins
- checkAll() and uncheckAll() methods
- configuration of richselect with options from HTML
- inputHeight in case of labelPositon top
- tree ignores open call for already opened items
- regression in getValue() of colorpicker
- attribute checking in block-selection for datatable
- value property in tabview
- fixed datathrottle usage
- Date serialization during ajax operations
- resetting uploader after file choosing
- focusing checkbox editor in Crhome
- disabled counter in IE10
- options with quote in multi-select
- regression in getValue of colorpicker
- resetting text filter value after reloading data from the server side

- filters in treestore
- autoheight in templates
- tabbar "more" icon styling
- blockselection and custom scroll
- blockselecting and dbl-click events
- blockselection in scrolled state
- blockselection in the empty datatable
- getValue(), "editable" value
- setLocale() and arrays extend
- setValue() and getValue() of datepicker
- setLocale() extend for complex properties
- remove icons from Scheduler's calendars
- timepicker position
- FileManager locale: confirmation popups; collisions resolving without url handler
- conflicts between native navigation keys and navigation hotkeys
- server side filter reacts on navigation keys
- object id for gridsuggest
- solve request collisions (delete+rename/move/create)
- createFolder for fileManager
- default hotkeys for windows
- treestore serialize (filtering)
- provideAPI
- moving items in tree and tree table while in the filtered state
- missed methods in TreeCollection
- "format" of datepicker and filemanager fixes
- processing setState with empty filter value
- select filter in FF
- regression in click handling inside of multi-level popup
- incorrect filter processing during setState call

Version 2.2

Breaking changes

- width and height settings for all data components include size of scrollbar (so component will always take exactly the same size as was defined by width and height properties)
- ajax.header was deprecated in favor of ajax.headers

If you are updating from Webix 1.x - please check the Migration guide.

New features

- data binding for Tree, TreeTable and TreeCollection
- optional Today and Clear buttons in the Calendar
- ability to define file types for file uploader
- webix.ajax API can be used to fetch binary data (file downloading via AJAX)
- optional hover for rows in DataTable
- horizontal and vertical auto-scrolls during drag-n-drop for data components
- extra options for filtering in Tree and TreeTable
- improved scrolling on touch devices
- advanced tooltip options for DataTable
- all data components supports object notation for $css property
- onClick and other mouse handlers can use any CSS class, not only the first one
- view.show can be used without animation
- markup.init will return newly constructed UI
- columns size and order can be restored through the setState API
- removeHotKey accepts human-readable key names

Fixes

- addView for TabView doesn't support extended tab configuration
- assigning the same CSS class during layout reconstruction
- calendar is incompatible with Bootstrap
- calendar: cellHeight support
- changing value in combo after focus out
- click event handling in IE8
- CSS for collapsed accordion items in compact and flat skins

- custom and touch scroll in Tree
- custom id in scheme.$init for Tree and TreeTable
- datePickerFilter can't be combined with other filters
- double call for dynamic loading in the Tree
- dynamic loading in filtered state for Tree and TreeTable
- export to Excell and null values in the DataTable
- header menu and column hiding
- header sorting for clicks in rich-filters in the header
- headermenu for grid with auto-config and remote data loading
- hotkeys in IE8
- IE8 fixes and pair event handling (context menu)
- IE8 sorting fix (map method)
- IE8 XML parsing
- incorrect filter value preservation during DataTable repainting
- incorrect id in the onDataRequest event
- incorrect menu sizing
- incorrect scroll state for the hidden datatable
- incorrect scroll state processing for grid with non-aligned rows
- incorrect size setting from dtable.setState
- incorrect state of headermenu after menu reloading
- invalid handling of 0 as the option value
- invalid result of hasEvent call after detaching event
- iOS fix for full-screen mode; uploader fix (status processing)
- minutes containers and browser zooming in calendar
- multiselect and idSpace fix
- non-numeric arguments in math formulas in the datatable
- popup closing delay
- popup doesn't adjust after inner content resize
- preserving value of multi-select filter after DataTable refresh
- regression in addView for accordion
- regression in block selection in DataTable
- regression in context handlers
- regression in input disabling from config
- regression in isolate tabbar handling
- repainting after removing item in the Tree
- replacing hidden cell in multiview
- richSelectFilter and getFilter API
- scroll state in DataTable after hide|show operations

- selection in dataview with dynamic loading
- setState in DataTable will ignore columns with non-unique ids
- sorting ignores state set by the markSorting API
- special chars handling by select filter
- spinner progress position
- styling of sparse datatable
- template breaks on "->" chars
- text vertical align for inputs inside toolbar
- uncheck for checkboxes in indeterminate state
- uncheck of initially enabled radio button
- unnecessary error message in case of AJAX call canceling
- unwanted onclick reaction on buttons in IE
- uploader active area for macos
- x-scroll in autosized lists

Version 2.1

Major features

- Better compatibility with Bootstrap and jQuery
- Icon font contains 479 icons now (Font Awesome updated to 4.2)
- "strict" mode for Webix
- Datatable and treetable math can be extended with custom functions
- Uploader can work in Internet Explorer 8-9

If you are updating from Webix 1.x - please check the Migration to latest Webix version

Minor features

- ability to send extra data with file upload
- addView() and removeView() methods can be used to add and remove tabs in tabview
- find API for datatable and treetable
- addOption() method supports option object as argument
- addDropZone() method for uploader to allow file upload by drag-n-drop on html area
- webix.ui.animate to animate view creation
- tabview can have custom width for different cells
- tab close and tab/segment remove events
- separator for multiselect

Fixes

- dataprocessor do not use scheme serializer before data sending to the server side
- view.show works correctly for tabview
- richfilter styling
- styling for invalid radio button
- dblclick events in the list
- unfiltering multi-select filter
- colspans in case of many datatables per page
- validation and third-party elements in the form
- destroy component in the tab after tab closing
- non-existent hours in specific time zones (Brazil for example)
- incorrect sizing for adjust command in datatable

- scroll config option for datatable
- sorting datatable in filtered state
- using markSorting against hidden grid
- error in math results in empty cell
- edit events in liveEdit mode
- htmlbutton template typo
- segment button display on iOS
- treestore filtering
- regression in the carousel view
- sorting of hidden column
- adjustColumn against the hidden column
- multiple multiselects per page
- column d-n-d and rotated headers
- incorrect parsing of width in complex menu while loading config from xml
- export of data with special characters
- reloding data in menu with hidden items
- ungroup API inconsistence
- ie mobile animation
- mouse-wheel other custom scrolls
- pie chart map clearing
- invalid this reference in webix.require callback
- addView can't be used with accordion
- eachColumn works incorrect after column's reconfiguration

Version 2.0

Major features

- Promises API for all ajax operations
- Progress bars and overlays
- Improved keyboard navigation
- Extra locales added
- Package includes source map files
- [pro] PRO edition includes Pivot component
- [pro] Multiselect and multitext inputs
- [pro] Advanced editors for DataTable and Property views
- [pro] Datasuggest and Gridsuggest views for form controls and editing;
- [pro] Colspans and Rowspans in DataTable
- [pro] Column menu in DataTable
- [pro] Grouped columns in DataTable
- [pro] Advanced filters for DataTable
- [pro] Optional header rotating in DataTable
- [pro] Optional custom scrollbars
- [pro] Extra 300 locales

If you are updating from Webix 1.x - please check the Migration to latest Webix version

Minor features

- custom height for header lines in datatable
- dark style for carousel
- getState api for dataprocessor
- onViewMove and onViewMoveEnd events to movable
- optional close button for tabs
- password editor
- simple way to add icons to tabs
- storage.clear for all models
- if id for data object was not provided, value will be used as id
- onMenuItemClick can block menu hiding
- overlay works for all components
- parse can accept raw ajax response and webix components
- suggest for text editors

Fixes

- ability to define popup as raw json config
- active style for danger button
- zero border radius in Safari
- accordion labels
- active content
- adjust doesn't work for columns with string template
- adjustRowHeight in grid with dynamic loading
- bar chart border correction
- changing master id doesn't work for bind forms
- closing popup editors after editor closing
- custom suggest types in combo and richselect
- datatable lost sorting marker after column resize
- datepicker in grid and empty value
- icon sizes
- icon tabs vertical align
- IE8 compatibility
- incorrect header size after setColumnWidth call
- isBranch API fixed
- loading xml data into template
- popup closes after click on button in the list
- regression in suggest and edit configuration
- right split recalculation
- serializing state of hidden filters in datatable
- series in case of custom x-axis
- series in chart with custom scale
- sizing of carosel after adding|deleting extra views
- slashes in templates
- space mode ignores borderless setting of layout
- text wrapping in the rich select control
- textarea and label styles
- type $css property for unitlist
- yCount and context menu

Version 1.10

Full list of changes and fixes

New functionality

- onSubmit event for Form ("enter" listener for "text"-based controls inside form)
- target for menu links
- column batches in datatable

Updated functionality

- each and eachChild iterators accepts "all" flag for TreeStore
- "property" styles in "flat" and "touch" skins
- textarea styles
- unsync api for backbone integration

Fixes

- not valid HTML for textarea control
- focus in empty form
- removeHotKey removes all hotkey handlers except of one specific
- webix.ui usage for cell replacement
- dynamic inputs can corrupt form's data
- double rendering in window
- changing month in main view (case of last day selected)
- month changing in case of last days selection (like 31 March than go to Feb)
- autosuggest in combo breaks after form.clean
- backbone integration and empty collection
- indeterminate checkbox styling and some other tree styles
- cancel validation for hidden form fields
- pager styling
- pre-selecting combo and richselect options in case of custom template for list
- options mapping
- options mapping
- carousel sizing (carousel was completely rebuilt: view-based, layout is inside)

Version 1.9

Major new features

- Hotkey for inputs Check details.
- HTML links in menu Check details

Full list of changes and fixes

New functionality

- webix.onAjaxError event
- ability to set default calendar sizes in skin
- bind can be be used to add new item into collection
- dayTemplate property
- focus style for buttons in flat and compact skins
- getHeaderNode method
- hotkey for controls
- menu links
- refreshFilter api

Updated functionality

- calendar sizes in touch skin
- eachColumn for visible and all columns
- eachRow and data each can loop through hidden elements
- form.focus() will focus first focusable item in the form
- strToDate will not break when incoming parameter is already a Date

Fixes

- async data loading in IE
- better handling of server side sorting and server side filtering
- data updates for checkbox in grid
- dnd in treegrid on header
- event for master checkbox
- hidden column in datatable with colspans
- incorrect work of date, combo, richselect editors
- label height
- labels reset
- lint semicolon error fix
- menu link width

- next/prev toggle buttons
- option width support
- placeholder in datepicker
- regression in reloading layout content with webix.ui
- select editors in property
- showColumn against initially hidden columns
- sync webix.require without callback
- validation css for non-input controls
- webkit transitionEnd prefix (scroll was not disappeared in Chrome)
- standalone pager

Version 1.8

Responsive Layouts and Tabbar

- Layout view can be hidden or moved if there's not enough space for them on the screen. Check details.
- Tabbar tabs can be moved to a related popup if there're not enoght space for them on the screen. Check details.

Disabling dates in Calendar

Calendar dates can be disabled to prevent their selection. Check details.

Breaking changes

- webix.proxy.$callback was replaced with webix.ajax.$callback
- adjustHeaders deprecated

Full list of changes and fixes

New functionbality

- limited width for tabs and popup with additional tabs
- onClick handler accepts names from $scope
- server side filtering and sorting for datatable
- template@type = clean
- template@onClick handler
- predefined webixbutton css class
- getChildViews(), getBody() methods to scrollview
- "local" proxy for working with localstore
- hide/show of top level views
- tooltip option for inputs

Updated functionality

- popup in tabbar refactoring
- adjust property accepts "data" and "header" values, adjustHeaders deprecated
- show and hide refactoring
- min|max date refactoring
- direction top and bottom for animation in multiview
- attributes, placeholder and readonly works for all inputs
- datastore importData can work with primitive sources
- data@save = true, works the same as @save = @url

- cssFormat receives 4th parameter - column id

Fixes

- regression with popup rendering
- tabview operations in hidden state
- _refresh_scroll for boolean "scroll" config; padding for popup
- hiding popup on resize; better popup style
- popupTemplate customization, size calculation
- scroll for contextmenu or submenu
- closing of nested popups
- position of centered window in case of fullpage scroll
- sizing of horizontal accordion
- clipboard operations on macos
- scrolling in datatable on macos
- clipboard operations and dyn. grid
- pager initialized after master component doesn't work correctly
- datatable blocks global page scrolling
- regression in dataprocessor url handling
- drag column over split area
- obj.open can be any positive value to mark branch open
- webix.require doesn't trigger callback for already loaded files
- regression in suggest for plain text fields
- collections and options for editors
- double-rendering in case of fillspaces
- collections and options in datatable
- webix.extend and webix.EditAbility
- math operations with complex keys
- webixEvent doesn't allow to block event
- using Webix with requireJs
- initial toggle state must be defined by "value"
- updateItem overwrite target instead of updating
- processing proxy callback
- tab id can't contain a single quote
- IE8 and list autosizing
- empty cell in math formulas
- column.minWidth has lesser priority than grid.minColumnWidth
- calendar doesn't reset zoom level on hide
- touch scroll for datatable

- onChange event for counter view

Version 1.7

Breaking changes in API

datatable.locate method returns object with "row" and "column" properties, in previous version result object has "row" and "col" properties respectfully.

Default skin

Default skin changed to flat (you still can use the old skin by using skins/air.css)
Compact skin changed to flat theme as well (old one renamed to skins/aircompact.css)

Improvments in API

- text sorting mode for datatable
- autoheight property for "property" view
- ability to edit math formulas in the datatable
- "touch" mode for multiselect

Full list of changes and fixes

New functionality

- "text" sorting mode for datatable
- autoheight for "property" view
- editMath option for datatable
- keyPressTimeout option for the suggest
- multiselect="touch" for all data components
- new compact skin
- placeholder for richselect and datapicker
- touch-multiselect mode

Updated functionality

- pager styling
- placeholder for richselect and datepicker
- folder codebse/imgs to codebase/fonts
- datatable.locate returns different data structure
- new skin based samples
- tree and treetable can render hierarchy from db starting from any level

- webix.message.keyboard disables all text input for modal boxes

Fixes

- adjustHeaders in case of colspans or rowspans
- ajax error for destroyed views
- combo autocomplete after moving focus to the different control
- cssFormat adds garbage to css string
- custom checkbox and custom radio clicks in iframes
- customScroll in datatable and removeView
- datafeed works only with strings
- dataprocessor with connector as proxy can send empty request to a server side02:43 PM
- drag-n-drop in latest FireFox
- export of treetable data to pdf and excel
- filter values lost after column resize
- freezing when focus target can't be found
- getChildViews in accordion
- getItemNode in case of split
- isDirty status and named buttons
- layout resize fix (maxlength should be checked before its usage in object)
- minHeight and inner content auto-sizing
- multi-level form validation
- native event absent in onDrop handlers
- popup close delay is too big on desktop
- refreshColumns doesn't work correctly without first parameter
- regression in hierarchical data parsing from XML
- removeView for named field in form
- render calls for inputs in hidden views
- rendering of hidden select and combo
- rendering of layout with all cells hidden
- resizing in form nested in accordion
- richselect and combo in IE8
- suggest box render self in case of empty suggest list
- uploader demo
- Web Fonts bug in Google Chrome
- window positioning for windows with "move" property

Version 1.6

Breaking changes in DataProcessor

Parameters of onBeforeSync, onAfterSync was changed

onError event replaced with two new events - onBeforeSaveError and onAfterSaveError

Improved support for mobile devices

- win8 touch support
- drag-n-drop on touch devices

Improvments in Server side integrations

- full support of REST API
- data in components can be updated from server side
- client side code can't be broken by server side errors anymore
- ability to set custom headers for server side calls

UI components

- better memory cleaning after component destruction
- improved API for complex forms

Full list of changes and fixes

New functionality

- multiview.config.fitBiggest option
- win8 touch support
- drag-n-drop on touch devices

1.5.1

- columns.optins accepts url of data feed
- combo and richselect: setting value on list load
- datastore importData
- getDirtyValues and getCleanValues methods
- getFormView
- live edit functionality
- locate method for chart (required for dblclick event handling)

- math operations in treetable
- onBeforeAjax event
- onBeforeDateSelect/onAfterDateSelect events to calendar
- onColumnResize and onRowResize events
- relatedView and relatedAction attributes
- required and validateEvent attributes for form elements
- sort:"server" in datatable
- unbind and unsync, correct destruction of synced and binded elements

Updated functionality

- disabled tap iOS7 highlighting
- addCss and removeCss attempts to set styles directly in DOM without full repainting
- error handling by dataprocessor, better support of REST mode
- "move" command attempts to make changes directly in DOM without full repainting
- special handling for dnd in case of mobile webkit
- updating data in the client side component after saving
- webix.env.mouse contain names of pointer events for current devicePM
- xml data feeds for suggests

1.5.1

- all layout components can use events
- destructor correctly removes the related views
- function resolving can work with both "someMethod" and "someMethod()" notations
- getColumnConfig returns config for hidden columns
- setColumnWidth uses column id as parameter, not column index

Fixes

- branch deleting in dyn. loading mode webix-hub/tracker#84
- config options and html template during parsing from html markup
- data reordering in datatable in case of mobile webkit
- destructed views not removed from auto-resize list
- destructing suggest if a master control of this suggest is destructed
- detection of Win Phones
- disabled styling in case of labelPosition="top"
- focus in inline editor after component repaint
- inline editors and correct edit state in datatable
- layout resizing on touch devices
- min value for slider
- scrollview not adjust self after inner content change in case of touch

- styling of list during drag-n-drop
- styling of row during drag-n-drop
- tree editor position self incorrectly
- webix.DragControl on touch devices
- webix.html.pos in case of touch events

1.5.1

- $scope not defined inside of window
- adding borderless element to layout through addView results in the incorrect sizing
- addView and removeView do not updated elements collection for rich formsPM
- after deleting view from the form its value still accessible throug getValues
- camelCase parameters while init from HTML
- checkbox react on empty space clicks, #75
- clock icon
- custom scroll and disabled areas
- custom scroll doesn't react on addview command
- custom scroll in suggest boxes
- datatable blocks whole page mouse-wheel scroll
- delete dataprocessor on master view destruction
- dynamic loading for deep sub levels
- elementsConfig doesn't work for nested fieldset
- empty text in combo input;
- fieldset breaks parentView chain
- form with scroll ignores height
- getChildViews for fieldset
- getFormView throws an error for element outside of form
- illegal comma removed
- invalid behavior for movable window and position:center
- invalid editor sizes after column or row resizing
- isDirty doesn't work for the fresh form
- isolate property and multiview, #79
- popup zIndex was not increased on show
- regression in popup editors
- regression in richselect related to after-loading value
- regression in xml-xhtml data parsing
- submenu that is not based on menu
- unnecessary custom scroll in datatable
- window sizing ignores min-width for the inner content

Version 1.5

Improved support for mobile devices

- new skin for mobile UI (demo)
- functionality of desktop UI adapted to touch events and gestures

Improvments in DataTable

- adjustRowHeight method added to datatable (autodetects and applies the correct row height)
- fillspace can be used for multiple columns (pecent based sizing in datatable)
- richselect can be used as editor in the datatable
- checkboxes can have checkValue and uncheckValue options

Improvments in Angular and Backbone

- [angular] webix-ui is compatible with ng-repeat
- [angular] webix-data works for options in combo and select
- [angular] init through angular directive links event handlers to the current scope
- [backbone] handling of reset event
- [backbone] handling of models with getters
- [backbone] WebixView is compatible with backbone 1.1
- [backbone] using sync with already loaded collection

Full list of changes and fixes

New functionality

- ability to define form as editor
- activeTitle option for the tree (open/close an item on click)
- adjustRowHeight method
- checkValue and uncheckValue for checkbox
- checkValue and uncheckValue for checkbox and radio editors in the grid
- className correctly processed while parsing data from the html
- getItemNode method
- global hotkeys for edit operations and window closing
- htmlbutton button type
- minWidth setting for grid's column
- onValueSuggest event for the suggest control
- retina support for canvas

- richselect editor for the grid
- webix.once method

Updated functionality

- $formElement deprecated
- a bit better visual position for showItem
- ability to toggle accordion items in any modes
- backbone integration - reset event and getter usage
- button and inputs styling for font-awesome
- cssFormat can be a string with name of function
- editCell in datatable can be called without column id
- fillspace can be used on multiple columns with different weights
- function names can be resolved in custom scope
- getSelection -> getSelectedId
- idColumn to csv abd jsarray data drivers
- init through angular directive links method to the current scope
- look of tabbar in clouds skin
- selectFilter in datatable sorts options by default
- suggest can use complex popups
- support for multiline rows in datatable
- tab navigation works for grid without active editor
- webix-data can be used to define the select options
- webixEvent and webixData works for object init (angular)
- window closing and modality

Fixes

- slider for touch
- autoscroll and autoopen during dnd
- autosizing of templates
- backbone view must return this
- calling ungroup on clearAll
- carousel resize after changing structure
- combo editor do not work with config.options
- datatable selection (multiselect was always applyed)
- dblclick doesn't work in cell selection mode
- debug file throws an error while generation error message for empty data set
- deleting selected row in treetable may cause auto-loading attempt
- disabled buttons in IE8
- form with scroll still uses maxHeight

- fullScreen for latest devices
- getItemNode for datatable
- getSelectedId doesn't work when plain id requested
- grouplist width
- icon font on old mobile devices
- icons in Touch skin
- incorrect $scope for child views
- incorrect processing text nodes in xml
- incorrect rendering of touch scroll for datatable in Chrome Mobile
- invalid uid generation code
- item styling for the the selected state
- menu vertical align
- named types for treetable
- named types in datatable
- ng-repeat and webix json integration
- onChange event in text inputs
- onRotate was called on window resize
- options collections and backbone 1.1
- overflow for label
- parsing long text content from xml in FF
- regression in angular init
- regression in getSelectedId
- regression in webixElements (angular)
- regression in window position
- resizing of templates and onload handler in iframe
- richselect icon
- selection mode affects the id to string conversion
- solution for 4K text node Firefox limitation
- styling of disabled elements in IE
- sync ignores configuration parameter
- sync with backbone ignores already loaded data
- tabbar do not self adjust after active tab change
- text-based validation rules not recognized
- using calendar suggest for text inputs
- using js code literals in xml config
- view throws an error if angular provides non collection as a data
- webix_security flag not set correctly for the connector calls

Version 1.4

Advanced data selection controls

- mutli-column select box RELATED SAMPLE: Grid Suggest
- rich content select box RELATED SAMPLE: Data Suggest

New Server Integrations

- loading and saving data through websockets (details)
- loading and saving data through indexedDB (details)

Improvements in Window Positioning

- window can have complex relation size and position (details)
- window can be shown in fullscreen mode

Other

- autoConfig option for the datatable
- dataprocessor tracks data moving events
- keyboard navigation for list component
- correct sizing of layout with hidden pannels
- elementsConfig supported for nested collections
- getSelection deprecated in favor of getSelectedId
- better styling for icon buttons
- webix.onReady event
- webix.ui.zindexBase property added
- different small fixes in UX and styling

Version 1.3

New skins

- 6 new skins added (details)

Improvments in UI

- video player component (ui.video) added (details)
- API and look of carousel control improved (details)
- charts can use logarithmic scale (example)

Other

- small fixes in form controls and datatable

Version 1.2

Integration with AngularJS

- webix-ui directive to define webix views directly in HTML
- webix-show, webix-event, webix-data directives to link webix components and scope
- webix component can be used with angular data bindings
- check details

Integration with Backbone

- webix components can load data from Backbone Collections
- webix components can save data back to Backbone Collections
- WebixView, that can be used as normal Backbone View
- Backbone Router can be used to alter Webix Layouts
- check details

Others

- setContent method for template component
- isolate configuration property for layout components
- onBeforeDropOut event added
- tooltip functionality added to the DataTable, TreeTable and Tree
- more than 50 different fixes

Version 1.1

Server side integration

- all components can save data through REST API
- offline and caching loading strategies
- custom data saving and data loading transports

UI improvments

- "disabled" configuration options for all views (including all form controls)
- webix.history can be used with multiview control
- per-submenu configuration is possible ("config" property of menu item)
- improved visual filtering in treetable and tree

API

- onViewResize event added
- "disabled" option added for all views
- ability to define XSS safe templates

Fixes

- popup's visibility on iOS
- incorrect sizing of multiview and accordion
- incorrect behavior of drag-n-drop in datatable
- setValue doesn't work for radio buttons

Version 1.0.2

New functionality

- getText method for datatable
- lineMaxLimit parameter that cuts a line in "non-item" position
- ui.fullScreen solution for FF
- default size of resizer changed
- xml parse can recognize arrays
- addView adds to the end of layout if index was not defined
- skin updates, important flags removed where possible
- csrf key now sent through http headers

Fixes

- IE ignores hotkeys
- IE8 doesn't generated dblClick events
- IE8 sets invalid value after changing cell value with select editor
- minWidth and maxWidth settings from xml
- loading tree-like data from XML
- datatable do not allow to define order of columns during export to pdf and excel
- incorrect remove action for local and session storage
- regression in layout rendering when views are added through addView
- dataprocessor and id change during binding
- label align in segmented button
- incorrect in-layout positions after showBatch call
- invalid animations in FF and Chrome

Version 1.0.1

New functionality

- layouts are correctly shown when they are zoomed by a browser
- selectFilter can show data from the attached collections
- better strategy for x-layout rendering
- API calls against hidden items in menu
- init from html|xml markup improved
- dataprocessor can have different urls for different action
- value attribute for tabview
- getTopParentView method added for all views
- getPopup method added to the datepicker
- setHTML method added to the label
- setValue and getValue methods added to the multiview

Fixes

- invalid size and focus of popup editors
- toggle button ignores inputWidth settings
- regression in treetable checkbox behavior
- regression in datatable markup parsing
- conflict between data and content properties of template
- row markers are not removed during clearAll
- mulitiview with no animation
- chart rendering in multiview (no animation case)
- label position in pie chart
- validation and htmlform control
- incorrect column autosizing in case of hidden container
- native selection during cell resize
- hideItem throws an error for menu's item which was already hidden
- incorrect handling of custom popups in editors
- sizing of scrollview was broken
- window ignores y parameter of show command
- dnd in tree as child

UI Components

The Webix UI library supplies over 40 fully customizable components to help you build interfaces of different kinds, nicely present data and work with it. There are also multiple controls and specific features to make your applications good-looking and user-friendly. All the components are objects made with the webix.ui() constructor where the type of the component is set by the 'view' property (the declarative syntax is available as well).

Layouts

Layouts help you to organize your web page, define its overall look-and-feel. You can arrange the content in columns and rows, divide it into groups and control through navigation or animation.


Accordion


Carousel


Form


Layout


Portlet


Multiview


Scrollview

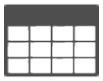

Tabview


Toolbar

Data Components

Data components are used to present data of a specified type on the page. Such components manage the space allocated for their data and provide functionality for its access and configuration.


Chart


DataTable


Dataview


Grouplist


List


Organogram


PropertySheet


Tree


TreeTable


Unitlist

Context Components

Context components let displaying spatial and temporal information on some user's action, such as click on a button or filling in a field.



HTML-oriented Components

Html-oriented components supply handy means for using habitual HTML elements in an application. There you can specify HTML templates, inline frames, forms etc.



Auxiliary components

Auxiliary components help you to complete the picture of your application, to add special features that can't be achieved with the standard set of components presented above.



Controls

Controls are mainly responsible for handling events since they enable user-app interaction by triggering functions attached to them. Note that any event can be attached to any component by referring to its ID.



 Counter

 Radio

 Text

 Datepicker

 Richselect

 Textarea

 Icon

 Search

 Toggle

 Fieldset

 Select

UI Extensions

UI extensions are helpful components that are mostly task-oriented and intended to accomplish a certain purpose. They are already quite configurable and need only initializing on the page to start serving.

To see a complete list of extension components and find out details of their use, please, go to the chapter - UI Extensions.

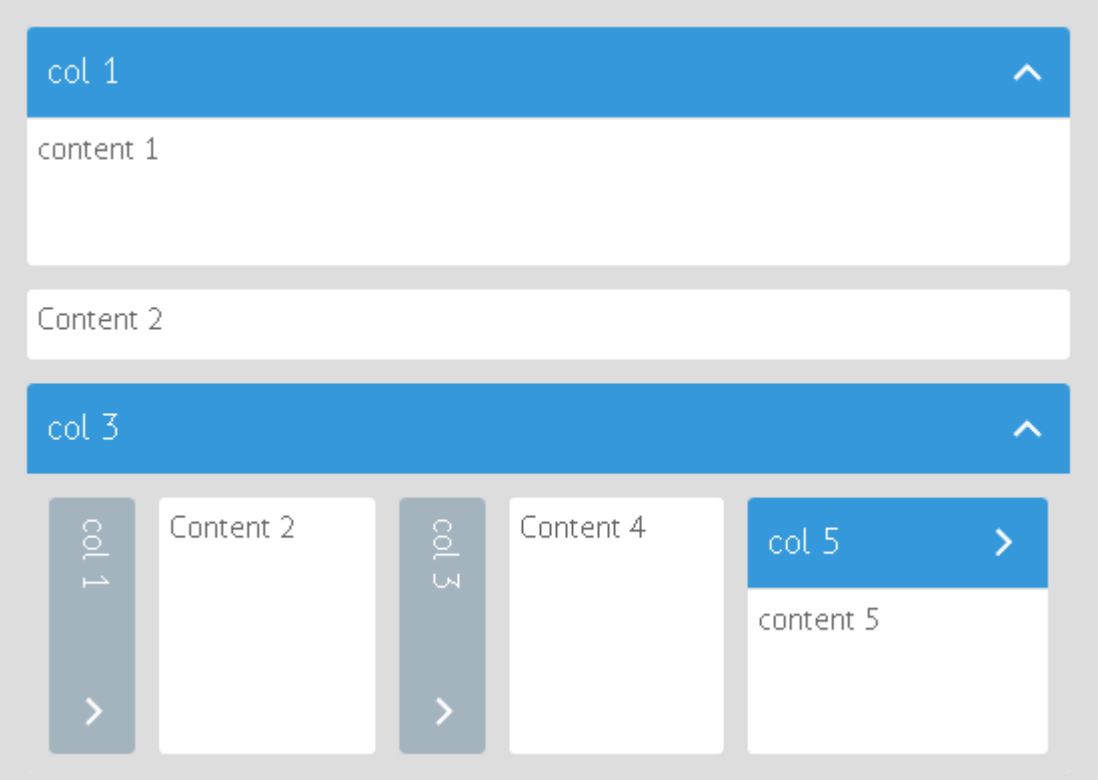
Accordion

API Reference

- Methods, properties and events
- Samples

Overview

Accordion consists of panes arranged into layout rows and cols. Each pane, or **accordion item**, features **body** and **header** and can be fully expanded and collapsed by clicking its header. You can place text or any component into each pane. The component inherits from layout.



Initialization

```
Accordion with three panes

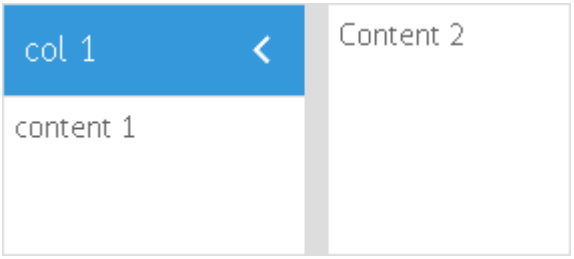
webix.ui({
    view:"accordion",
    multi:true,
    cols:[ //or rows
```

```
        { header:"col 1", body:"content 1" },
        { header:"col 2", body:"content 2" },
        { header:"col 3", body:"content 3" }
    ]
});
```

RELATED SAMPLE: Complex Accordion

Comments:

- By essence, each row/column is a ui-related accordionitem component; however, it's direct initialization isn't compulsory - you can just enumerate its **properties** in the row (column) as shown above.
- Moreover, you can omit the *view:*"*accordion*" line as well. The component will be drawn anyway as soon as you divide a row/column into **header** and **body** parts.
- There can be as well rows/ cols without headers. It goes without saying that they are non-collapsable.



RELATED SAMPLE: Horizontal Accordion

Note that **view name can be omitted** (view:"accordion"). If you place an array of objects with **body** and **header** into row or column - the view will be automatically recognized as accordion.

```
webix.ui({
    cols:[ //or rows
        { header:"col 1", body:"content 1" },
        { header:"col 2", body:"content 2" }
    ]
});
```

RELATED SAMPLE: Layout with Headers

Working with Accordion

Accordions differ by their possibility to show and hide panes at the same time. There's a special **multi** parameter to set the necessary mode.

- **multi:true (default)** - all panels can be expanded as well as collapsed at a time;

- **multi:false** - only one panel can be (and must be) visible at a time;
- **multi:"mixed"** - all panels can be expanded at a time, but you cannot close them all. One panel remains visible.

Related Articles

- Setting Borders to the Components
- Adding Headers to Components
- Sizing Components
- Resizing Components
- Layout
- Multiview
- Scrollview
- Carousel
- Accordion CSS Image Map

Accordionitem

Accordionitem inherits from view and is an integral part of the ui-related accordion component. You need at least two accordionitems to make a workable accordion.

Any accordionitem must have two properties: **header** for heading and **body** for main text or some ui component.

```
webix.ui({
     rows:[ // or cols
        {
            view:"accordionitem",
            header:"Pane 1",
            headerHeight:20,
            headerAlt:"Pane 2 Closed",
            body:{ view:"form", ...} //ui component
        },
        {
            view:"accordionitem",
            header:"Pane ",
            headerAlt:"Pane 2 Closed",
            body:"This is Pane 2 body", //just text
            collapsed: true
        }]
})
```

Comments:

- **header** - defines text heading for a panel in an expanded state;
- **headerAlt** - defines text heading for a panel in a collapsed state;
- **headerHeight / headerAltHeight** - defines the height of a panel in an expanded/collapsed state;
- **collapsed** (false or true) - defines the panel that will be expanded/collapsed on the app's loading.

If you initialize **accordion** like below, you can omit accordioitem initilization:

```
webix.ui({
    view:"accordion",
    rows:[
        {header:"Pane", body:"Pane 1"},
        {...}
       });
```

RELATED SAMPLE: Accordionitem

Accordionitems can be collapsed and expanded prograsmmatically with the help of dedicated methods:

```
$$('item1').collapse();
$$('item1').expand();
```

```
//don't forget to specidy IDs to accordionitems!
```


If the items of your accordion are arranged in cols, their headers are collapsed to vertical panels as you close an accordionitem.

Available only in **PRO Edition**

Barcode

API Reference

- Methods, properties and events
- Samples

Overview

The Barcode widget represents text values in a machine-readable format by varying the widths and spacing of parallel lines. The widget uses Canvas for lines rendering.



Basic Initialization

To begin with, create a new HTML file and include the related Webix code files in it.

Required code files are:

- webix.css
- webix.js

```
A basic html page with the included code files

<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Barcode</title>
    <script src="../codebase/webix.js" type="text/javascript"
charset="utf-8"></script>
    <link rel="stylesheet" href="../codebase/webix.css" type="text/css"
charset="utf-8">
```

```
</head>
<body>
    <script>
        //here you will place your JavaScript code
</script>
</body>
</html>
```

Next, you should create a div container to place your barcode into.

```
<div id="box" style="width:300px;height:200px;"></div>
```

Now you can initialize the component. To create a new Barcode object, you need to use the next constructor:

```
Barcode constructor

barcode = new webix.ui({view:"barcode"});
```

If you want to place the component into the container, you should specify the container property inside of the view constructor:

```
Basic barcode configuration

barcode = new webix.ui({
    container:"box",
    view:"barcode"
});
```

RELATED SAMPLE: Initialization

Configuration settings

You can set the following properties to configure Barcode:

- **type** - sets the barcode type. There are 3 built-in types: "ean8", "ean13" and "upcA" ("ean13" is the default one):

```
webix.ui({
    view:"barcode",
    type: "upcA"
});

//or
var barcode = webix.ui({
    view:"barcode"
});
barcode.define("type", "upcA");
```

- **value** - sets barcode value (text):

```
webix.ui({
    view:"barcode",
    value: "123456789012"
});
```

- **width** - defines the view width

```
webix.ui({
    view:"barcode",
    width: 250
});
```

If you want to set flexible width, you need to set the **-1** value for this property:

```
webix.ui({
    view:"barcode",
    width: -1
});
```

- **height** - defines the view height

```
webix.ui({
    view:"barcode",
    height: 180
});
```

If you want to set flexible height, you need to set the **-1** value for this property:

```
webix.ui({
    view:"barcode",
    height: -1
});
```

- **paddingX** - sets the left and right paddings (space between widget border and bars)

```
webix.ui({
    view:"barcode",
    paddingX: 20
});
```

- **paddingY** - sets the top and bottom paddings

```
webix.ui({
    view:"barcode",
    paddingY: 10
```

```
});
```

- **textHeight** - sets the height of the text block

```
webix.ui({
    view:"barcode",
    textHeight: 20
});
```

- **color** - sets the bar color

```
webix.ui({
    view:"barcode",
    color: "#000"
});
```

Setting Value

You can set the value in two ways:

via the **value** property in Barcode configuration

using the **setValue** method

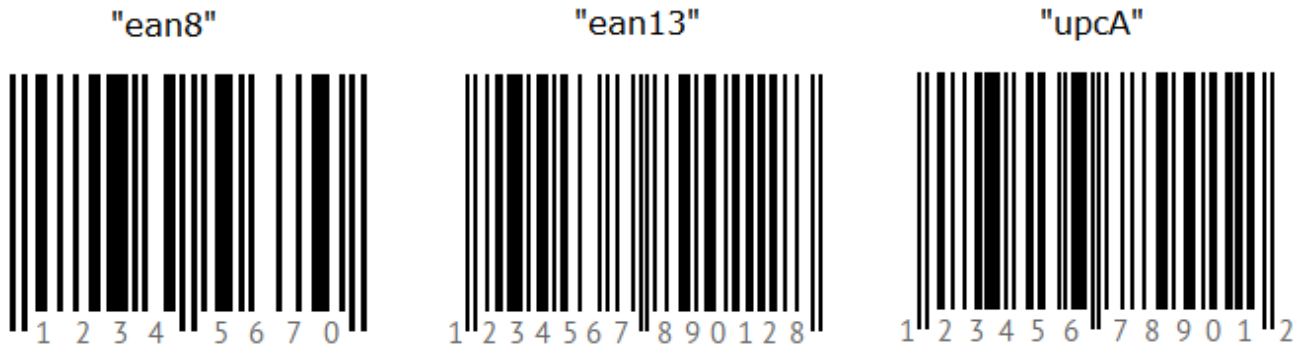
```
var barcode = new webix.ui({
   view:"barcode",
   value: "123456789012"
});
// set the new value
barcode.setValue("453456789015");
```

Type Definition

Webix Barcode includes 3 definitions for most popular barcodes (EAN8, EAN13 and UPC-A):

- ean8
- ean13
- upcA

"ean13" is the default type.



Use the **type** property to set the Barcode type:

```
barcode = webix.ui({
    view:"barcode",
    type: "upcA"
});
```

The type of Barcode can be changed dynamically via the **define** method. After the changes are applied, you need to call the **render** method to redraw Barcode with the new type:

```
barcode.define("type","ean8");
barcode.render();
```

RELATED SAMPLE: Built-in types

Custom Types

If you need create a custom barcode, you can add the necessary type to the existing Barcode view.

Each barcode type should contain a function that returns a sequence of “0” and “1” chars based on the incoming text value. This 0-and-1 character combination will be used for bar rendering.

The Library will render black bars for “1” chars and leave gaps for the “0” ones.

To add a new type you need to use the **webix.type** method that takes two parameters:

- ui object that the type will be created for (webix.ui.barcode for Barcode)
- object with type configuration

```
webix.type( webix.ui.barcode, {
    /* type configuration here */
});
```

Type configuration of Barcode must contain 3 properties:

- name - {string} the name of the code
- encode - {function} the function that returns sequence of “1”and “0” chars for black-and-white bar rendering

- template - {function} the function that returns text for barcode

Let's consider an example of the "code39" barcode configuration:

1) define type name in the type configuration:

```
 name:"code39",
```

2) set an array with encodings for 43 characters, consisting of uppercase letters (A through Z), numeric digits (0 through 9) and a number of special characters (-,., $, /, +, %, and space):

```
encodings:[
    "110101001011",
    "101101001011",
    ...
]
```

3) set the **template** function that will convert the incoming text to uppercase and add “*” as start and stop characters:

```
 template: function(value){
      return ("*" + value + "*").toUpperCase();
 }
```

4) define the **encode** method that will apply 0-1 sequences from *encodings* array to incoming text value

```
encode: function(value){
    var chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-. $/+%*";
    var code =””;

    // apply template method
    value = this.template(value);

    for(var i=; i<value.length; i++){
        var pos = chars.indexOf( value.charAt(i) );
        if (pos < )
            return "";
        if (i)
            code += ““;

        // apply encoding for character
        code += this.encodings[pos];
    }
    return code;
}
```

So, the whole "code39" type definition will be as in:

```
webix.type(webix.ui.barcode, {
    // name
    name: "code39",

    // encodings for "A","B", etc. characters
    encodings: [
        "110101001011",
        "101101001011",
            ...
    ],

    // applies encoding for incoming text
    encode: function(value){
        var chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-. $/+%*";
        var code =””;

        // apply template method
        value = this.template(value);

        for(var i=; i<value.length; i++){
            var pos = chars.indexOf( value.charAt(i) );
            if (pos < )
                return "";
            if (i)
                code += “”;

            // apply encoding for character
            code += this.encodings[pos];
        }
        return code;
    },

    // returns text for code39 barcode
    template: function(value){
        return ("*" + value + "*").toUpperCase();
    }
 });
```

Here is an example of the "code39" usage:

```
barcode = webix.ui({
   view: "barcode",
   type: "code39",
   value: "CODE 39"
});
```

The result of applying the "code39" type is given in the picture below:



RELATED SAMPLE: Custom types

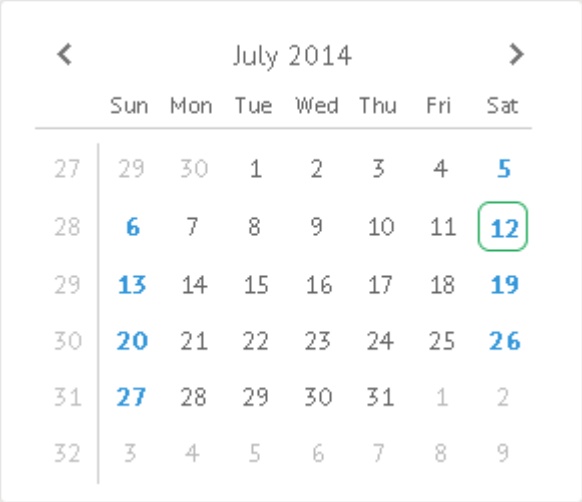
Calendar

API Reference

- Methods, properties and events
- Samples

Overview

UI-related calendar inherits from view and allows you to include the Date and Time picker calendar into an app. It is a handy and fully interactive tool to pick the necessary time and date when filling a form. The dedicated timepicker allows setting the necessary time that will be displayed below the calendar area.



Initialization

```
webix.ui({
    view:"calendar",
    id:"my_calendar",
    date:new Date(2012,3,16),
    weekHeader:true,
    events:webix.Date.isHoliday,
    calendarDateFormat: "%Y-%m-%d",
    width:300,
    height:250
});
```

RELATED SAMPLE: Calendar

Comments:

Here is a calendar with basic parameters.

- **weekHeader** (boolean) - if *true* shows weekday above;
- **events** (CSS class or specific cases) - e.g. **webix.Date.isHoliday** highlights dayoffs;
- **date** (string, object) - specifies the start date in the "year, month, day" format with months starting from 0. In case of an empty new Date(); you'll be shown current date;
- **dayTemplate** (function) - specifies a function that sets a template for each day cell. By default the date is displayed without any styling;
- **icons** (boolean) - specifies whether to show "Today" and "Clear" icons.

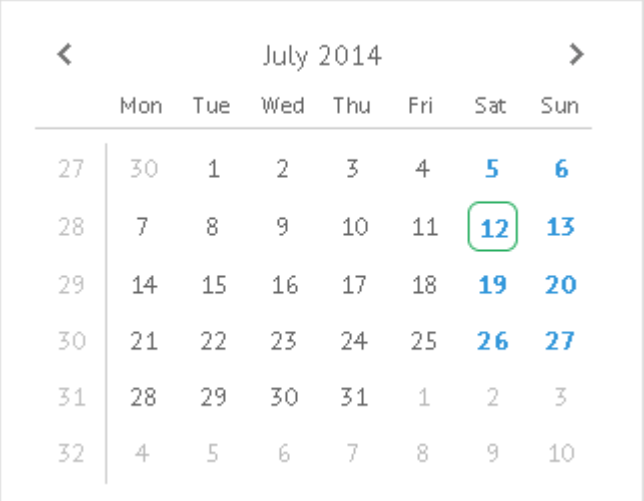
To see the full list of Calendar properties please refer to the corresponding chapter of the documentation.

Start Date

By default the calendar starts a week on Sunday. However, you can also set Monday as a start day.

```
webix.Date.startOnMonday = true;

webix.ui({
    view:"calendar", ...
});
```



RELATED SAMPLE: Calendar: Setting Start Date

Working with Calendar

Setting and Getting Dates

Setting dates means their selection. Dates can be set programmatically after calendar initialization:

- **setValue**(new Date(2012,3,30)); - common method for setting values to the controls;
- **selectDate**(new Date(2012,3,30)); - calendar specific method equal to the above mentioned one;

- **showCalendar**(new Date(2012,3,30)); - makes the calendar **show** the specified date, but doesn't select it.

Getting dates can be done with two corresponding methods:

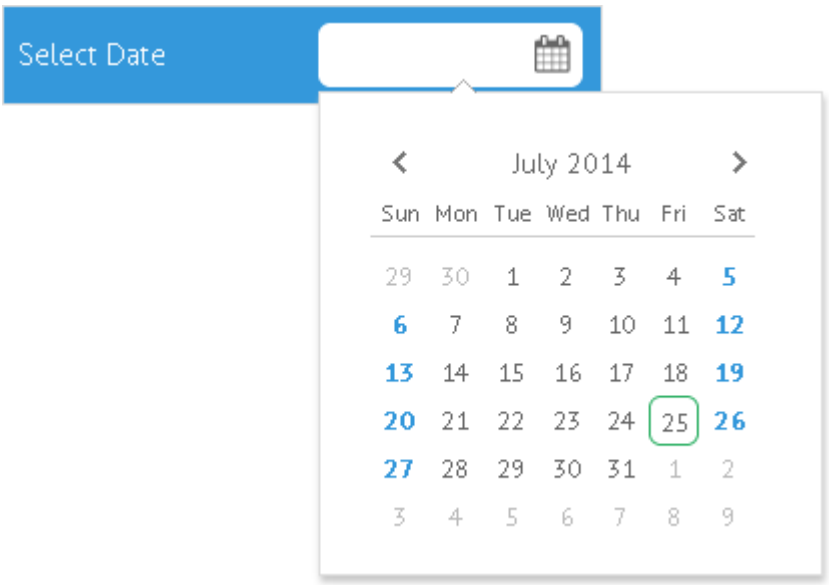
- **getValue()**; - common method for getting the control's value.
- **getSelectedDate()**; - calendar specific method.
- **getVisibleDate()**; - gets the date even with no selection applied. Date is defined according to the specified rules.

```
$$('calendar1').selectDate(new Date(2012,3,30));

$$("calendar1").getValue(); // returns the non-formatted date object
                            // -> Mon Apr 30 2012 00:00:00 GMT+0300 (EEST)
```

Datepicker

Calendar can be displayed on customer request, for instance when a user wants to pick a date and time to fill the form. For these needs a **datepicker** control is used. Note that here you needn't initialize Calendar - it will appear as soon as you click a datepicker icon. In this case the calendar comes with current date displayed.

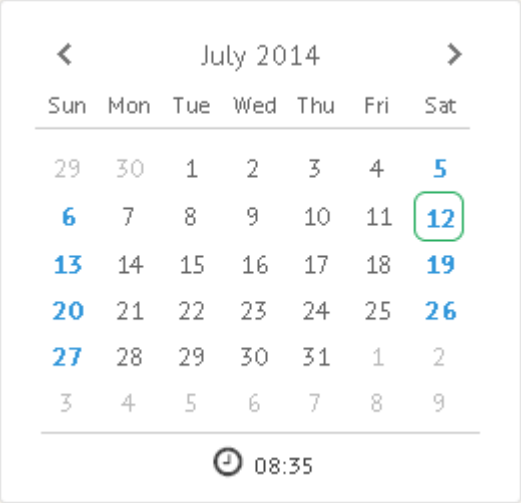



```
webix.ui({
    view:"toolbar",
    type:"MainBar",
    elements:[
        {view:"datepicker",  name: "select_date",  label: 'Select Date' }
    ]
});
```

RELATED SAMPLE: Date picker in Calendar

Timepicker in Calendar

Timepicker is a calendar feature that allows picking time. Being one of the component's properties, it is *false* by default:




```
webix.ui({
    view:"calendar",
    date: new Date(2012, 3, 16, 8, 35),
    timepicker:true,
    timepickerHeight:50 //optional, default is 30
})
```

With timepicker enabled, time is shown below month view within the calendar body. There're three options for the clock:

- If the time is defined in the *new Date(YYYY, MM, DD, HH, mm)* constructor, it's shown according to the specified values;
- If date is set with no time specified (new Date(2012, 3, 16)) - the clock shows midnight;
- If the *new Date* function comes with no parameters, current date and time is shown.

RELATED SAMPLE: Timepicker in Calendar

Time selector can be hidden and shown back by **timePicker** property with *true/false* value.

Localization

How do we turn a standard calendar into the one with "foreign" month and day names? Here localization can help you.

Locale is a set of rules that tell us how to display months and days. It's a kind of internal "translator" for UI components.

First of all you should specify a locale or a set of needed **locales** (en-US, fr-FR, ru-RU) and then apply them to the calendar. By default **en-US** locale is used.

```
webix.i18n.setLocale('fr-FR');
```

Afterwards the calendar will look like this:



RELATED SAMPLE: Localized Calendar

You can as well alter current working locale, e.g. change text of "Today" and "Clear" icons:

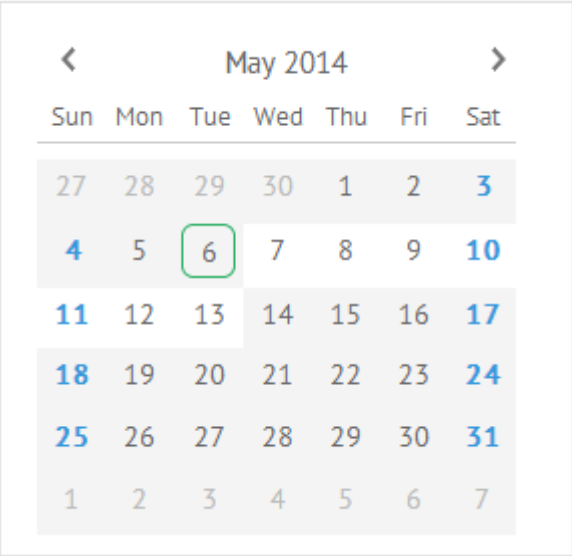
```
webix.i18n.calendar.clear: "Clear New";
webix.i18n.calendar.today: "Today New";
webix.i18n.setLocale();
```

You need to call the setLocale method to make changes come into force.

Disabling Period in Calendar

The whole calendar can be disabled by a common property:

```
{view:"calendar", disabled:true}
```



RELATED SAMPLE: Disabled dates

To disable a certain period in the calendar, which presupposes specific CSS and blocking of click events, you can go by the two ways:

Defining "active" period

Use the dedicated **minDate** and **maxDate** properties to limit the period that will be available for clicking and, hence, selecting:

```
{ view:"calendar", minDate:'2014-05-07', maxDate:new Date(2014, 4, 13) }
```

The dates can be defined either as **date object** or **date string** formatted under the current locale. Here the default **en-US** parseFormat is used, "%Y-%m-%d".

Defining "blockDates" function for custom logic

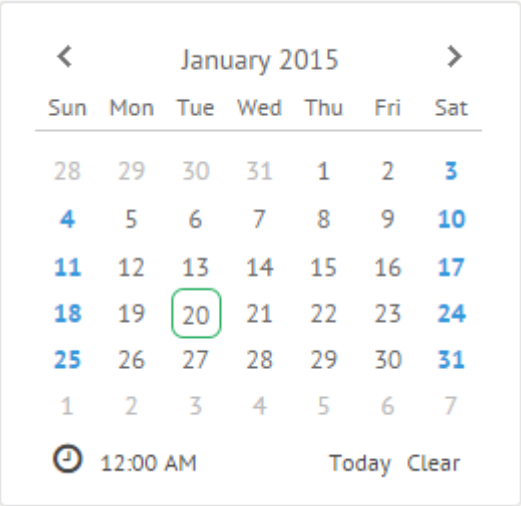
The function should return true for the dates that should be disabled in the calendar. Here all date up to 2014 will be disabled:

```
webix.ui({
    view:"calendar",
    blockDates:function(date){
        if(date.getFullYear()<=2013)
            return true;
    }
});
```

CSS class applied for disabled dates (**webix_cal_day_disabled**) can be redefined.

Today and Clear Buttons

The Today button allows selecting the current date. The Clear button removes date selection.



These buttons are optional and specified in the **icons** array. The code below creates the buttons' config:

```
icons: [
   // 'Today' definition
   {
        template: function(){
            return "<span class='webix_cal_icon_today webix_cal_icon'>"
                +webix.i18n.calendar.today
                +"</span>";
        },
        on_click:{
            "webix_cal_icon_today": function(){
                this.setValue(new Date());
                this.callEvent("onTodaySet",[this.getSelectedDate()]);
            }
        }
    },
     // 'Clear' definition
    {
        template: function(){
            return "<span class='webix_cal_icon_clear webix_cal_icon'>"
                +webix.i18n.calendar.clear
                +"</span>";
        },
        on_click:{
            "webix_cal_icon_clear": function(){
                this.setValue("");
                this.callEvent("onDateClear",[this.getSelectedDate()]);
            }
        }
    }
]
```

Both buttons have two properties:

- **template** - the template that defines presentation of an icon

- **on_click** - specifies the onClick handler that processes clicking an element with certain css

By default, these buttons are visible, but you can hide them by setting the **icons** property to **false**.

RELATED SAMPLE: Calendar

Related Articles

- Date Formatting Methods
- Number Formatting Methods
- Working with Dates
- Date and Number Localization
- Calendar CSS Image Map

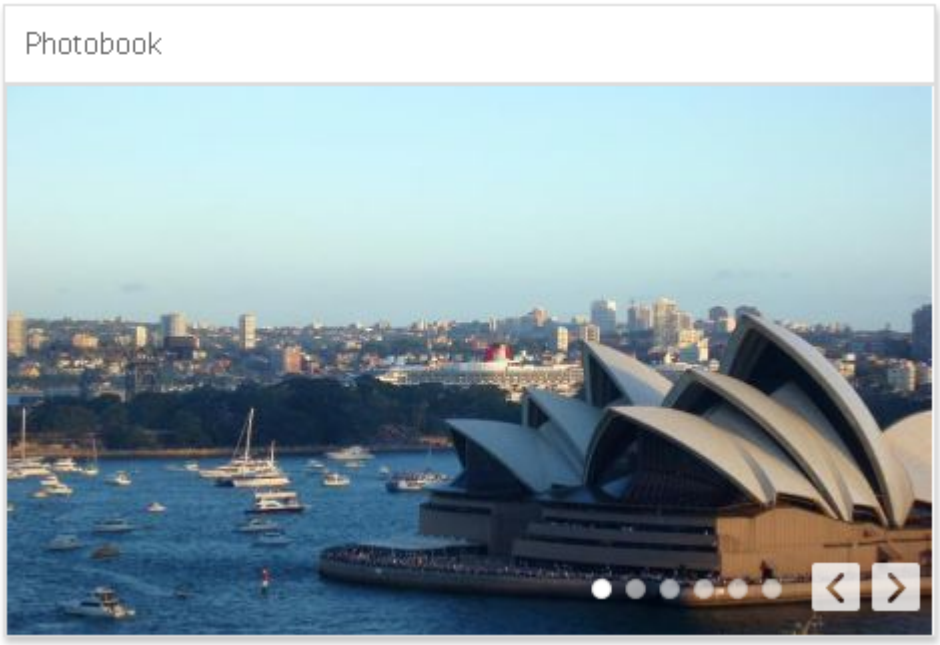
Carousel

API Reference

- Methods, Properties and Events
- Samples

Overview

Ui-related carousel is designed to present Webix components in one view with the ability to switch between them by clicking on the dedicated buttons or sliding (on touch devices).



RELATED SAMPLE: Carousel

Initialization

Carousel items are arranged into either **rows** to scroll through them vertically or in **cols** to scroll horizontally. Each row/column includes Webix component, e.g:

- simple or data template for showing texts and images;
- any data component like dataview, chart, datatable;
- layout of any complexity.

```
webix.ui({
    view:"carousel",
    id:"carousel",
```

```
    cols:[
        { view:"template" },
        { view:"chart" },
        { view:"datatable" },
        { rows:[...]} //layout
    ]
});
```

RELATED SAMPLE: Carousel

Normally, component configuration is stored separately while **cols/rows** contain an array of corresponding variables. It makes code clear and easy to read.

Navigation Area

Carousel comes equipped with a **navigation** panel that contains:

- **navigation items** (circles) that show the number of carousel views and highlight the opened one;
- **navigation buttons** (right/left) placed together to the right of the circles.

You can either click buttons or items to get to the needed view. On touch devices views are switched by swipe movements.

Redefining Navigation



RELATED SAMPLE: Navigation types

Navigation area is controlled via **navigation** object property of the carousel component. From there you can:

- **separate navigation buttons** and place them on the right and left sides of the view, define **"side"** type;

- **remove** navigation **items** (circles) or **buttons** by setting *false* value to the corresponding property.

```
view:"carousel",
cols:[...],
navigation:{
    type: "side",
    items:false,
    //buttons:false
}
```

Custom Navigation Area

Any Webix component can be used to navigate through the carousel by using its API.

For instance, you can use one-row dataview with thumbnails of carousel views.



RELATED SAMPLE: List navigation

In layout carousel and the component used for navigation are stored in different rows or cols:

```
rows:[
    {view: "carousel",
     id:"carousel",
     cols: []
    },
    {view: "dataview",
     id: "imageList",
     yCount: 1, //one row
     select: true, //item selection enabled
     scroll: false, //non-scrollable
     data: [...]
```

```
    }
]
```

Switching is enabled by catching carousel onShow and dataview onItemClick events:

```
// when dataview item is clicked, the needed view is shown
$$("imageList").attachEvent("onItemClick",function(id){
    $$(id).show();
});

//when carousel item is shown, its thumbnail in dataview is selected
$$("carousel").attachEvent("onShow",function(id){
    $$("imageList").select(id);
});
```

Learn more about how to show and hide Webix components and study selection pattern of data components.

Scrolling Speed

The **default scrolling speed** of the carousel is **300ms**. To change it use the following:

```
webix.ui({
    view:"carousel",
    scrollSpeed: "500ms"
});
```

However, in case of significantly slower scrolling speed smoothness may degrade.

Working with Carousel

1 . You can programmatically navigate through the carousel with the help of the following switching functions: - **showPrev()** - takes to the previous view; - **showNext()** - takes to the next view.

These functions can be attached to custom buttons or any on-page and application events:

```
 { view:"button", value:"Previous View", click:function(){
$$('carousel').showPrev(); }
```

2 . With touch-based devices initial scrolling pattern can be changed depending on the screen orientation. Scrolling is adjusted by the dedicated method:

```
$$("carousel").adjustScroll();
```

Setting the Active Element for the Carousel

3 . Active carousel item is a view that is currently shown. Carousel views can be treated either by their IDs or by their position in the **cols**/**rows** array (zero-based numbering).

- **setActive**(id) - shows a view with the specified ID;
- **setActiveIndex**(index) - shows a view with the specified index;.

4 . The same way you can get the currently shown (active) item:

- **getActiveId()**; - returns ID of the active view;
- **getActiveIdIndex()**; - returns index of the active view.

```
$$("carousel").getActiveIndex();
```

Related Articles

- Adding Headers to Components
- Event Handling
- Sizing Components
- Resizing
- Redefinition of Components
- Layout
- Multiview
- Scrollview
- Carousel CSS Image Map

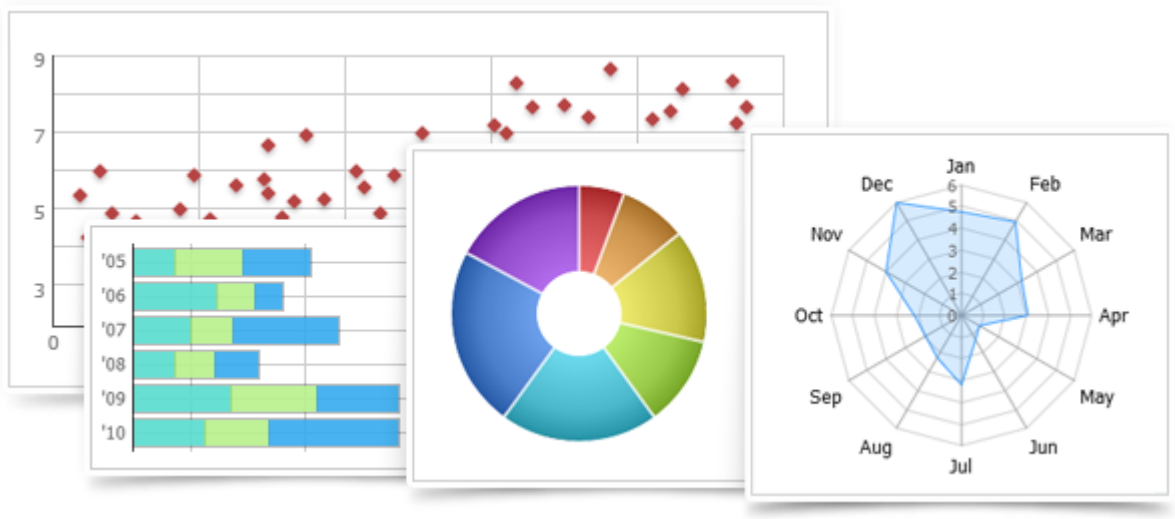
Chart

API Reference

- Methods, parameters and events
- Samples

Overview

Chart is a ui-related component that inherits from view and helps present data in different graphic figures like colored columns, pie pieces or lines. Like all the data-presenting components it takes initial data in various formats.



Initialization

The most important chart properties include **type**, **template** and **value** for data as well as criteria for data presenting (e.g. **step** of X axis in a bar chart). Specify the chart ID if you are going to work with it later.

Initialization of a Bar Chart

```
webix.ui({
            view:"chart",
            type:"bar",
            value:"#sales#",
            color:"#f6960a",
            border:true,
            xAxis:{
                template:"'#year#"
            },
            yAxis:{
```

```
                start:,
                end:100,
                step:10,
                ...
                }
            },
            data: dataset
        });
```

RELATED SAMPLE: Chart: XML Dataset

To learn about all the chart properties, please refer to the corresponding chapter of the manual.

Working with Chart

- Chart Types
- Setting Chart Scales
- Chart Series - Comparison Charts
- Setting Legend for a Chart
- Setting Chart Tooltip
- Data Grouping
- Operations with Chart Data
- Chart Colouring

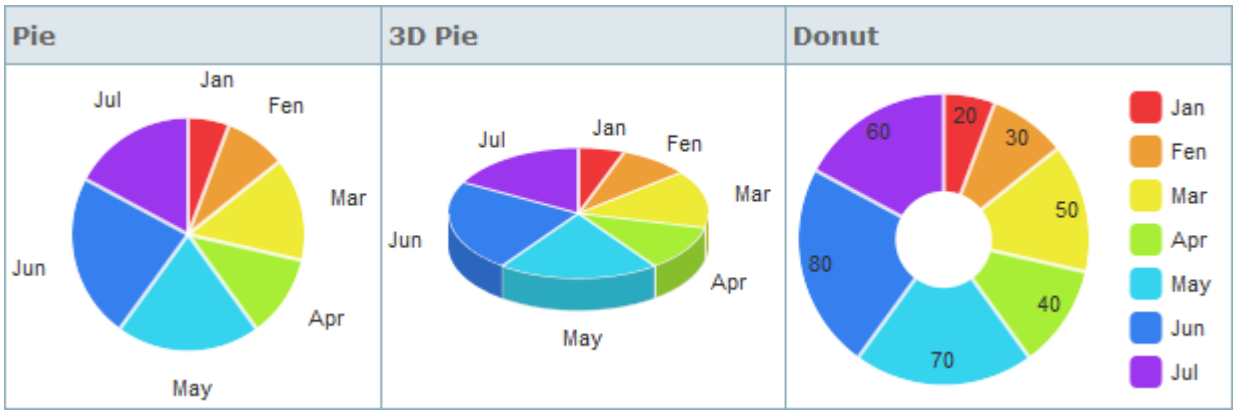
Related Articles

- Data Object Operations
- Supported Data Types
- Redefinition of the Components
- Event Handling
- Operations with Data Items
- Chart CSS Image Map

Chart Types

Here we speak about various chart types at your disposal. Basically, you should set the **type** property of the chart constructor to the necessary value to switch to the corresponding chart type.

Pie, 3D Pie Chart and Donut Charts



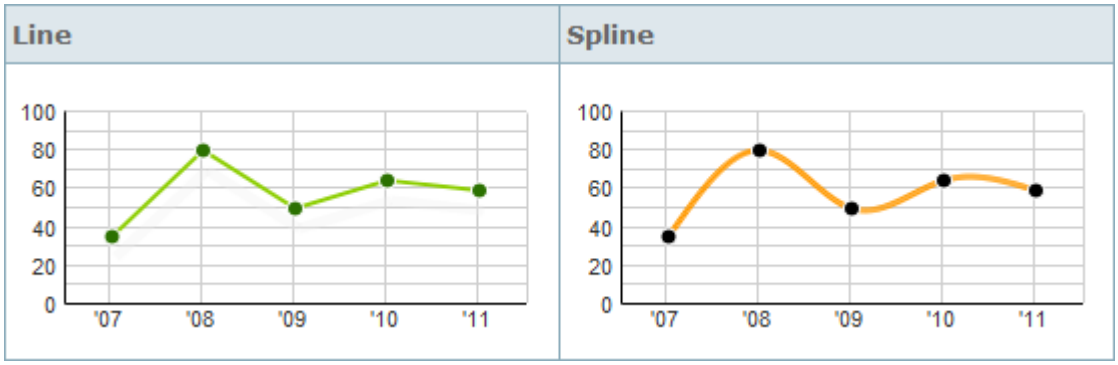

```
Initialization

webix.ui({
    view: "chart",
    type:"pie", //or "pie3D", "donut"
    ...
})
```

RELATED SAMPLE: Pie Chart: Basic Initialization

Settings for Pie Charts

Line and Spline Charts



Line chart features a broken line to join the points while a spline chart has a curved one.

```
Initialization

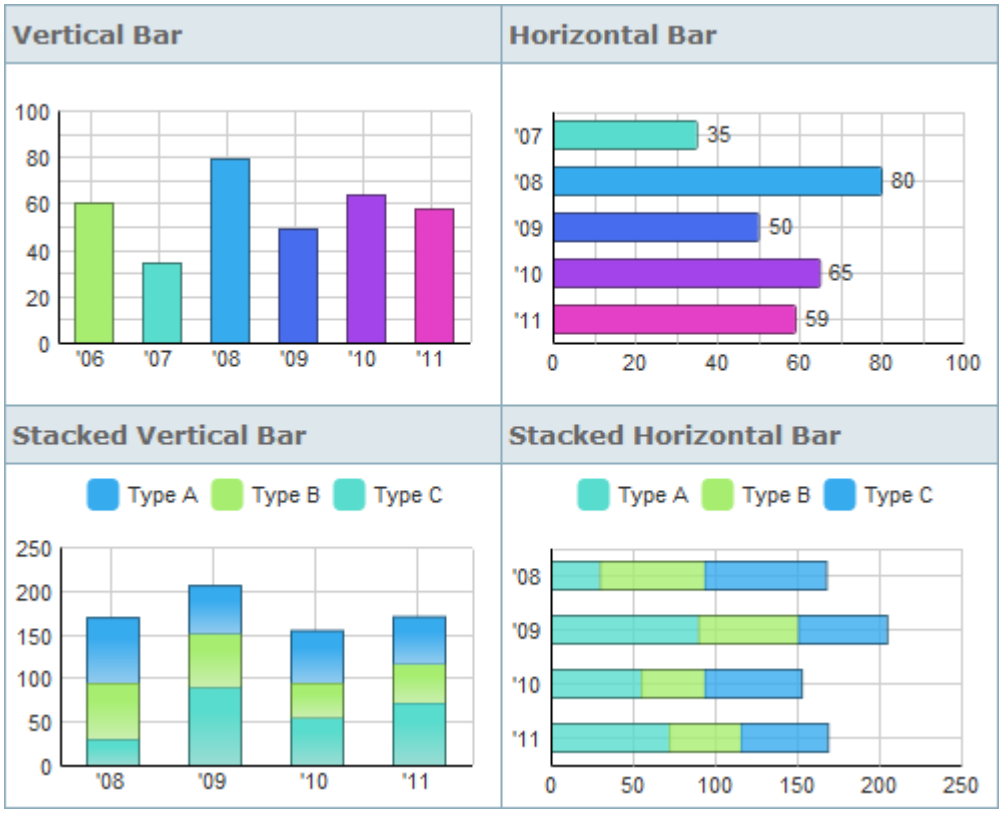
webix.ui({
    view:"chart",
```

```
    type:"line" //or "spline"
    ...
})
```

RELATED SAMPLE: Line Chart: Basic Initialization

Settings for Line Charts

Bar, BarH, StackedBar and StackedBarH Charts



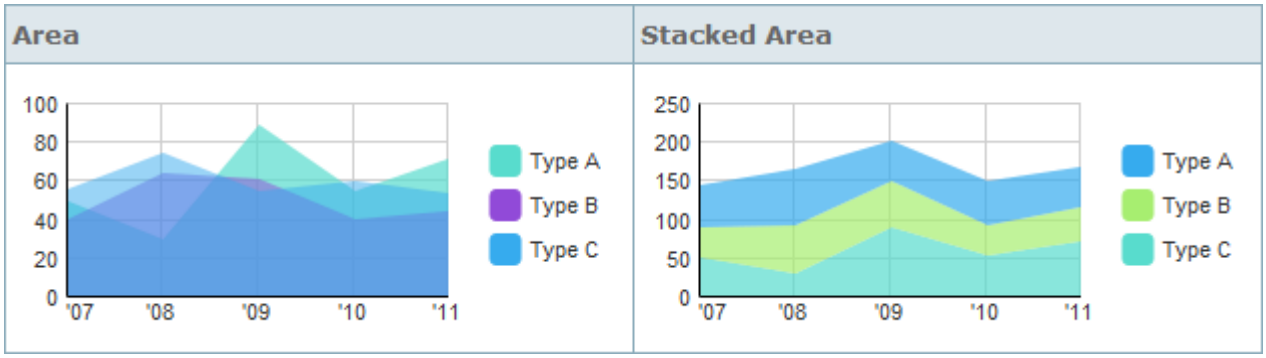

```
Initialization

webix.ui({
    view:"chart",
    type:"bar" //or "barH", "stackedBar", "stackedBarH"
    ...
})
```

RELATED SAMPLE: Bar Chart: Basic Initialization

Settings for Bar Charts

Area and stackedArea Charts



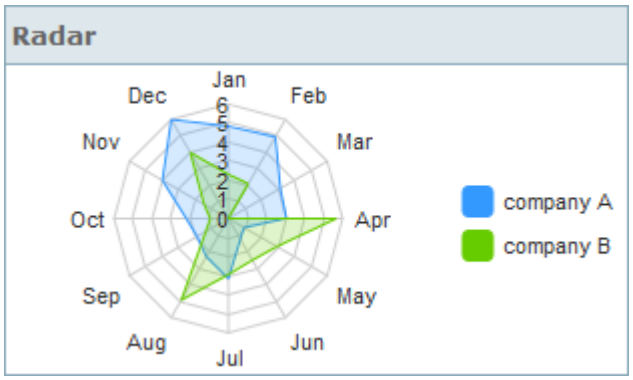

```
Initialization

webix.ui({
    view:"chart",
    type:"area", //or stackedArea
    ...
})
```

RELATED SAMPLE: Area Chart: Basic Initialization

Setting for Area Charts

Radar Charts



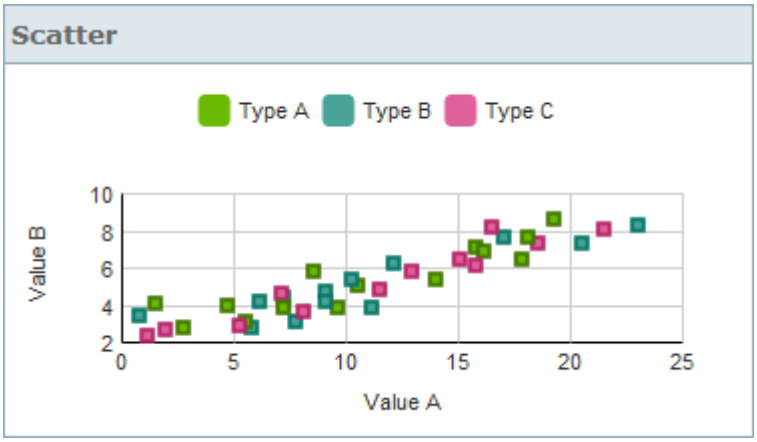

```
Initialization

webix.ui({
    view:"chart",
    type:"radar",
    ...
})
```

RELATED SAMPLE: Radar Chart: Points

Settings for Radar Chart

Scatter Charts




```
Initialization

webix.ui({
    view:"chart",
    type:"scatter",
    ...
})
```

RELATED SAMPLE: Scatter Chart: Basic Initialization

Settings for Scatter Chart

Settings for Pie Charts

Data Presentation

Template items are taken from the dataset defined for the component.

```
data=[{sales:"20", month:"Jan", color: "#ee3639"}]
```

- **value** ('#sales#') - data item that is taken for graphical representation. Its values form 'pie' pieces;
- **label** ('#month#') - data item used for making text label for 'pie' pieces;
- **labelOffset** (number) - the offset of sector labels from the pie frame (20px by default);
- **pieInnerText** ('#sales#') - text (numeric) values displayed on each 'pie' piece;
- **color** ('#color#') - color for each piece. Can be either in the dataset, or if not mentioned, default colors will be displayed.

Styling and Positioning

These settings can be easily omitted yet add style to your chart.

- **radius** (number) - radius of the chart 'pie';
- **x** (number) - horizontal offset of pie center from left side of the parent container;
- **y** (number) - vertical offset of pie center from the top border of the parent container
- **gradient** (boolean) - gradual color shifting;
- **shadow** (boolean) - decorative shadow for the component;
- **cant** (number) - defines the slope angle of 3D pie(from 0 to 1, 0.5 by default);

Back to all chart types

Settings for Bar Charts

Data Presentation

- **value** (template) - template from the dataset for data item to present graphically;
- **data / url** (string/url) - the datasource;
- **label** (template) - text label for each bar;

Learn how to draw chart scales in the dedicated chapter of the manual

Styling and Positioning

- **barWidth**(number) - width for each bar in pixels;
- **radius** (boolean) - if true, sets curved bar tops;
- **gradient** (boolean, string, function)) - gradient for each bar;
- **padding** (obj) - offset from the whole chart to the parent container {top:50, left:0, right:30, left:40}
- **color** (template, hex code, function) - color effects for the bars;
- **alpha** (number between 0 and 1) - transparency of chart bars;
- **border** (boolean) - if *false*, removes a black border of the bars. *True* by default.

Read more about Chart Coloring

Bar Chart Types

Table 1 Bar chart types

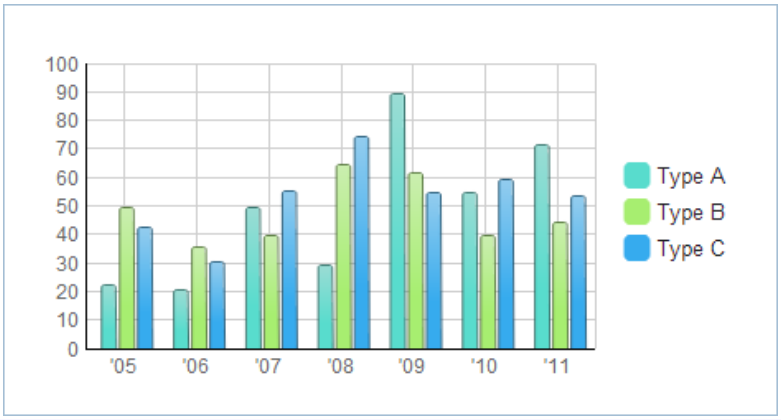
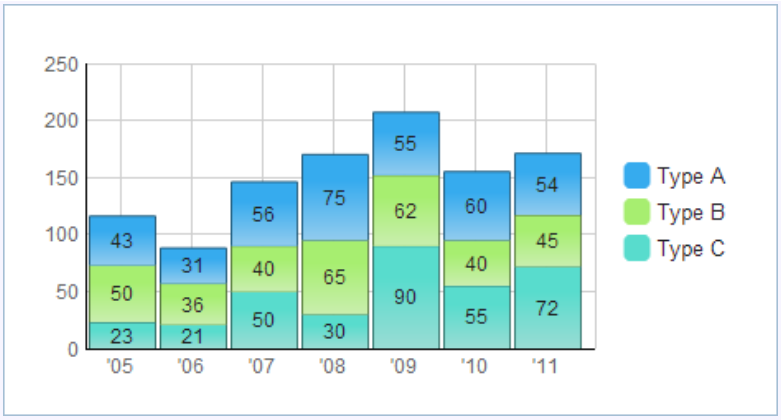
Type	Vertical Bars	Horizontal Bars
Single data	bar	barH
Several data values	stackedBar	stackedBarH

StackedBar and **stackedBarH** charts display data of several configuration objects (chart **series**), e.g. graphical representation of sales of three companies.

Bar Chart Series

Series can be defined for all bar charts. Series with **bar** and **barH** chart differ from those of **stackedBar** and **stackedBarH.** With the former bars are placed by each other in groups, which is greater for comparison. Moreover, the legend for them allows hiding and showing the dedicated chart by single mouseclick.

Table 2 Presenting several data properties on bar chart

Using series	
'stackedBar'/'stackedbarH' charts	

RELATED SAMPLE: Bar Chart: Several Graphs in One Chart

The **rules of series definition** are described here.

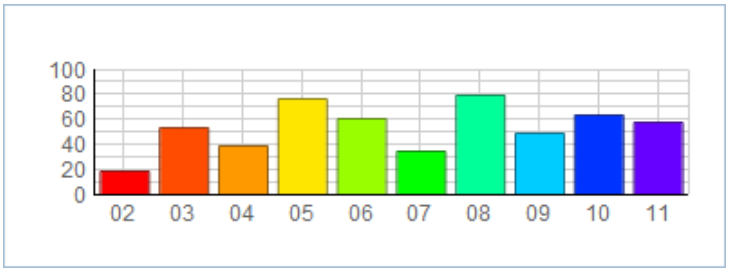
Presets for Bar Chart

The library provides 3 presets for Bar charts. To set one of presets you should specify the preset property for the chart:

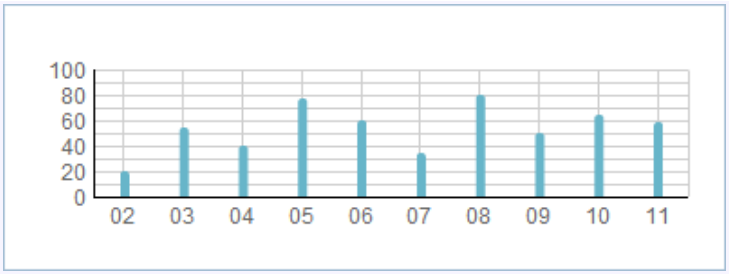
```
webix.ui({
    view: "chart",
    type: "bar",
    preset: "stick",
    ...
};
```

RELATED SAMPLE: Bar Chart: Style Presets

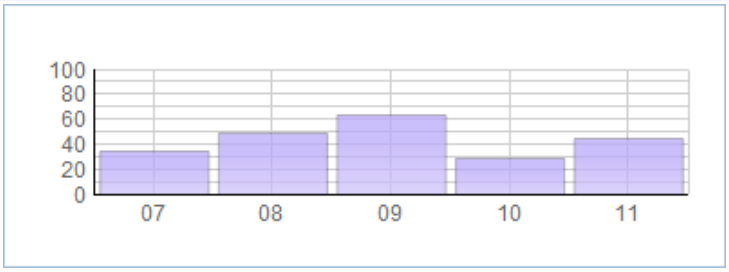
Table 3 Bar chart presets

preset:"column"	

```
preset:"stick"
```



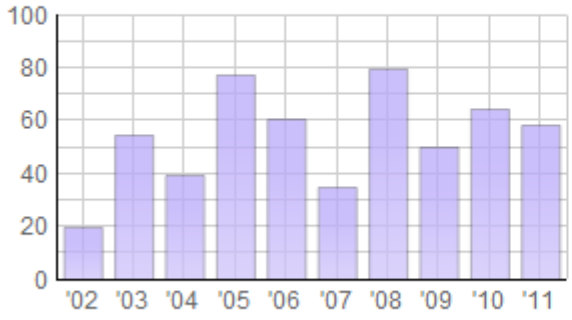
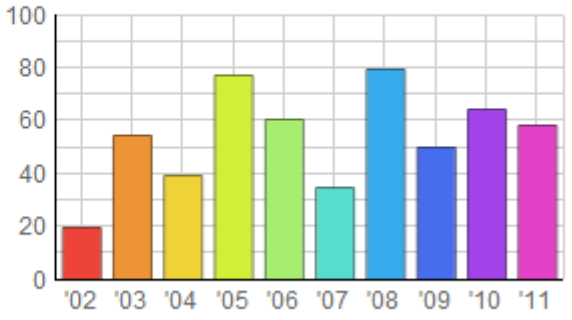

```
preset:"alpha"
```



Back to all chart types

Color Effects

Chart elements can be colored in various way, fixed color being defined by the **color** parameter that takes hex code. By default items are colored in a rainbow manner.

Fixed color	Color template
	

Custom color

Different color for items can be defined in the dataset already.

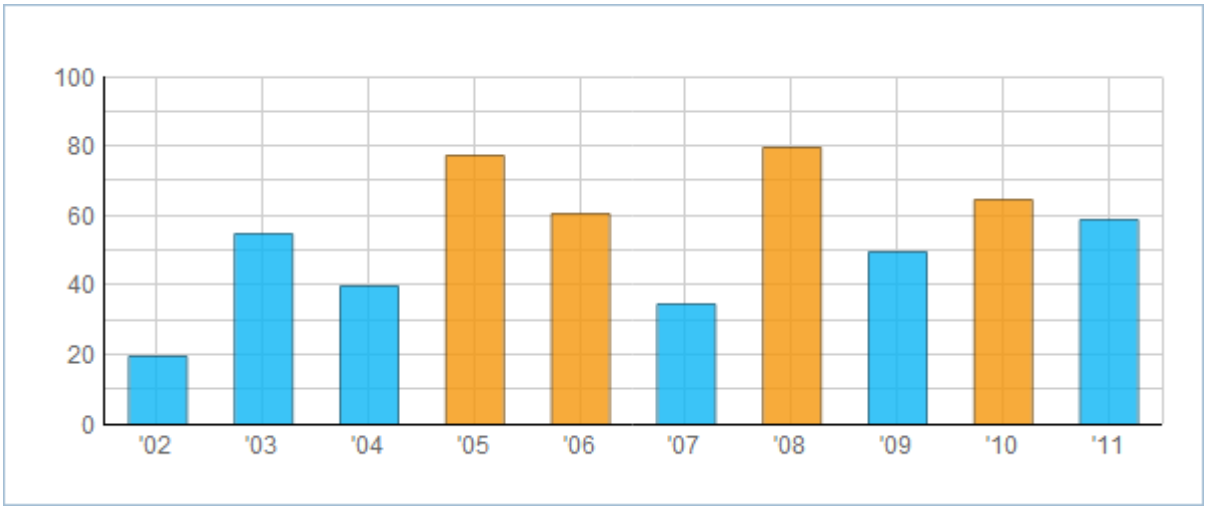
```
var data=[
    { id:1, sales:20, year:"02", color: "#ee4339"}
];

webix.ui({
    view:"chart",
    type:"bar",
    value:"#sales#",
    color:"#color#",
    ...
})
```

RELATED SAMPLE: Basic Coloring

Color Template(Function)

If you want chart elements being colored according to a specific criterion, you should define colors with the help of a function.

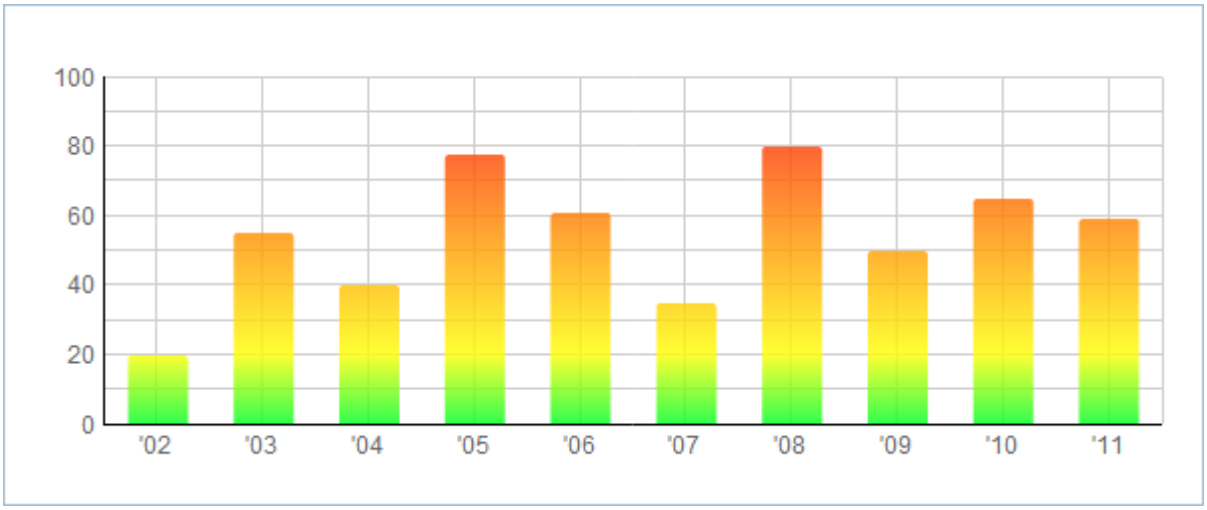



```
Here sales bars are distinquished by sales amount.

webix.ui({
    view:"chart",
    type:"bar",
    value:"#sales#",
    color:function(obj){
        if (obj.sales > 60)
            return "#f6960a";
        else
            return "#0ab6f6";
    }
})
```

RELATED SAMPLE: Specifying Color as a Function

Color Gradient



Gradient can be applied to bar and pie charts. It doesn't contain any information about the data, still it make the chart look more stylish and attractive.

Gradient property can take the following values:

- **boolean** - true or false, the easiest way to enable/disable gradient;

- **string** - "falling", "rising" to the direction of color shade; "light" and "3D" to add style effects for gradient;
- **function**.

Colors for gradient are defined with the help of a **addColorStop()** method that takes gradient position (a number between 0.0 and 1.0) and desired color codes as parameters.

```
gradient:function(gradient){
    gradient.addColorStop(1.0,"#FF0000");
    gradient.addColorStop(0.2,"#FFFF00");
    gradient.addColorStop(0.0,"#00FF22");
}
```

RELATED SAMPLE: Color Gradient

By default gradient is disabled in the pie chart, so you need to enable it by assigning **true**value for the gradient parameter.

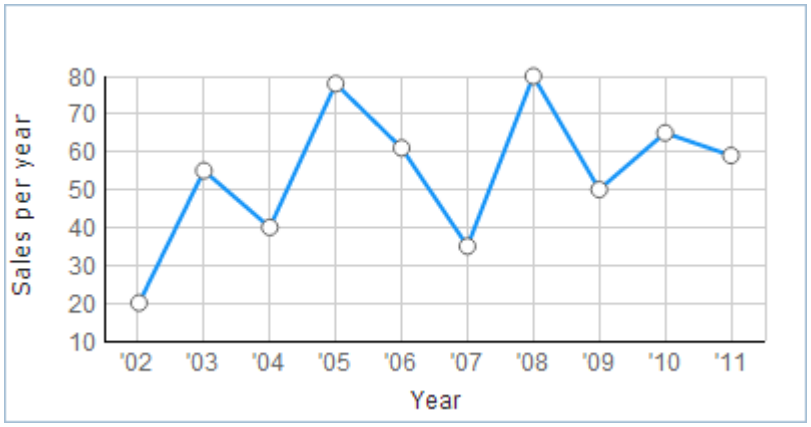
Setting for Line Charts

Here various properties of line charts are listed. The properties construct the chart object. Some of the properties contain an object as value. Data-presenting properties are compulsory while a number of styling options can be omitted or replaced.

Data Presentation

- **value** (template) - the data item chosen for graphical representation; as a rule, its values are taken for text labels of the Y axis;
- **data** / **url** (string/url) - datasource for the component;
- **label** (string) - text labels for each point on the line;

Setting titles for each axis:



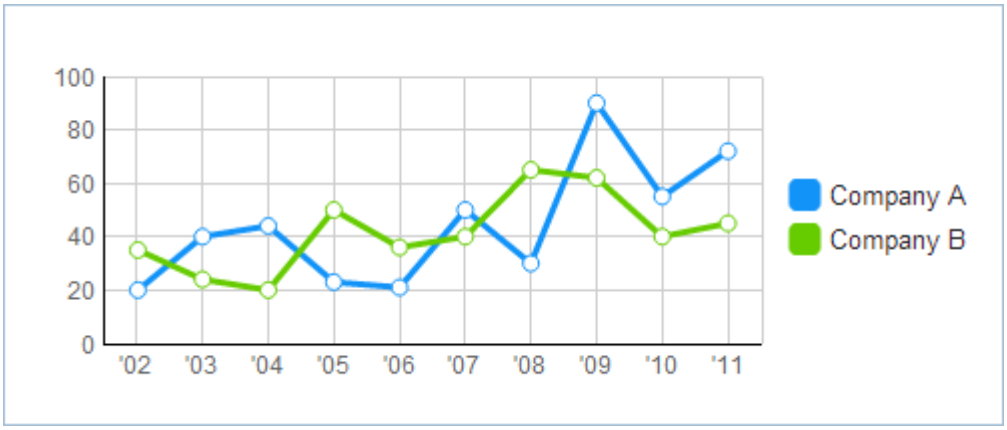
Here you should add **title** parameter for xAxis and yAxis constructor:

```
xAxis:{...
      title:"Some Text_1"},
yAxis:{...
      title:"Some Text_2"}
```

RELATED SAMPLE: Line Chart: Axes

Learn how to set scales for the chart in the corresponding chapter of the manual.

Chart Series. Several Graphs in One Chart



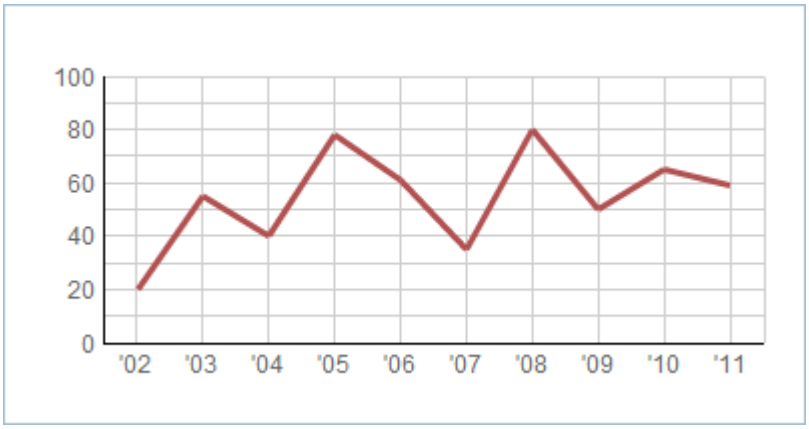
RELATED SAMPLE: Line Chart: Several Graphs in One Chart

Read more about Chart Series.

Styling and Positioning

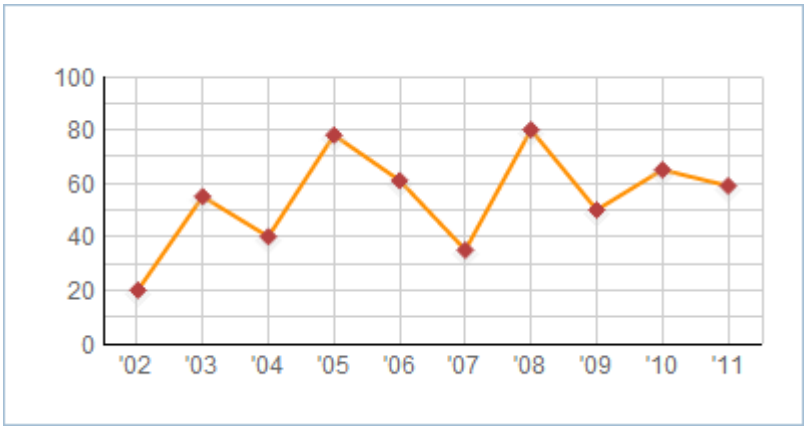
- **item** - a point on the line, a stop for each value. It includes:
  - **radius** (number) - radius on the point in the chart;
  - **eventRadius** (number) - radius of the virtual circle which events will fire for items within;
  - **borderColor** (hex code)- border for the point;
  - **color** (hex code) - inner color of the point.
- **line** - a broken or curved (in case of a spline chart) line.
  - **color** (hex code) - color for the line;
  - **width** (number) - line width.
  - **shadow** (boolean) - shadow for nthe line (optional);
- **offset** (number) - offset for the first item in a chart, the initial numeric value;

Getting rid of points on the line




```
item:{radius:}
```

Making square points for the line




```
item:{
       borderColor:"#b64040",
       color: "#b64040",
       type:"d",
       radius:3,
       borderWidth:1 //here you make it square
      }
```

RELATED SAMPLE: Line Chart: Styling

Style Presets

There're three style presets for the Chart component

- **plot** - a blue line with white points;
- **diamond** - an orange line with wine red square points;
- **simple** - a light green line with dark green points.

All of them are set with the help of **preset** property.

RELATED SAMPLE: Line Chart: Style Presets

Back to all chart types

Settings for Area Charts

- **value** (template) - item template from the dataset;
- **data / url** - the datasource;
- **color** (hex code, template) - color for the colored area;
- **alpha** (a number between 0 and 1) - the transparency of the colored area.

Area chart can be supplied with common functionality that is true for all chart types:

- **Scales,** horizontal and vertical lines with values. See Rules of Scale Building for details.
- Tooltips, pop-up infotips that show the exact value of the key points.
- Legend, colored markers that explain what each chart within the scales mean.

Chart Series

Both **Area** and **StackedArea** charts support presenting graphs for several objects (series).

Series for chart is an array of objects with settings for each graph. Details here.



RELATED SAMPLE: Area Chart: Several Graphs in One Chart

RELATED SAMPLE: Stacked Area Chart

Back to all chart types

Settings for Radar Chart

Custom Initializzation

Data Presentation

- **value** (template) - item template from the dataset;
- **data / url** - datasource;
- **item** - object property that contains settings for radar points. If you don't want to set point, omit this setting.
  - **borderColor** (hex code) - border color of the point;
  - **radius** (number) - radius on the point in the chart;
  - **eventRadius** (number) - radius of the virtual circle which events will fire for items within;
  - **color** (hex code, template) - color for radar points;
  - **borderWidth** (int) - border width of the point (for square points;
- **line** - object property that contains setting for the line
  - **color** (hex code) - color of the outline;
  - **width** (number) - width of the outline;
- **fill** (hex code) - color to fill the area limited by the outline;
- **disableLines** (boolean) - enable/disable the line that joins points.

Scales

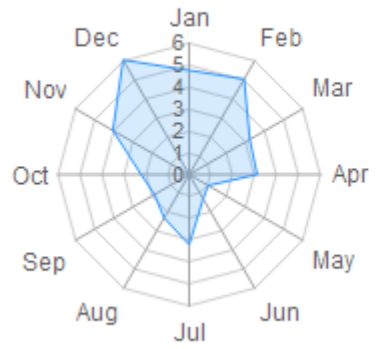
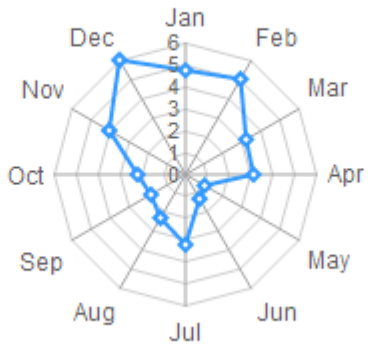
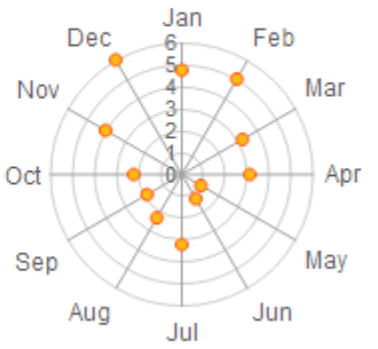
- **xAxis** - label for each radar line that goes from the center to the outside:
  - **template** (string) - text labels for lines;
  - **lineColor** (hex code) - color of radar lines;
- **yAxis** - settings for the radar map and inner lines(circles):
  - **lineShape** (string) - shape of the outer border of the radar area. '*arc*' helps make a circle.
  - **bg**(hex code) - background color for the radar area;
  - **lines** (boolean) - if *false*, inner circles of the radar area aren't shown

Radar Charts Presets

The library provides 3 presets for Radar charts. To set one of presets you should specify the preset property for the chart:

```
webix.ui({
    view: "chart",
    type: "radar",
```

```
    preset: "line",
    ...
};
```

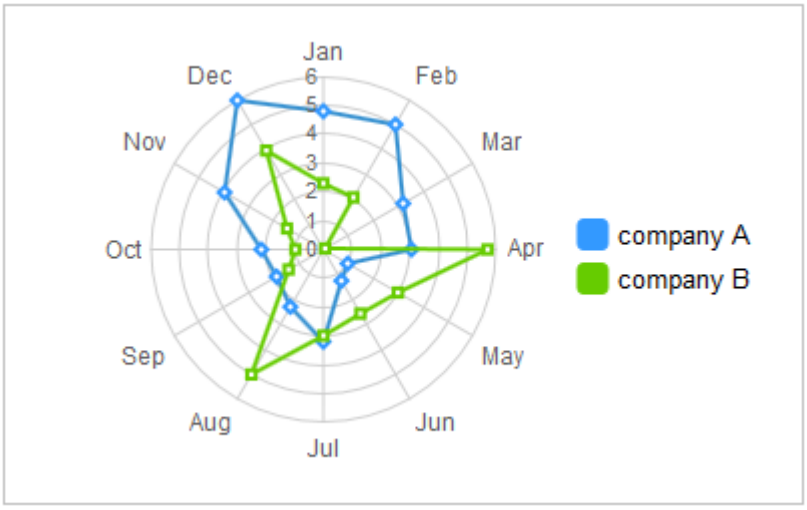
preset: 'area'	preset: 'line'	preset: 'point'
		

RELATED SAMPLE: Radar Chart: Style Presets

Radar Chart Series

Chart series is an array of settings for each graph to display different data in comparison.

In case of chart series, legend markers become buttons to show and hide dedicated charts.



RELATED SAMPLE: Radar Chart: Lines

Read more about Chart Series.

Back to all chart types

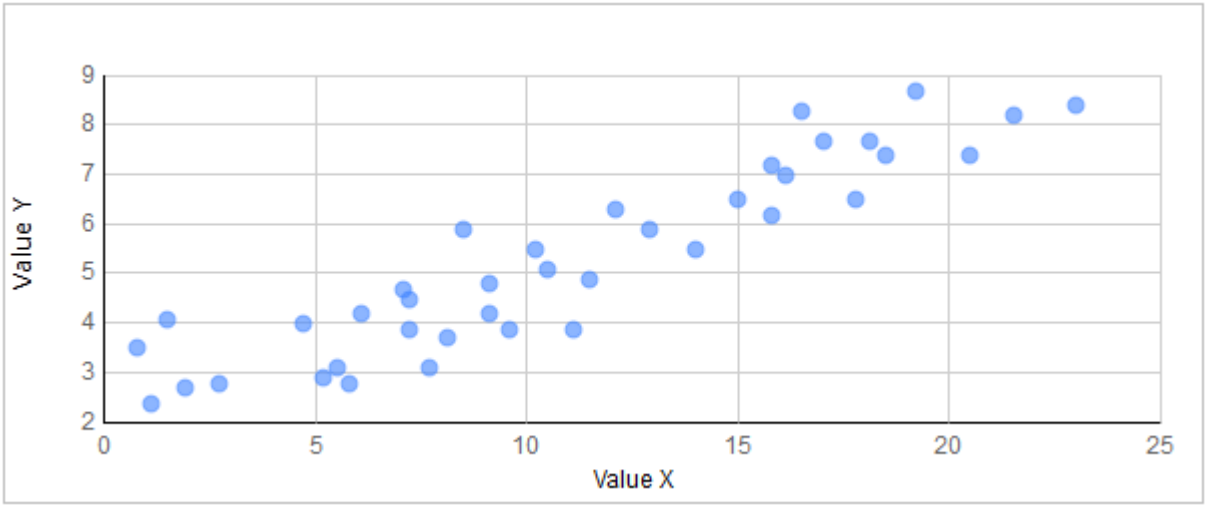
Settings for Scatter Chart

Custom Inititalization

- **value** (template) - item template for Y axis;
- **xValue** (template) - item template for xAxis;
- **data /url** (url/string) - the datasource;
- **item** - an object property that contains setting for each point in the graph:
  - **radius** (number) - radius on the point in the chart;
  - **eventRadius** (number) - radius of the virtual circle which events will fire for items within;
  - **borderColor** (hex code) - the color of the point border;
  - **borderWidth** (number) - the width fo the border (if set, point take square shape);
  - **color** (hex code) - color of the points
  - **shadow** (boolean) - if *true* makes point cast shadows. *False* by default.

Style Presets

Presets simplify chart initing and contain item setting. They are defined by the **preset** property and can be of the following types:



- **round** - semi-transparent blue round points (on the picture above)
- **square** - green squares with borders
- **diamond** - wine red diamond-shaped points

RELATED SAMPLE: Scatter Chart: Style Presets

Scatter Chart Series

Chart series allow presenting different graphs within on eand the same chart. The graphs differ in color. Settings for the graphs form objects that are placed into the **series** array.

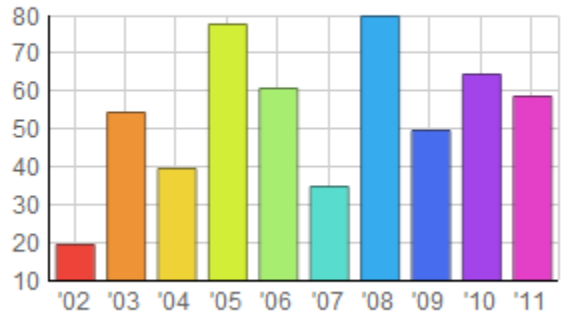
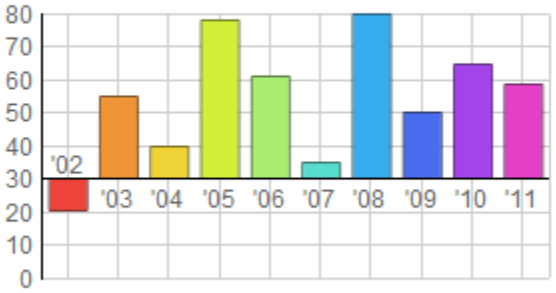


RELATED SAMPLE: Scatter Chart: Several Graphs in One Chart

See the common rules of **series definition** here.

With chart series, legend should be used to distinguish between graph values. At the same time, legend markers become clickable and allow to show and hide dedicated graphs.

Setting Chart Scales

Default scale	Custom scale
	

Scales are included into the following chart types:

- Line and spline charts
- Bar charts (optionally)
- Area charts
- Scatter chart

At the same time, radar chart has a specific chart type.

Scales consist of **Y** and **X** axes that are chart properties which values are objects containing several properties. Note that scales won't appear during chart initialization by default if you don't include these properties into the chart constructor:

```
webix.ui({
    view:"chart",
    ...
    xAxis:{
        property:"value"},
    YAxis:{
        property:"value"}
})
```

- A vertical, **yAxis** of a chart is formed according to the values of data, which progress you want to show graphically. Naturally, the values are numeric. This data is specified by chart's **value** property.
- A horizontal, **xAxis** displays progress of the data according to the specified criterion.

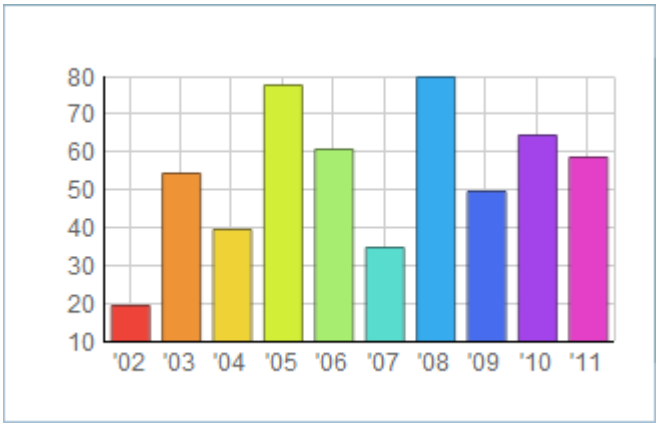
If needed, any axle can be omitted.

Take the dataset looks as follows and we want to present sales progress during several years. Each data item contains sales amount and the year.

```
JSON data
```

```
[
    { id:1, sales:20, year:"02"},{ id:2, sales:55, year:"03"},
    { id:3, sales:40, year:"04"},{ id:4, sales:78, year:"05"},
    { id:5, sales:61, year:"06"},{ id:6, sales:35, year:"07"},
    { id:7, sales:80, year:"08"},{ id:8, sales:50, year:"09"},
    { id:9, sales:65, year:"10"},{ id:10, sales:59, year:"11"}
]
```

Default Scale



Y-axis Formation

If you set an empty yAxis property object **(yAxis:{})**, the dataset will be anylized and the scale will be formed automatically. Here:

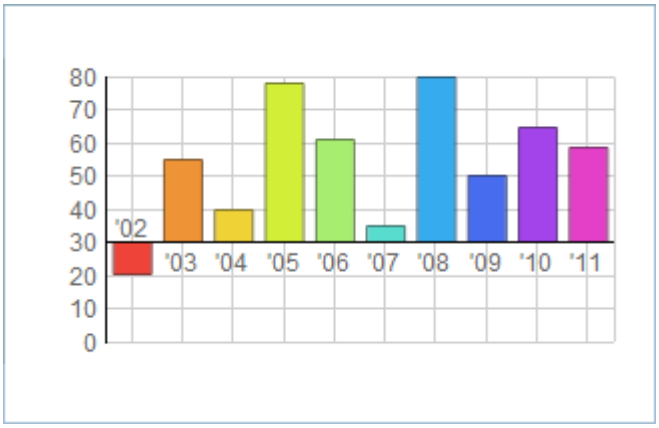
- the optimal step for units will be 10 as all the values are lower than a hundred;
- the lowest unit value for the scale will be 20 as the lowest value in the dataset is 20 and the programm takes the closest ten's place.
- the highest unit value for the scale will be 80 since it's the highest data value.

X-Axis Formation

Minimal compulsory API for the x-Axis includes

- **template** - data item from the dataset. Here is it **#year#**. Each value turns into a labelled unit on the line.

Custom Scale



Y-axis Formation

Y-axis object is constructed with the following properties:

- **start** (number) - start position for the axis, the value of the first unit;
- **end** (number) - max value for the axis, the last unit;
- **step** (number) - 'distance' to the next unit (not necessarily labelled with text);
- **template** - template for text label of the axis. Template function can define which numeric values to display. If not specified, text values are displayed depending on the 'step' parameter.

If you don't want to label each unit, you can set a function template to label only units multiple of the specified number (here it's 20):

```
template:function(obj){return (obj%20?"":obj)
```

X-axis Formation

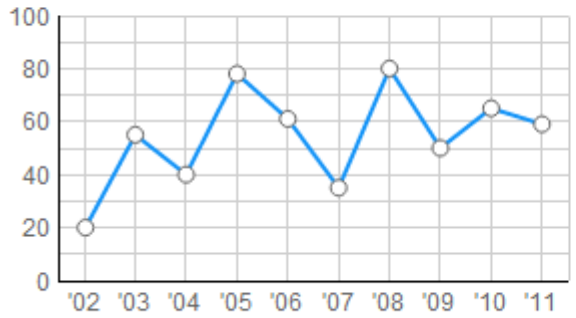
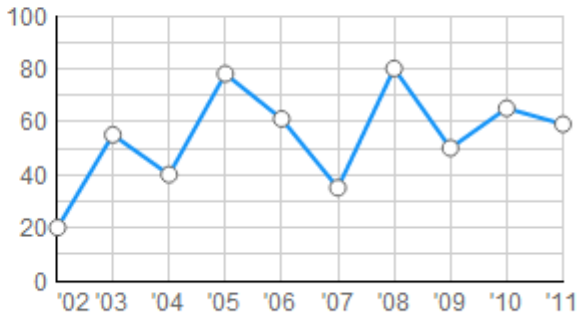
- **template**(template) - template for text labels on the X axis;
- **lines** (boolean) - if *false*, the line for the axis isn't shown. By default this value is *true*.
- **origin** (number) - it's the property of the chart constructor, not the xAxis object. However, its value affects the position of x-Axis relative to the Y-axis. Data items with values lower that the stated one, are displayed below the line.

There exists a possibility to create a fully custom axis for working with dates.

Offset

For line, scatter and area charts you can set the offset property which defines whether the first item of the scale will be drawn with the offset equal to the half of the scale's step (relative to the origin of the chart).

The default value for line and scatter charts is *true*. The default value for area charts is *false*.

offset:true	offset:false
	


```
webix.ui({
    view:"chart",
    type:"line",
    ...
    xAxis:{
        property:"value"},
    YAxis:{
        property:"value"},
    offset:false
})
```

RELATED SAMPLE: Line Chart: Axes

Logarithmic Scale

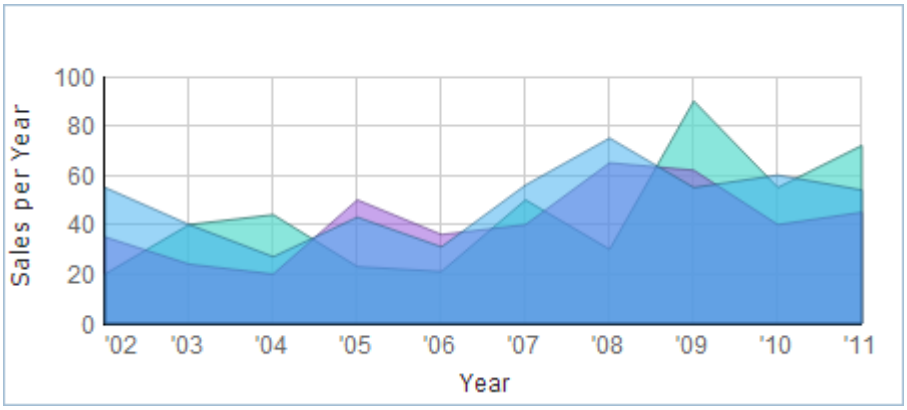
Webix charts are equipped with two types of **scale**:

- **"linear"** - used in all charts by default;
- **"logarithmic"** - enabled by chart **scale** property:

```
webix.ui({
    view:"chart",
    type:"bar",
    scale: "logarithmic",
    ...
});
```

RELATED SAMPLE: Chart: Logarithmic Scale

Titles for the Axes



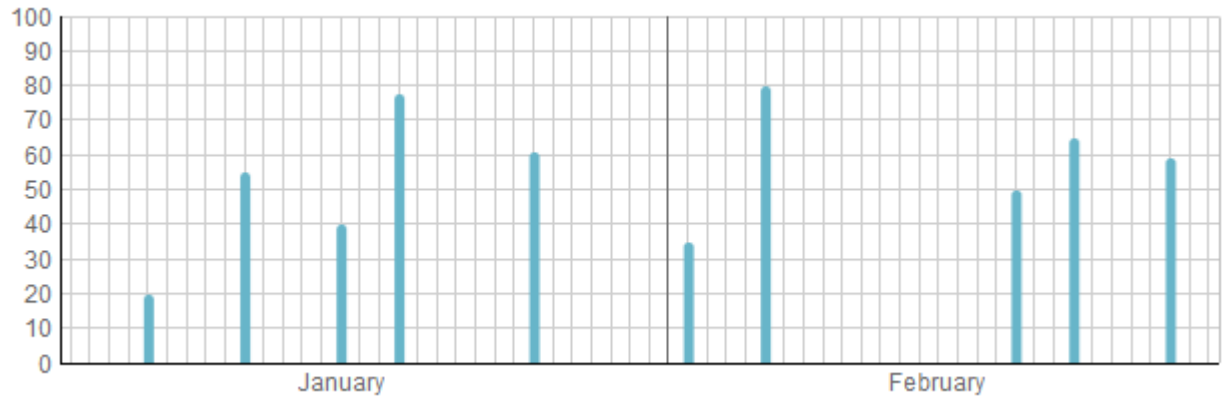
Titles are included into axes' objects:

```
webix.ui({
    view:"chart",
    ..config..,
    xAxis:{ ..
            title: "Year"},
    yAxis:{..
            title:"Sales per Year"}
})
```


In case of horizontal bar charts (type **barH**, **stackedBarH**) X and Y axes exchange their properties.

Custom X-Axis Formation

Sometimes you need to create a detailed scale, especially when you work with dates.



RELATED SAMPLE: Bar Chart: Custom xAxis

Here you work with dates and sales per date stored in JSON dataset. Note that dates are stored as strings.

```
var dates_dataset = [
    { id:1, sales:20, date:"05.01.2012", color: "#ee4339"},
    { id:2, sales:55, date:"10.01.2012", color: "#ee9336"},
    { id:3, sales:40, date:"15.01.2012", color: "#eed236"},
    ...]
```

Sales are represented by y-Axis values, while dates form the **timeline** on the x-Axis, each unit representing a day of month. The quantity of unit per month should be equal to the real quantity of days in this or that month, which isn't a static value.

In this case, the looks of the X-axis is totally customized features several object and function properties, namely **template, lineColor, value** and **units**.

Units

The object properties set the first and the last unit for the scale as well as sets a step between neighbouring units.

```
units:{
       start:new Date(2012,,1), //starting from January, the 1sr
       end:new Date(2012,1,29), //finishing on February, the 29th
       next:function(d){
             return  webix.Date.add(d, 1, "day", true);
            }
```

Step is set by the add function that takes **date object**, **number of units** to add and **unit name** as parameters.

Template

Template defines text label for an axis. Normal chart scale features one label below an axis, but here we need to underwrite each month of the timeline.

The function below allows placing the right label under the 15th day of each month:

```
 template:function(obj){
                return
obj.$unit.getDate()==15?webix.Date.dateToStr("%F")(obj.$unit):""
            },
```

The function gets day number from the Date object of each unit. If it equals to 15, the date is parsed to string and formatted to output the full name of the month. Otherwise, the unit receives no label.

LineColor

Line color within x-Axis object defines the color of vertical scale units. In this case we need that a line for the first day of February is darker than the others.

```
 lineColor:function(obj){
                return obj.$unit.valueOf() == (new
Date(2012,1,1)).valueOf()?"#737373":"#cfcfcf"
            }
```

The function defines the value of each unit. If is equals fo the February, the 1st, it return dark grey (#737373). Otherwise, the color is light grey.

Value

Values are not displayed right in the chart, yet shown with a help of the tooltip. Here we can set an approproate formatting method for

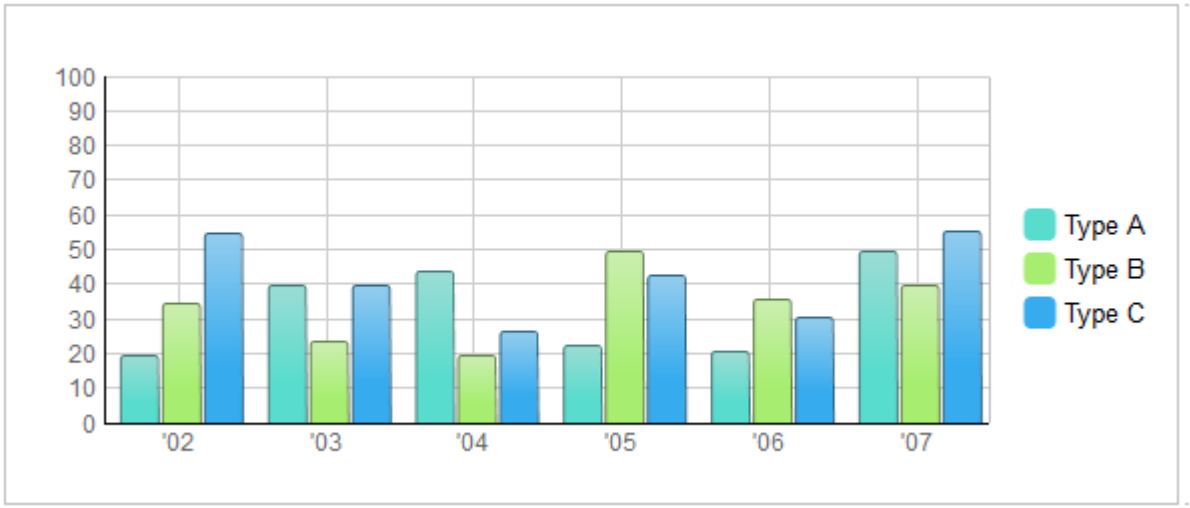
Date string is taken from the dataset (obj.date) and turned to date object formatted under the "*day number-month number-full year*" pattern.

```
  value:function(obj){
                console.log(webix.Date.strToDate("%d.%m.%Y")(obj.date))
                return webix.Date.strToDate("%d.%m.%Y")(obj.date);
            },
```

Chart Series

Chart series help display comparative graphs. In other words, you have an ability to present data of different objects within one and the same shart to see their dynamics in comparison. The functionality is true for line, bar, radar and area charts.

For instance, you want too see the sales progress of three companies during some time period.



First of all, each item from dataset must contain data for all the needed objects (here - companies).

```
JSON Data

var multiple_dataset = [
    { sales:"20", sales2:"35", sales3:"55", year:"02" },
    { sales:"40", sales2:"24", sales3:"40", year:"03" },
    { sales:"44", sales2:"20", sales3:"27", year:"04" },
    { sales:"23", sales2:"50", sales3:"43", year:"05" },
    ....
];
```

Sales, *sales2* and *sales3* are sales amounts for three different companies, each of which should be represented in a separate chart. On the picture above, there are three bar chart series differing from each other in color.

In fact, here we initialize one component - **bar chart** and distribute data among its **series**. Hence, settings for the bar charts form an array called **series**. Each array item includes data values as well as settings for each series.

```
webix.ui({
    view:"chart",
    type:"bar",
    series:[
        {   // 1st company - turquoise blue bars
            value:"#sales#",
            color: "#58dccd",
```

```
            tooltip:{
                template:"#sales#"
            }
        },
        {   //2nd company - green bars
            value:"#sales2#",
            color:"#a7ee70",
            tooltip:{
                template:"#sales2#"
            }
        },
        {   // 3rd company - blue bars
            value:"#sales3#",
            color:"#36abee",
            tooltip:{
                template:"#sales3#"
            }
    }]
})
```

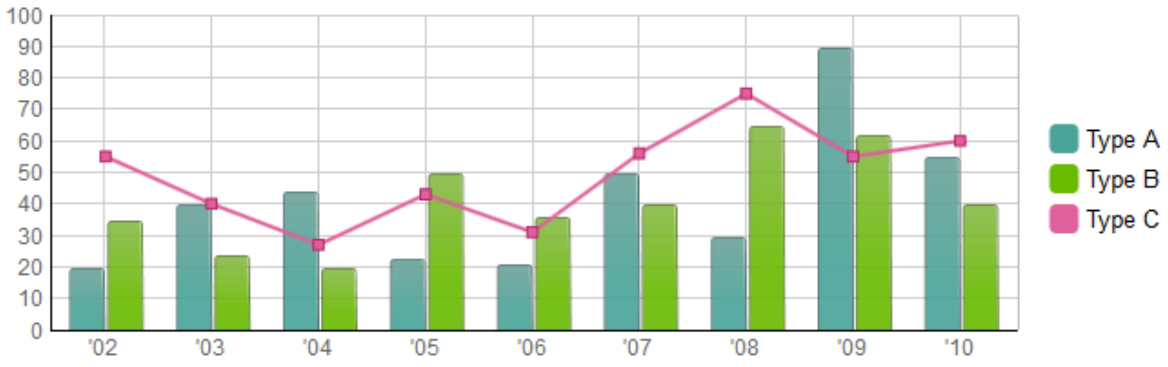
RELATED SAMPLE: Bar Chart: Several Graphs in One Chart

Chart series require a legend to explain different charts within one and the same scales. Legend markers act as well as buttons to show and hide dedicated chart series.

If you work with one and the same chart type, the **series** object only **item** settings should be included. Look for them in the setting for each chart type.

Different Chart Types

Still, chart series can encompass different chart types.



RELATED SAMPLE: Bar Chart: Different Types in One Chart

This time, the **series** object for a differing chart type should include the whole set of its properties:

```
webix.ui({
    container:"chartDiv",
    view:"chart",
    type:"bar",
    //bar chart config
    series:[
        {   //setting for each bar
            value:"#sales2#",
            color:"#69ba00",
            tooltip:{
                template:"#sales2#"
            }
        },
        {   //setting for a line chart
            type:"line",
            value:"#sales3#",
            color:"#36abee",
            item:{
                borderColor: "#b7286c",
                color: "#de619c",
                type: "s"
            },
            line:{
                color:"#de619c",
                width:2
            }
    }]
})
```

Operations with Chart Data

Getting an Item's Value

```
Getting the value of **year** property


            $$("chart").attachEvent("onItemClick", function(id){
            id = this.getItem(id).year;
            alert(id);
        });
```

RELATED SAMPLE: Chart: Events

Adding Items

While adding an item, make use of the **add()** method and state new data as an object and position of the new item:

```
var counter = 12;
        function addNew () {
            $$("barChart").add({
                year:"'"+counter,
                sales:"some_value"
            });
            counter++;
        }
```

Additionally, you can sefine a **counter** that sets the starting position of the new items to observe dataset integrity.

Item Deletion

To delete an item you should pass its ID into the **remove()** function (zero-base numbering) or make use of **first()** and **last()** methods to delete first and last items from the dataset.

```
$$("barChart").remove(4); // the 5th item will be removed

function deleteFirst(){
            $$("barChart").remove($$("barChart").getFirstId());
        }
```

RELATED SAMPLE: Chart: Adding/Deleting Data Items

Study the corresponding chapter of the manual to learn more about adding and deleting of items.

Sort Data

Here you use a **sort()** method that takes property from the initial dataset and sorting manner (ascending or descending) as parameters.

```
$$('chart').sort('#year#','desc');
```

RELATED SAMPLE: Chart: Sorting

Sort method in API Reference

Filter Data

As a rule, **filter()** method takes a function as parameter. In the function you specify the data you want to be filtered.

```
Make the chart show you only sales exceeding 50.

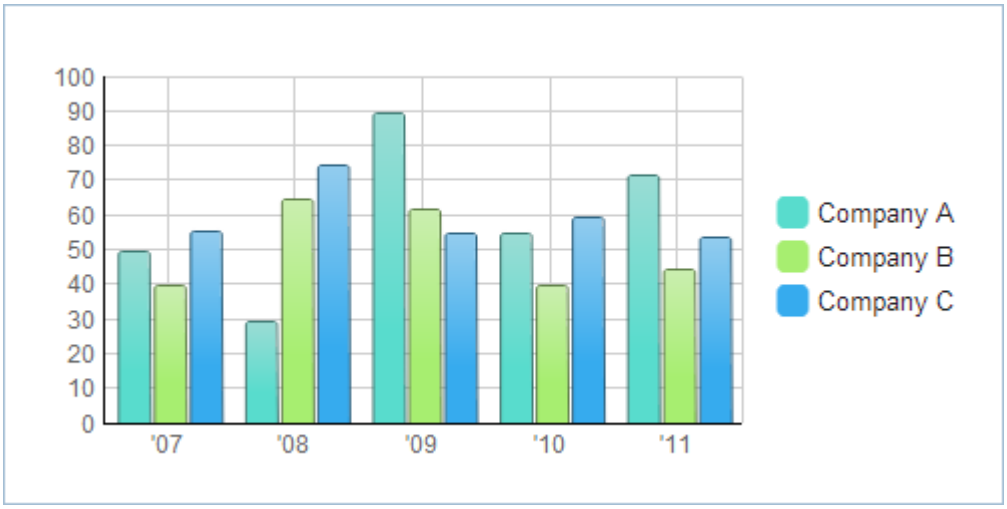
function filter1(){
            $$('chart').filter(function(obj){
                return obj.sales >50;
            });
            }
```

RELATED SAMPLE: Chart: Filtering

Rules of Data Filtering

Setting Legend for a Chart

Chart legend explains data and color correspondance. Generally, to specify the chart legend you should use the legend property:



Specifying the legend for a chart

```
webix.ui({
    view:"chart",
    type:"bar",
    ...
    legend:{
        values:[
            {text:"Company A", color:"#58dccd"},
            {text:"Company B", color:"#a7ee70"},
            {text:"Company C", color:"#36abee"}
        ],
        align:"right",
        width:110,
        layout:"y"
    }
})
```

RELATED SAMPLE: Bar Chart: Several Graphs in One Chart

See the full list of **legend parameters** in API Reference.

Default Legend

Built-in means allow defining a default legend. For these needs, you should specify what element from the initial datasource will be in the legend area. Default legend is great for pie and bar charts with multicolored bars and pie pieces.

Then, the legend takes a color value from the dataset:

```
JSON Dtata

[
    { sales:"20", month:"Jan", color: "#ee3639" },
    { sales:"30", month:"Fen", color: "#ee9e36" }
]

webix.ui({
    view:"chart",
    type:"pie",
    value:"#sales#",
    color:"#color#",
    legend:"#month#",
    ...
});
```

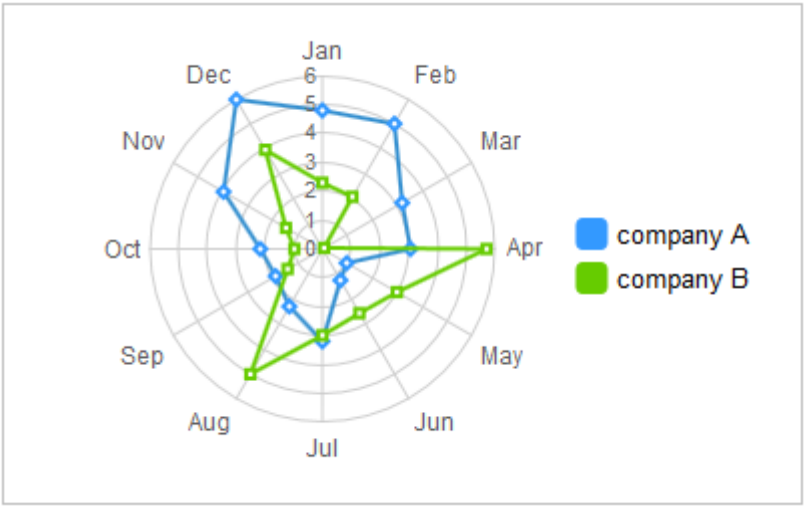
RELATED SAMPLE: Pie Chart: Basic and Custom Legend

See the full list of **legend parameters** in API Reference.

Custom Legend

- Legend customization includes marker dimensions and arrangement, their alignment towards the chart, colors and border as well as custom text for the markers.
- Custom legends are vital with chart series for different graphs need to be explained for users could read the charts.
- Legend markers defined for chart series are used to **show** and **hide dedicated graphs** (lines, areas, bars..) by clicking on their signs. Invisible charts can be shown back by clicking thr greay 'inactive' marker sign.

Legend markers become "show-and-hide" buttons for all charts with series **except for stacked charts** (stackedArea, stackedBar, stackedBarH) for such charts presuppose series usage and don't inply visual comparison.




```
Custom legend

webix.ui({
    view:"chart",
    type:"",
    ...
    legend:{
        layout:"y",
        width: 110,
        align:"right",
        valign:"middle",
        marker:{
            width:15,
            radius:3
        },
        values:[
            {text:"company A",color:"#3399ff"},
            {text:"company B",color:"#66cc00"}]
    }
});
```

RELATED SAMPLE: Radar Chart: Lines

See the full list of **legend parameters** in API Reference.

Customizing Legend Markers

By default, legend markers are squares with rounded edges that feature 3-px border radius. To change this pattern, use the **marker** parameter of the into the legend property.

```
legend:{
    ...
    marker:{
        type:"round",
        width:15,
```

```
        radius:3
    },
}
```

RELATED SAMPLE: Radar Chart: Areas

'item' marker

Webix charts offer a possibility to set for a marker the same template that uses items of the related graph.




```
legend:{
    values:[{text:"Company A"},{text:"Company B"},{text:"Company C"}],
    ...
    marker:{
        type: "item",
        width: 18
    }
}
```

RELATED SAMPLE: Line Chart: Legend Markers

Setting different markers for different graphs



To set different markers for graphs you should:

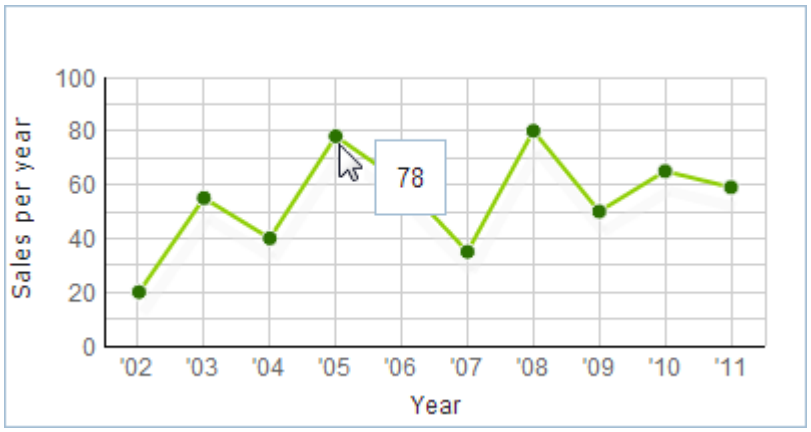
- Specify legend items manually, through the **values** parameter (don't use **template**);
- Use **markerType** property to assign the desired marker type to a graph.

```
legend:{
    layout:"y",
    align:"right",
    valign:"middle",
    width:100,
    values:[
        {text:"Company A",color:"#00ccff"},                    //the
default marker
        {text:"Company B",color:"#e9df40"},                    //the
default marker
        {text:"Average",color:"#b25151",  markerType: "item"} //the
redefined marker
    ]
}
```

RELATED SAMPLE: Bar Chart: Different Types in One Chart

Chart Tooltip

Tooltip is a graphical UI element that becomes visible when a user places a mouse pointer over the predefined area. It's used as an infotip giving additional information on user request.



Tooltip helps interprete chart values, giving presice numbers for bars as well as points in line, area, scatter and radar charts. Tooltip isn't normally used with pie charts as its values are written on pie pieces.

Chart tooltips are displayed in squares and include value of the the element you've places a cursor over.

Useful tip - define **eventRadius** for chart with a tooltip **bigger** than an item radius.

In essence, tooltip works under Event Handling principles: it appears and disappears on "on MouseOver" and "onMouseOut" events respectively, both of them being attached to an item in the chart. Items are usually small and users need to be very presice with cursor pointing. Setting **eventRadius** bigger doesn't change anything in the chart appearance but it simplifies tooltip triggering.

Tooltip Values

Tooltip values are defined on the base of datasource values. While the chart represents a numeric graphically, tooltip gives literal explanation of this value.

Let's take the left chart from above. It shows sales amounts per each year. Years are displayed over a horizontal scale while sales amount defines the height of the point on the vartical scale.

```
JSON Data

var dataset = [
    { id:1, sales:20, year:"02"},
    { id:2, sales:55, year:"03"},
    ...];
```

Tooltip here takes **#sales#** value and shows the exact sales amount as you place a cursor over the point.

```
webix.ui({
    view:"chart",
    type:"line",
    value:"#sales#", //settings for yAxis
    xAxis:{
            template:"'#year#"}, //settings for xAxis
    ....
    tooltip:{
           template:"#sales#" //tooltip
         }
    eventRadius: 10
})
```

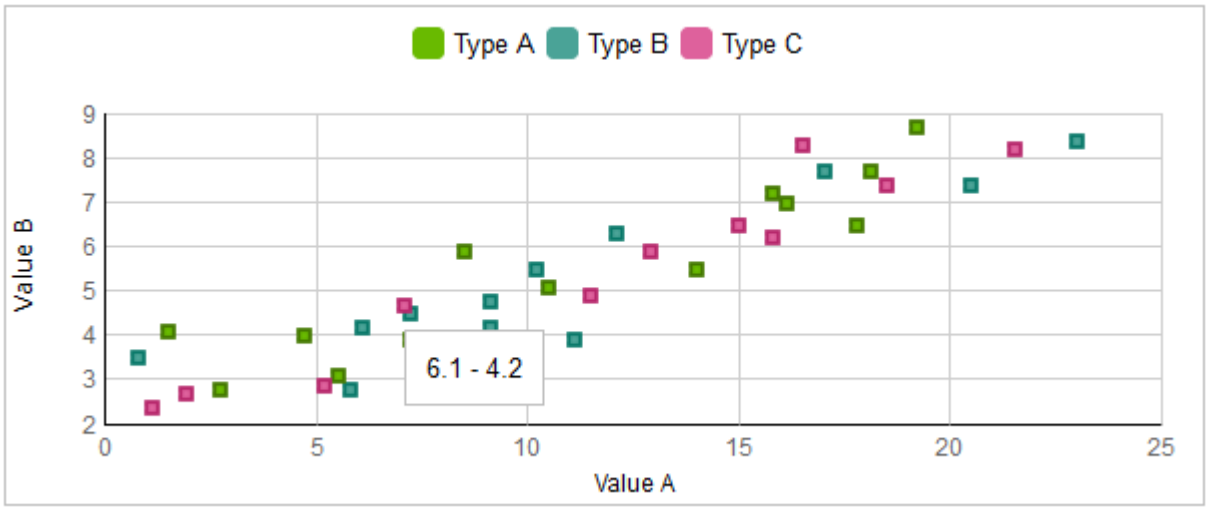
RELATED SAMPLE: Line Chart: Styling

Templates may include several properties of one and the same data object.

If you want to see year - sales tooltip for the chart above, change the tooltip template and include #year# to it:

```
tooltip:{
           template:"#year# - #sales#" //
         }
```

Such tooltips are especially convenient for Scatter charts where points aren't connected by lines and it's difficult to read exact value:



RELATED SAMPLE: Scatter Chart: Templates

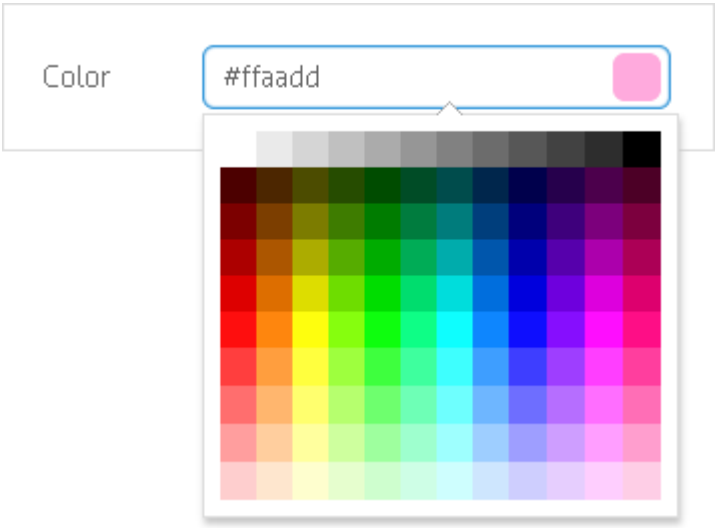
Colorboard

API Reference

- Methods, properties and events
- Samples

Overview

UI-related colorboard is a tool to get the hex code of a color that you are going to use for styling. By default, colors are presented in a rainbow-like manner. The detalization level you set manually.



Initialization

Colorboard is used in conjunction with a colorpicker control. In such a case, you needn't to initialize a colorboard since it will appear as soon as you click a control.

When used separately, colorboard is inited in the following way:

```
webix.ui({
        view:"colorboard",
        cols    :30,
        rows    :30,
        width   :400,
        height  :400
});
```

RELATED SAMPLE: Colorboard

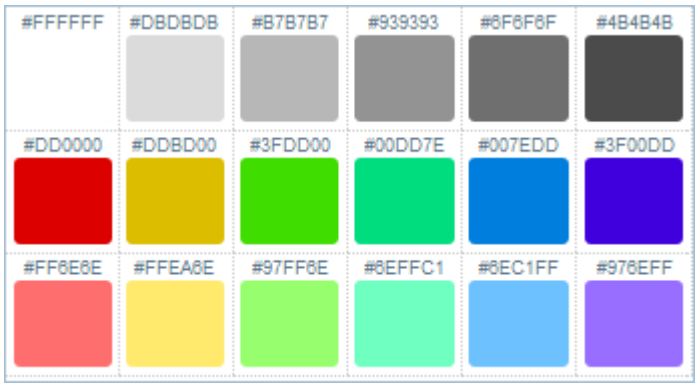
Comments:

- **width** and **height** denote the dimensions of a square with a color palette;
- **cols** and **rows** denote the number of colors in each axle - the bigger it is, the more detailed is the palette;
- gradient nature of the colorboard can be controlled via lightness values:
  - **minLightness** (0.15 by default) - start value of lightness (the darkest part);
  - **maxLightness** - (1 by default) - end value of lightness (the lightest part).

Working with Colorboard

You can customize your colorboard by:

- **setting the way of color presentation**




```
webix.ui({
    view:"colorboard",
    cols:7,
    rows:4,
    width:600,
    height:370,
    template:"<div class=\"value_cell\">" +
            "<div>{obj.val}</div>" +
            "<div class=\"color_block\"
style=\"background:{obj.val};\"></div>" +
            "</div>"
});
```

RELATED SAMPLE: Palette - Templates

- **defining your own color set**



```
var colors = [
    ["#C6D9F0", "#8DB3E2", "#548DD4"],
    ["#F2DCDB", "#E5B9B7", "#D99694"],
    ["#EBF1DD", "#D7E3BC", "#C3D69B"]
];

webix.ui({
    view:"colorboard",
    ...
    palette:colors
});
```

RELATED SAMPLE: Colorboard - Colors

Deriving Selected Color Value from the colorboard:

Normally, selected color is derived with the help of the **getValue()** method:

```
$$("colorboard1").getValue();
```

In addition, color values can be logged once **val**, **control** and **ev** parameters are passed into the logging function.

```
function doLog(val, control, ev){
        webix.message(this.config.id + " - " + val);
    }
```

Related Articles

- Making Templates
- Component Styling

Context

API Reference

- Methods, properties and events
- Samples

Overview

Ui-related context points to an element that will be shown on the page on right mouse click. The context looks like a popup window where you can place any HTML element or UI component.

Initialization

```
webix.ui({
        view:"context",
        width: 300,
        body:{ ....},
        padding:20,
        master:"areaA"
            });
```

RELATED SAMPLE: Context Menu: UI Components Inside

Comments:

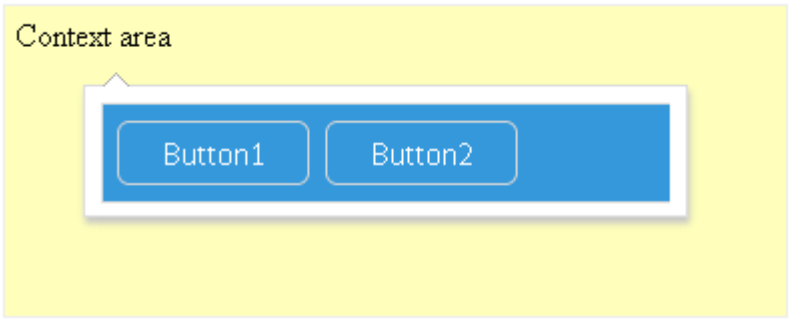
- **master** - the ID of an HTML container that will be shown on page loading. It is the area where the right mouse click will make context elements visible. This element is included into the *body* property;
- **body** - an element (ui component) that appears on right mouse click;
- **width** - width of UI component that apears on right mouse click;
- **padding** - offset from context border and its inner contents.

Context features **hidden** property set to *true* by default. It can be changed during component init, but this won't make much sets since context purpose is to be hidden and appear only on right click.

Working with Context

What can be a context element?

- **Any ui-related component**. For instance a two-button toolbar:




```
webix.ui({
    view:"context",
    body:{view: "toolbar", cols:[
            {view:"button", value:"Button1", width: 100},
            {view:"button", value:"Button2", width: 100}]
    },
    width: 300,
    master:"context_area"
});
```

RELATED SAMPLE: Context Menu: UI Components Inside

- **Any HTML element** referred to by the ID of the div containter it is placed in.



The container is specified in the body constructor as value of **content** property.

```
body:{
    content:"details"
}
```

Content will get elements that are inside the container.

```
<div id="details">
```

```
    Power of HTML5
    <hr>
    <img src='http://Webix.com/codebase/images/bg_touch.gif'>
</div>
```

RELATED SAMPLE: Context Menu: HTML Content

Related Articles

- Contextmenu
- Popup

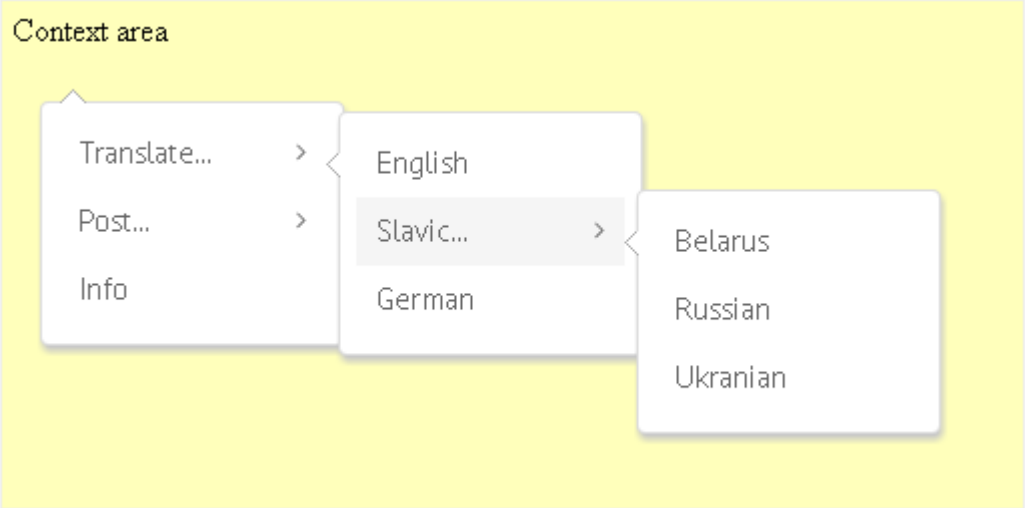
ContextMenu

API Reference

- Methods, properties and events
- Samples

Overview

Contextmenu is triggered by right mouseclick. Ui-related Contextmenu inherits from both window and list. It looks like ui-related menu and shares most of its properties, methods and events.



Initialization

1 . ContextMenu - List

```
webix.ui({
    view:"contextmenu",
    id:"my_menu",
    data:[
        "Add",
        "Rename",
        "Delete",
        { $template:"Separator" }
        ,"Info"
    ],
    master:"areaA" //  ID of a DIV container
});
```

RELATED SAMPLE: Context Menu

Data property includes list items in the window, **separator** stands for a line to divide window areas while **master** property defines the area where the component is initialized.

2 . ContextMenu - Menu

```
webix.ui({
    view:"contextmenu",
    id:"my_menu",
    data:[
        { value:"Translate...", submenu:[
                "English", "Slavic...", "German"
        ]},
        { ...},
        { ... }
    ]
    master:"areaB"
});
```

Here **data** includes menu constructor with some items having submenus.

RELATED SAMPLE: Context Menu

Working with ContextMenu

Getting an Item's Value

You can access any element of a contextmenu and its submenus and attach events to them with the help of the following patterns:

```
//for list contextmenu
$$("my_menu").attachEvent("onItemClick":function(id){
    webix.message(this.getItem(id).value);
});

//for "menu-like" contextmenu
$$("my_menu").attachEvent("onItemClick", function(id){
    var menu = this.getSubMenu(id);
    webix.message("Click: "+menu.getItem(id).value);
})
```

Linking a Contextmenu to a Component

A contextmenu can be initialized for any Webix component. A three-step process includes the following stages:

1) Firstly, you should include an **onContext** property into the component's construction:

```
webix.ui({
     view:"list",
     id:"list",
     //list config
     onContext:{} // the empty object
});
```

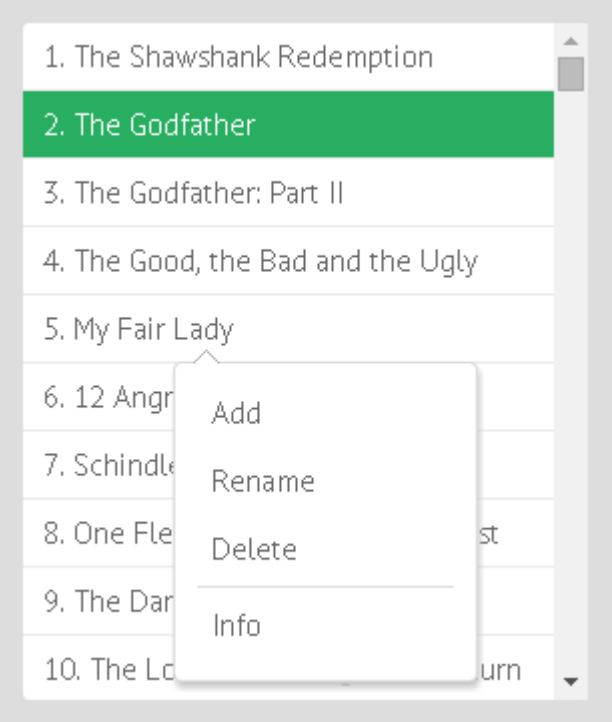
2) Secondly, you initialize a contextmenu in a separate within a separate **webix.ui** function:

```
webix.ui({
    view:"contextmenu",
    data:[....],
    ...
});
```

3) Thirdly, you link the contextmenu to the necessary component with the help of either

- an **attachTo()** function:

```
$$('contextmenu1').attachTo($$('list'));
```



RELATED SAMPLE: Context Menu: Attaching to List

- or contextmenu **master** property that points to the necessary component object:

```
webix.ui({
    view:"contextmenu",
    data:[....],
    master:$$("list")
```

```
});
```

Note You can link either to a **component object** or to its **HTML container** (like any other HTML on page). The difference lies within data components - to attach context menu to their items, you need to link to a component **object**.

A component HTML container is stored in its **$view** specific property:

```
$$("list").$view; //returns <div class="webix_view webix_list" ...></div>
```

```
Attaching contextmenu to list

//works with list items
$$('contextmenu1').attachTo($$('list1'));

//works with the whole list
$$('contextmenu1').attachTo($$('list1').$view);
```

It is the nice option to attach contentmenu to Webix controls, e.g. button or any HTML elements on the page:

```
$$('contextmenu1').attachTo($$('button1').$view);

$$('contextmenu1').attachTo(document.getElementById("myElement"));
```

Getting Properties of the Master Component

Like with context, the area upon which the context menu is called master area or (if you call it over a component or its item), **master component**.

Master component object is retrieved with the **getContext()** property.

```
webix.ui({
    view:"contextmenu"
    on:{
         onItemClick:function(id){
            var context = this.getContext();
            var list = context.obj; //list item objetc
            var listId = context.id; //id of teh clicked list item

            webix.message("List item:
<i>"+list.getItem(listId).title+"</i> <br/>
                          Context menu item:
<i>"+this.getItem(id).value+"</i>");
```

RELATED SAMPLE: Context Menu: Attaching to List

Related Articles

- List

- Menu
- Event Handling
- Context

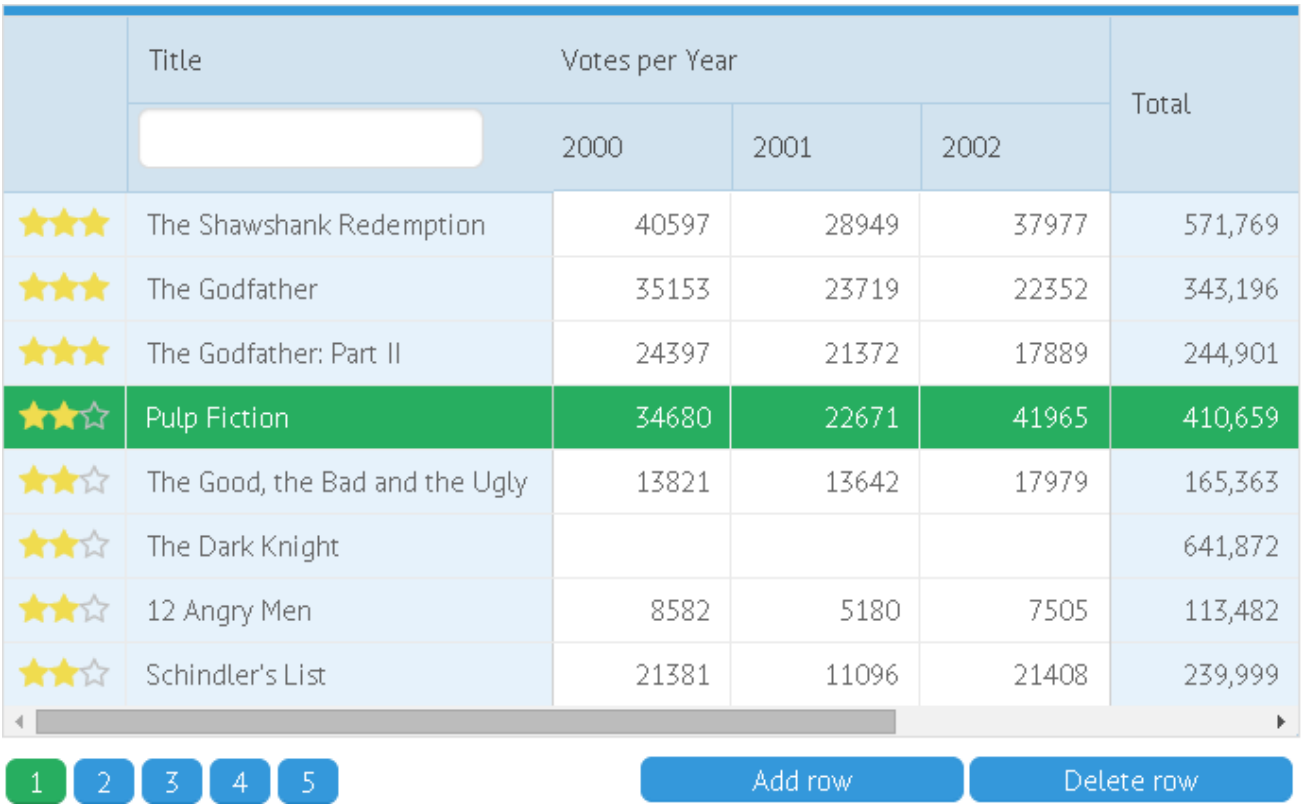
DataTable

API Reference

- Methods, properties and events
- Samples

Overview

DataTable is a pure JavaScript component that provides professional look-and-feel and convenient programming model for displaying data in a scrollable and sortable table. It's a powerful yet easy-to-use control
that supports different data sources (xml, json, csv, jsarray, html tables) and has various possibilities (filtering, sorting, pagination, resizing, styling, copying etc.) still being very fast handling thousands of records. Read more...



Guides

Configuration ◦ General columns settings ◦ Creating index columns ◦ Headers and footers ◦ Colspans and rowspans ◦ Data loading ◦ Internationalization ◦ Sizing and resizing ◦ External configuration ◦ Adding Active Elements to Datatable Cells	Teaches you how to define the common structure and content of DataTable: columns, headers, dimensions and data inside it.
Operations with data ◦ Copy-pasting ◦ Editing ◦ Filtering ◦ Math operations ◦ Paging ◦ Selection ◦ Sorting	Discusses operations you can perform on the data in DataTable: filtering and sorting techniques, inline editing, work with the clipboard, etc.
Look-and-feel ◦ Cell templates ◦ Number and date formatting ◦ Styling ◦ Split mode or 'frozen columns'	Focuses on how to fine-tune the DataTable appearance: applying templates to the data, 'freezing' columns, redefining the default component CSS classes, etc.
Integration ◦ DHTMLX ◦ jQuery ◦ Server-side solutions	Says about libraries and platforms you can integrate DataTable with.
Export ◦ Excel ◦ PDF	Describes the formats and procedures you've got to import/export the DataTable's data.

Overview

DataTable is a pure JavaScript component that provides professional look-and-feel and convenient programming model for displaying data in a scrollable and sortable table. It's a powerful yet easy-to-use control that supports different data sources (xml, json, csv, jsarray, html tables) and has various possibilities (filtering, sorting, pagination, resizing, styling, copying etc.) still being very fast handling thousands of records (dynamic loading).

Main features

1. Supported data formats: xml, json, jsarray, csv
   Developers can use their favourite formats to specify data. All popular data formats are supported.

2. Initializing from an HTML table
   Any HTML table can be turned into DataTable by a single command.

3. Linkable tables
   This feature allows developers to create multiple DataTable instances on a page and easily implement various relationships through APIs.

4. Simple integration with server side
   Simple integration with any server side solution, such as PHP, .NET, Java.

5. Full API support
   A full range of API with the detailed descriptions is available.

6. Support for internationalization
   Support for internationalization allows applications be translated and modified into local versions for thousands of languages.

7. Flexible columns
   Each column in DataTable is configured individually: width, height, color etc.

8. Complex headers and footers
   The header(footer) can be presented as a single or multi line (the number of sublevels is unlimited) and contain almost any content: text, an image, an html input, a link, a built-in filter, colspans, rowspans.

9. Different selection modes
   You can set 7 different modes specifying how data will be selected by: cell, row, column, multi cell, multi row, multi column, block.

10. Styling with standard css
    You can use CSS in conjunction with DataTable content to archieve the desired look-and-feel. Css can be applied to the whole table or to a specific column, row, cell.

11. Resizable columns and rows
    Columns width or rows height can be changed by draging the related border.

12. Copying to/from clipboard
    By common CTRL+C/CTRL+V keyboard shortcuts data can be copied to some other application for further processing or moved within one or several DataTables.

13. On-the-fly filtering

    The library offers abundant possibilities of filtering, starting from built-in filters (text and select) and ending with fully custom filtering logic.

14. Sorting a column by clicking the header with data type detection

    This feature allows users to sort data (according to one of 4 predefined sorting types: integer, date, string and string_strict) by a single clicking the header. Custom sorting rule can be defined as well.

15. Variable length and localizable pagination

    One or several pagers can be added to the app to display data page by page. Data will not be retrieved from server side until the related page is opened. The number of rows per page, templates for navigation controls are set up.

16. Fixed one or more left (right) columns when table is scrolled horizontally

    The library provides the 'split' mode that allows you to have the table separated into two parts: 'fixed' and 'scrolled'.

17. Possibility to show or hide columns

    Any column in the table can be hidden (showed) programmatically.

18. Math formulas for cells

    Developers have an opportunity to load data that are stored as mathematical expressions. DataTable parses, evalutes such expressions and presents result in the related cells.

19. Scrolling options for table viewport

    When your viewport is less than a page you can enable vertical(horizontal) scrolling to be able to see the hidden info. You also have the possibility to enable scrolling by whole rows.

20. Possibility to set different heights for rows and different widths for columns

    You can set different widths(heights) for columns(rows). Set values will be saved regardless of any other enabled size parameters.

21. Export to PDF, Excel

    DataTable can be exported to PDF, Excel for further processing. Besides, you can define how the table will look in the document after it is exported by configuring various server side options.

22. Integration with the Webix 3 library

    The library can be integrated with three Webix 3 components: WebixLayout, WebixGrid and WebixMenu.

23. Integration with jQuery

    DataTable is integrated with jQuery that allows you to follow the main mantra of the jQuery library while developing - running with the least amount of configuration.

24. Dynamically fetching the rows from the server when the table is scrolled up and down

    Working with large datasets, developer can load data dynamically to reduce complexity and improve performance. Polling interval and the number of loaded records are controlled.

25. Callbacks for all events: load, sorting, etc.

    Numerous events lets you handle any action or proceeding that can happen in the app.

Getting started

To make a great app you need to learn different things but if you're new to DataTable you might be wondering where to begin.

Some platforms (Java, .Net, PHP) get ready-to-use solutions for data loading, saving, exporting.

For understanding the basic concepts and getting the desired information in the most convenient way we offer you to follow this road map:

Tutorial - gives essential notes you should know to create the first DataTable-based application.

Detailed guidelines - the main chapter of the documentation covering the developer-oriented tasks in details.

- Configuration - explains how to configure a table: to specify and size required columns, set headers(footers) for them, populate the table with data and etc.
- Look and Feel - tells you how to fine-tune the appearance of your table and data in it with the help of css classes, templates and special modes.
- Operations with data - covers techniques for performing general operations under data in DataTable, such as sorting, filtering, copy-pasting and selection.
- Operations with cells - describes basic operations you surely will use while developing: getting/setting the value of a cell.
- Integration - says which libraries and how can be integrated with DataTable.
- Data export to PDF, Excel - shows the process of exporting data from DataTable to PDF and Excel documents.
- Saving data on server - describes server-side work.

Events handling - says how to use events: to attach and detach, which events can be handled and how.

API Reference - provides reference information you need for coding: all the available methods, parameters and events of the library.

Quick start with DataTable

This tutorial walks you through creating a basic application containing DataTable. As you follow the steps, you learn how to initialize DataTable, customize and configure its appearance, load data to it. The final code of the tutorial can be used as the start point while creating applications using DataTable.

RELATED SAMPLE: Quick start with DataTable

A new HTML file and related code files

To start, create a new HTML file and include the DataTable code files in it.

Required code files are:

- *webix.css*
- *webix.js*

```
A basic html page with the included DataTable code files

<html>
<head>
    <title>Quick start with DataTable</title>
    <script src="../codebase/webix.js" type="text/javascript"
charset="utf-8"></script>
    <link rel="stylesheet" href="../codebase/webix.css" type="text/css"
charset="utf-8">
</head>
<body>
    <script>
        //here you will place your JavaScript code
</script>
</body>
</html>
```

Including Webix files via Package Managers

OR, as an **alternative** to downloading, you can install Webix library via such package managers as **NuGet** and **Bower**.

Installing Webix requires by means of these tools requires executing just a line of code inside them (you don't need to download anything):

```
NuGet

nuget install Webix

//If you use Microsoft Visual Studio, execute this from Package Manager
```

```
Console
install-package Webix
```

```
Bower

bower install webix
```

Placing DataTable on a page

On the next step, you should create a *div* container where you then place your DataTable.

```
<div id="box" style="width:400px;height:200px;"></div>
```

Object constructor

After the place for DataTable has been prepared, move to its initialization.

To create a new DataTable instance you need to use the next constructor:

```
dtable = new webix.ui({ view:"datatable" })
```

Let's specify 2 mandatory parameters inside the constructor:

container - the HTML container for DataTable. We created it on the previous step.

columns - the array of objects, each of which specifies an individual column. Each object must at least has the **id** attribute set.

It addition to **id**, we will specify 2 often used column attributes: **header** (the header of a column) and **width** (the width of a column).

```
Basic DataTable configuration

dtable = new webix.ui({
        container:"box",
        view:"datatable",
        columns:[
            { id:"title",   header:"Film title",   width:200},
            { id:"year",    header:"Release year", width:80},
            { id:"votes",   header:"Votes",        width:100}
        ]
});
```

General settings

On the previous step, we have defined primary settings for DataTable. For more available configuration, read other articles in this documentation, such as:

- General columns configuration - brief announcement of available column settings;
- Selection - describes different types of selection (cell, column, row etc.) and their use;
- Sorting and Filtration - says about means for sorting and filtering.

Loading data

There are various ways to load data into DataTable (read about them in chapter Data loading).
We will use the easiest of the ways and specify data directly in the constuctor with the help of the data parameter:

```
dtable = new webix.ui({
        container:"box",
        view:"datatable",
        columns:[
            { id:"title",   header:"Film title",    width:200},
            { id:"year",    header:"Release year" , width:80},
            { id:"votes",   header:"Votes",         width:100}
        ],
        data:[
            { id:1, title:"The Shawshank Redemption", year:1994,
votes:678790},
            { id:2, title:"The Godfather",           year:1972,
votes:511495}
        ]
});
```

Congratulations! Now you can run your app and it'll produce DataTable shown at the very beginning.

RELATED SAMPLE: Quick start with DataTable

What's next

What can we suggest you to do next?

- Check the code sample of the mentioned above example (use the link above);
- Read about possible data formats and how you can get data from database;
- Find out about available columns configuration;
- Check other supplied samples.

Getting/setting the cell value

API of DataTable allows you to retrieve existing values of cells or set new ones.

Getting the value

To get the record object, you should use method item():

```
Getting the record object
```

```
grid = webix.ui({
    view:"datatable",
    data:[ {id:"row1", title:"The Shawshank Redemption", year:"1994"} ],
    ...
});
...

//'record' variable will contain an object of the related DataTable record
var record = grid.getItem("row1");

//you can access data members directly
var title = record.title;
```

Setting the value

To change the current value of a cell, you should use the following technique:

```
Setting the cell value
```

```
record = grid.getItem(row_id);
record[column_name] = new_value;

grid.refresh();
//or
grid.updateItem(row_id, record);
```

Methods refresh() and update() lead to one and the same result and don't have any specificity.

Getting values of cells range

When you work with a block of cells you can use 2 'special' methods allowing you to get values of several cells and perform some action on them:

- mapCells - gets the current values of the specified cells (called for each specified cell by turn).

```
Reversing text in the specified cells range
```

```
grid.mapCells(start_row,start_col,numRows,numCols,
function(value, row_id, column_id, row_ind, col_ind){
        return value.toString().split("").reverse().join("");
});
```

RELATED SAMPLE: Mapping cells

- mapSelection - gets the current values of the selected cells (called for each selected cell by turn).

```
Reversing text in all selected cells

grid.mapSelection(function(value){
        return value.toString().split("").reverse().join("");
});
```

RELATED SAMPLE: Mapping selection

For more on the mentioned and other related methods, see API Reference.

Configuration

- Columns Configuration
- Creating Index Columns
- Headers and Footers
- Datatable Rowspans and Colspans
- Data loading
- Sizing and Resizing
- Internationalization
- External configuration

Columns Configuration

Columns configuration is specified by parameter columns and all settings are managed on its level.

```
grid = new webix.ui({
    view:"datatable",
    columns:[{...}, {...}],
    ...
});
```

With the help of the columns parameter you can:

- specify headers/footers;
- set widths of columns;
- add built-in filter;
- set built-in sorting;
- set the text alignment in a column;
- set the format of data presentation;
- specify data of columns through math formulas;
- specify datasource for the column;
- define templates for data presentation;
- define individual css class for any column;
- hide/show a column;
- hide/show columns in groups;
- add/delete coulmns dynamically;
- group columns;
- define colspans and rowspans;

Automatic Column Configuration

Normally, you need to specify an array of column configurations for the datatable, which allows for setting various parameters needed by this or that datatable separately:

```
view:"datatable",
columns:[
    {id:"id", header:"Id", width:50, sort:"int"},
    {id:"title", header:[ "Film title",{content:"textFilter"}]
]
```

At the same time, you can switch to **data-based** column configuration provided by **autoconfig** property:

```
view:"datatable",
autoconfig:true
```

In this case *columns* array is no longer needed. Datatable will analyse the dataset passed to it and build columns automatically. The columns will have **default values** (no sorting or filtering, width of 50px, etc.)

Headers/footers

Enabled by parameters header, footer (by default, headers are enabled).

Specified by attributes **header**, **footer** and can be both *string* and *object*.

In case of *string*, you can specify just the label.

Object definition allows you to specify more complex column presentation. As an *object*, header/footer can contain the following properties:

- **text** - the text label of a column;
- **colspan** - the number of columns a cell should span;
- **rowspan** - the number of rows a cell should span;
- **content** - the built-in filter of a column (textFilter or selectFilter).

```
Defining header and footer for a specific column

//simple string
grid = new webix.ui({
        view:"datatable",
        ...
        footer:true,
        columns:[
            { id:"col1", header:"Column 1", footer:"Footer 1"},
            {...}
        ]
})
//object
grid = new webix.ui({
        view:"datatable",
        ...
        footer:true,
        columns:[{
            id:"col1",
            header:[{ text:"Column1", colspan:"2"}, {
content:"selectFilter" }],
            footer:[{ text:"Total", rowspan:"2"}, { text:" " } ]
        },
        ...]
})
```

RELATED SAMPLE: Standard header(footer) implementation

For more details, read the Headers and footers article.

Widths of columns

To set widths of columns you should use attribute **width**.

```
Setting the width of a column

columns:[
    { id:"col1",  width:200 }
]
```

RELATED SAMPLE: Sizing through the 'width', 'height' parameters

For more details, read the Sizing and resizing article.

Built-in filters

The header or footer of a column can contain filter. The following types of filters are available:

- *textFilter* - text filter. Retrieves values which contain mask defined through text field;
- *selectFilter* - select filter. Retrieves values which contain mask defined through dropdown list of possible values.
- and 6 more..

Filter is set by property **content** of the **header** attribute.

```
Adding filter to the header of a specific column

columns:[
    { id:"col1", header:[{ text:"Column 1", colspan:"2"}, {
content:"selectFilter" }]},
    {...}
]
```

RELATED SAMPLE: Filtering. Built-in Text Filter, Select Filter

For more details, read the Filtering article.

Built-in sorting

You are allowed to sort data in columns. The way of sorting depends on the column sorting type. There are 3 predefined sorting types:

- *int*

- *string*
- *string_strict*

Last one is a case-sensitive string sorting

The sorting is specified by attribute **sort**.

```
Activating sorting for a specific column

columns:[
    { id:"col1", sort:"string" },
    {...}
]
```

RELATED SAMPLE: Sorting. Using built-in means

For more details, read the Sorting article.

Text alignment

To set the text alignment in a column you should use attribute **css**:

```
Setting the text alignment in a column

<style>
.myStyle{
    text-align:right;
}
</style>

columns:[
    { id:"col1" css:"myStyle"},//in a separate css class
    { id:"col2", css:{'text-align':'right'} //directly in the attribute
]
```

Read more about using the **css** attribute in the Styling article.

Data formats

Formatting is applied to date and number values and defined for each column separately. To set some format for a column, use attribute **format**.

```
Setting the format for a specific column

columns:[
    //data will be formatted according to the current locale
    { id:"col2", format:function(value){ return
webix.ui.numberFormat(value)};}
]
```

RELATED SAMPLE: Using date templates

For more details, read the Number and date formatting article.

Math in columns

You can write simple math formulas to specify values in column cells. Formulas can be set for a whole column or for a single cell.

```
Using formulas for setting values in the whole column

columns:[
    { id:"col1", math:"[row_id,col_name] + [row_id,col_name]"}
]
```

RELATED SAMPLE: Using formulas

For more details, read the Using formulas article.

External Datasource for the Column

The **collection** property of the column allows syncing column data with that of a dataCollection object or any data management component.

```
{   id:"order-1",
    view:"datatable",
    columns:[
        { id:"product", header:"Product", collection:"dash-pro" },
        { id:"quantity", header:"Quatity",  ... },
        { id:"price", header:"Unit price",  ... }
    ],
    dataFeed:"data/orderproducts.php", //main datasource
};
```

Collection for the first column points to other **datatable** with **"dash-pro" ID**.

```
{   view:"datatable", id:"dash-pro",
    columns:[
        { id:"name", header:"Product name", ..  },
        {..//other columns}
            ],
   url:"data/products.php",
   save:"connector->data/products.php" //dataprocessor
};
```

With the help of this property you can also sync the datatable with various dataCollections.

Templates

Through attribute **template** you can set a template of cells presentation corresponding to your needs.

```
Using templates for configuration data of a column

columns:[
    { id:"col4", template:function(obj){return
obj.col1+obj.col2*2+obj.col3*3;}}
]
```

RELATED SAMPLE: Using string templates

For more details, read the Data templates article.

Styling

Many aspects of column can be customized to achieve the desired look-and-feel. Column style can be adjusted in a css class and applied through attribute **css**.

```
Applying a css class to a specific column

columns:[
    { id:"col1", css:"someStyle"}
]
```

For more details, read the Styling article.

Hiding/showing columns

Initially, all the datatable columns are visible by default unless a specific **hidden** property is used in its configuration:

```
var grid = new webix.ui({
    view:"datatable",
    columns:[
        { id:"col1", header:"Title", hidden:true}, //this column is hidden
initially
        { id:"col2", header:"Rating"} //this column is visible
   ]
})
```

Through the hideColumn() and showColumn() methods you can manipulate visibility of columns dynamically.

```
Hiding a column

grid.hideColumn("col2");
```

Setting hidden/visible columns in groups

Datatable API allows for setting the initially visible group of columns (**visibleBatch**) as well as show and hide any chosen column group defined by **batch** property:

```
grida = new webix.ui({
  view:"datatable",
  visibleBatch:1,
  columns:[
    { id:"id",  header:"#", css:"rank",  batch:2,   width:50},
    { id:"title", header:"Film title", fillspace:true },
    { id:"year",  batch:1,  header:"Released" , width:80},
    { id:"category", header:"Category", batch:1},
    { id:"votes", batch:3, header:"Votes",  width:100},
    { id:"rating", batch:3, header:"Rating", width:100},
    { id:"rank", header:"Rank", css:"rank",  batch:2,width:50}
  ]
});
```

Any column group can be shown with the showColumnBatch method that takes column **batch** value as parameter:

```
//show show id, rank
grida.showColumnBatch(2);
//show votes, rating
grida.showColumnBatch(3);
```

RELATED SAMPLE: Column Batches

Adding/deleting columns dynamically

Since datatable **columns** is an array of JSON objects, you can treat it like any JavaScript array. At any moment you can add elements (columns) to this array and delete elements from it - and then repaint the datatable with the new configuration.

```
Replacing current columns with new ones

grid.config.columns = [..new collection of columns..];
grid.refreshColumns();
```

In addition, Webix offers its own API for working with arrays:

- **webix.toArray()** - converts an array to "Webix" array;
- **array.insertAt()** - inserts an element to array into the specific position;
- **array.removeAt()** - removes an element from the array at the specific position.

```
Adding/Removing Separate Columns
```

```
var columns = webix.toArray(grid.config.columns);
//adding
columns.insertAt({
    id:"c"+webix.uid(),
    header:"New column"
},2);

//deleting
columns.removeAt(2);

//refreshing
grid.refreshColumns();
```

RELATED SAMPLE: Adding/removing columns

Grouping columns



The functionality is available in **Webix Pro** edition only.

Column batches can be used to organize groups of columns that are shown/hidden by clicking on the header of an aggregating column:

Film title	⊞ 2008	⊟ 2009		
	January	January	February	March
The Shawshank Redemption	825	6566	9189	5084
The Godfather	4744	8194	4167	9152
The Godfather: Part II	3239	7036	786	1391
The Good, the Bad and the Uc	6387	6991	7947	3580
Pulp fiction	4124	2714	6146	9972
12 Angry Men	5166	1567	6242	184

For these needs a multiple line header is used. The first line of an aggregating column is defined via an object with the following properties:

- **content** (string) - content-forming property, here: **columnGroup**. The property is also used to define header filters;
- **batch** (string, number) - ID of the group;

- **groupText** (string) - title of the group (it is shown when the group is in a closed state);
- **closed** (boolean) - the initial state of the group, *false* by default;
- **colspan** (number) - the number of columns in the group, including the aggregating one.

Also, each column that belongs to some group, should be given a group ID defined by **batch** property:

```
columns:[
    //aggregating column
    { id:"2008_1", header:[
      { content:"columnGroup", closed:true, batch:"2008",
groupText:"2008", colspan:12},
      "January"
    ]},
    //other columns from the group
    { id:"2008_2",  batch:"2008", header:[null, "February"] },
    { id:"2008_3",  batch:"2008", header:[null, "March"] },
    { id:"2008_4",  batch:"2008", header:[null, "April"] },
    ...
]
```

RELATED SAMPLE: Grouped columns in DataTable

Read more about colspans and rowspans in Webix datatable header.

Setting Colspans/Rowspans



The feature is available in **Webix Pro** edition only.

Datatable allows defining colspans and rowspans by **span** configuration provided with the dataset within the **data** property of the grid.

Each rowspan/colspan definition consists of the **row id**, **column id**, **width** and **height** of the span, **value** to display and **css** class to apply:

```
grida = new webix.ui({
    view:"datatable",
    columns:[...],
    data:{
        data:grid_data,
        spans:[
            [1, "title", 3]
        ]
    }
});
```

RELATED SAMPLE: Colspans in the DataTable

To learn more on this topic, please visit the Datatable Rowspans and Colspans article.

Creating Index Columns

In this chapter you will learn how to create index columns in DataTable.

There are 2 types of index columns:

- Static (rows indices aren't changed whether you move, delete rows etc.)
- Dynamic (rows indices can vary depending on the current position of rows)

Static index columns

To create a static index column in your datatable, use the following technique:

Initial table

	Film title	Year	Votes
1	The Shawshar	1994	678790
2	The Godfather	1972	511495
3	The Godfather	1974	319352
4	The Good, the	1966	213030
5	Pulp fiction	1994	533848
6	12 Angry Men	1957	164558

Sorted table (the rows preserved their indices)

	Film title	Year ^	Votes
6	12 Angry Men	1957	164558
4	The Good, the	1966	213030
2	The Godfather	1972	511495
3	The Godfather	1974	319352
1	The Shawshar	1994	678790
5	Pulp fiction	1994	533848

```
Specifying static index column in DataTable

dtable = new webix.ui({
    view:"datatable",
    columns:[
        { id:"index",   header:"",           sort:"int"},
        { id:"title",   header:"Film title", sort:"string"},
        { id:"year",    header:"Year",       sort:"int"}
    ],
    scheme:{
        $init:function(obj){ obj.index = this.count(); }
    },
    ...
});
```

RELATED SAMPLE: Index column

Let's consider what we do in the code above:

In the columns parameter we assign an individual column to serve as the index entry.

Then, we specify the scheme property with the **$init** key inside. The **$init** key runs when data is being loaded to the component initially and when data is being reloaded for new elements. The key is called for each data item, that's why we can use it as an iterator to assign indices to rows.

Read more about the key in the related article - 'Defining Data Scheme. $init key'.

Dynamic index columns

To create a dynamic index column in your datatable, use the following technique:

Initial table

	Film title	Year	Votes
1	The Shawshar	1994	678790
2	The Godfather	1972	511495
3	The Godfather	1974	319352
4	The Good, the	1966	213030
5	Pulp fiction	1994	533848
6	12 Angry Men	1957	164558

Sorted table (the rows took new indices)

	Film title	Year ^	Votes
1	12 Angry Men	1957	164558
2	The Good, the	1966	213030
3	The Godfather	1972	511495
4	The Godfather	1974	319352
5	The Shawshar	1994	678790
6	Pulp fiction	1994	533848

Specifying dynamic index column in DataTable

```
dtable = new webix.ui({
    view:"datatable",
    id:"mytable",
    columns:[
        { id:"index",   header:"",           sort:"int"},
        { id:"title",   header:"Film title", sort:"string"},
        { id:"year",    header:"Year",       sort:"int"}
    ],
    on:{
        "data->onStoreUpdated":function(){
            this.data.each(function(obj, i){
                obj.index = i+1;
            })
        }
    },
    ...
});
```

RELATED SAMPLE: Index column

Let's consider what we do in the code above:

In the columns parameter we assign an individual column to serve as the index entry.

With the on parameter we attach a handler to the onStoreLoad event.

"*data->onStoreUpdated*" is equal to *$$('mytable').data.attachEvent('onStoreUpdated', function(){...})*

The onStoreLoad event fires after data is changed in the data store.

The each method allows us to iterate through the collection of data items.

Headers and Footers

Headers and footers in DataTable are enabled by the header, footer parameters and configured by attributes **header**, **footer** in the columns parameter.

Header can be presented as:

- A single line header;
- A multiline header;
- A single line (or multiline) header containing colspans (rowspans), built-in filters;

Footer can be presented as:

- Single-line footer;
- Multiline footer;
- A single line (or multiline) footer containing colspans (rowspans), built-in counter;

Additionally, both header and footer can be rotated and take custom height.

Single line header

To define a simple (single line) header you should specify it as a string.

Do smth	IMDb	Year	Link
1	The Shawshank Redemption	1994	678790
2	The Godfather	1972	511495
3	The Godfather: Part II	1974	319352

In this case you can put the following to the header:

- *text labels*;
- *images*;
- *html inputs*;
- *links*.

```
Defining a single line header

columns:[
    //header as a button
    { id:"col1", header:"<input type='button' value='Do something'
class='dummy_button'>"},
    // header as an image
    { id:"col2", header:"<img src='../common/imdb.png'>"},
```

```
    //header as a text label
    { id:"col1", header:"Year"},
    // header as a link
    { id:"col4", header:"<a href='http://google.com'>Link</a>"}
]
```

RELATED SAMPLE: Different types of content in the header

Multiline header

You should use 'array' definition to split header into several subheaders (each value of the array specifies a single subheader).

#	Film title	Year	Votes
	Second line		
1	The Shawshank Redemption	1994	678790
2	The Godfather	1972	511495
3	The Godfather: Part II	1974	319352

```
Defining a multiline header

columns:[
    { id:"rank",  header:["#", ""]},
    { id:"title", header:["Film title", "Second line"]},
    { id:"year",  header:["Year", ""]},
    { id:"votes", header:["Votes", ""]}
]
```

Note, you can specify different number of subheaders for different columns (number of subheaders is equal to number of values in array).

RELATED SAMPLE: Multiline header

Complex header containing colspans (rowspans), built-in filters

To specify colspan(rowspan) in the header or put filter into it, you should specify the header as an object or array of objects.

When the header is presented as an object, it has the following properties:

text - the text label of a column;

colspan - the number of columns a cell should span;

rowspan - the number of rows a cell should span;

content - functional header content, built-in filters or group definition;

rotate - if *true*, switches header to rotated state;

height - defines a custom height for a header line.

Colspan

Film title	Year	Votes
Second line		
The Shawshank Redemption	1994	678790
The Godfather	1972	511495
The Godfather: Part II	1974	319352

To create a cell that will span across 2 (or more) columns you should use the following technique:

```
Using colspans

columns:[
    { id:"title", header:["Film title", {text:"Subheader", colspan:3}]},
    { id:"year",  header:["Year", ""]},
    { id:"votes", header:["Votes", ""]}
]
```

RELATED SAMPLE: Colspans and rowspans in the header

Rowspan

Film title	Year	Votes
The Shawshank Redemption	1994	678790
The Godfather	1972	511495
The Godfather: Part II	1974	319352

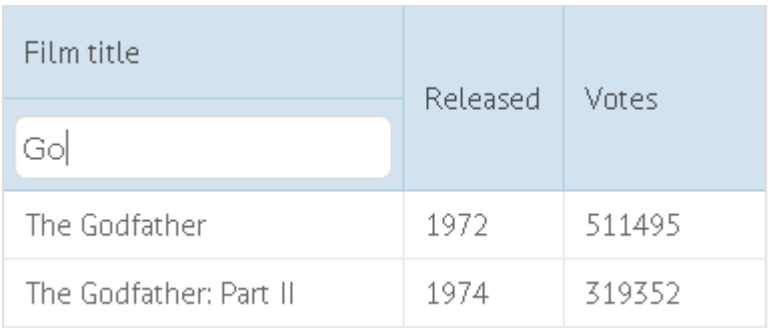
To create a cell that will span across 2 (or more) rows you should use the following technique:

```
Using rowspans

columns:[
    { id:"title", header:[{ text:"Film title", rowspan:2}]},
    { id:"year",  header:["Year", ""]},
    { id:"votes", header:["Votes", ""]}
]
```

RELATED SAMPLE: Colspans and rowspans in the header

Built-in filters



To put filter into the header, you should use the following technique:

```
Using filters in the header

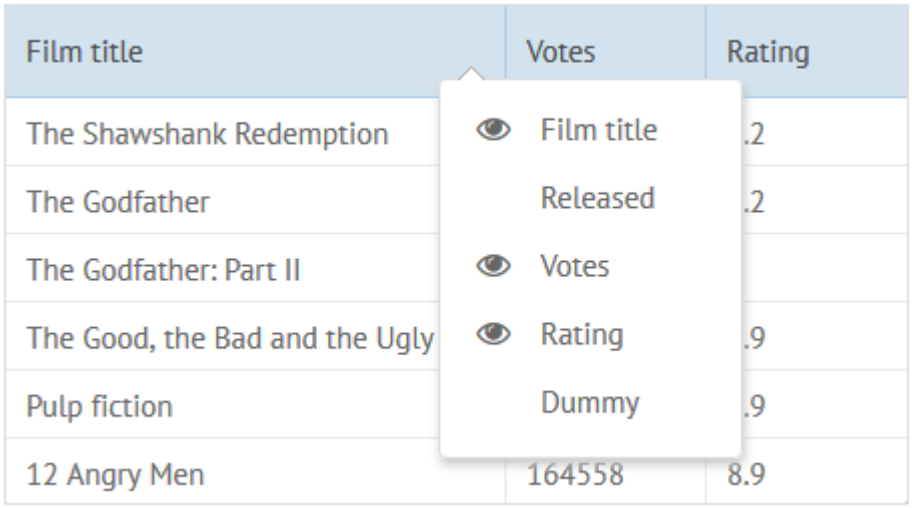
columns:[
    { id:"title", header:["Film title",{ content:"textFilter"}]},
    { id:"year",  header:"Released"},
    { id:"votes", header:"Votes"}
]
```

RELATED SAMPLE: Filters in the header

More information about built-in and custom filters is in the separate article.

Header Menu

> The functionality is available in **Webix Pro** edition only.



Built-in header menu allows end users to control visibility of the columns by several mouse clicks. The definition is as well simple:

```
{ view:"datatable", columns:[...], headermenu:true}
```

RELATED SAMPLE: Header menu in the DataTable

Headermenu can be customized. For more details, go to the Datatable Header Menu article.

Single line footer

By default, headers are enabled whereas footers are disabled.

So, before using footers - enable them.

```
Enabling footers in DataTable

grid = new webix.ui({
    view:"datatable",
    ...
    footer:true
});
```

Then, use all the same logic as for headers:

```
Using footers

columns:[
    //footer as a button
    { id:"col1", footer:"<input type='button' value='Do something'
class='dummy_button'>"},
    // footer as an image
    { id:"col2", footer:"<img src='../common/imdb.png'>"},
    //footer as a text label
    { id:"col1", footer:"Year"},
    // footer as a link
    { id:"col4", footer:"<a href='http://google.com'>Link</a>"}
]
```

RELATED SAMPLE: Standard header(footer) implementation

Multiline footer

Same as with header, multiline footer is comprised of an array of texts for each footer line.

```
Defining a multiline footer

columns:[
    { id:"rank",  footer:["Total", "Second line"]},
    { id:"title", footer:["", ""]},
    { id:"year",  footer:["", ""]},
    { id:"votes", footer:["12547", ""]}
```

```
]
```

Complex footer containing colspans/rowspans

Colspans and rowspans are used under the same logic as with headers. Footer line should be presented as an object that may contain:

text - the text label of a footer;

colspan - the number of columns a cell should span;

rowspan - the number of rows a cell should span;

content - the built-in functor of a column (summColumn);

rotate - if *true*, switches footer to rotated state;

height - defines height for a footer line.

```
Colspans in footer

columns:[
    { id:"rank",  footer:{text:"Total", colspan:3}},
    { id:"title", ...},
    { id:"year", ...},
    { id:"votes", footer:"12547"}
]
```

Sum counter in the footer

The library provides the **summColumn** counter that can be used to get the total value in the footer of a column.

To put the counter into the footer, you should use the following technique:

```
Using the sum counter in the footer

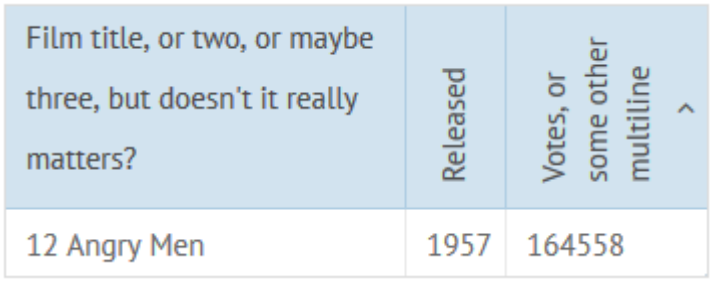
columns:[
    { id:"rank",  header:"", footer:{text:"Total:", colspan:3}},
    { id:"title", header:"Film title"},
    { id:"year",  header:"Released"},
    { id:"votes", header:"Votes", footer:{content:"summColumn"}}
]
```

RELATED SAMPLE: Counter in the footer

Rotated header/footer



Note that this feature is available in **Webix Pro** edition only.



To define rotated state for the header or footer you should specify it as an object (or array of objects in case of multiline header/footer).

Header height will be adjusted to text size.

```
columns:[
    {   id:"year",
        header:{ text:"Released", rotate:true },
        footer:{ text:"Footer", rotate:true}
    }
]
```

RELATED SAMPLE: Header rotate

Custom Height for Header and Footer

To set a custom height, present footer line as an object and use the **height** property:

```
columns:[
    {   id:"title",
        header:{ text:"Film title", height:25 },
        footer:{ text:"80px footer", height:80}
    }
]
```

RELATED SAMPLE: Custom height for header lines

Custom Header and Footer Content

All content elements that can be integrated to the datatable header or footer are stored in a **webix.ui.datafilter** object and feature the same configuration pattern.

There're two mandatory methods **refresh()** that provides logic and **render()** that draws an element. Their parameters include:

- **master** - component object (here: datatable);
- **column** - related column object;
- **node** - HTML element (here: header td).

You can either extend or redefine the functionality of any existing content element, or create a totally custom element

Extending Existing Content Element

For datatable footer there exists a pre-built **summColumn** element that counts the sum from column rows and displays them under the main part or the datatable.

If you need to show an average value instead of a sum inherit from the *summFilter* and redefine its **refresh()**:

```
webix.ui.datafilter.avgColumn = webix.extend({
    refresh:function(master, node, value){
        var result = ;
        master.mapCells(null, value.columnId, null, 1, function(value){
            value = value*1;
            if (!isNaN(value))
                result+=value;
            return value;
        });

        node.firstChild.innerHTML = Math.round(result/master.count());
    }
}, webix.ui.datafilter.summColumn);
```

Creating Custom Content Element

A custom content element can be created totally from the scratch.

It must possess the following mandatory **methods** (declare them even if you don't need any specific performance from them):

- **getValue** - gets the current value of an element;
- **setValue()** - sets the value to an element;
- **render()** - paints an element;
- **refresh()** - defines dynamic performance of an element.

And you may define as many methods as you need.

A content element can feature the following **properties**:

- **trackCells** (boolean) - indicates whether an element should be refreshed each time component data changes (if *true*, *refresh()* method of a content element will be called automatically).

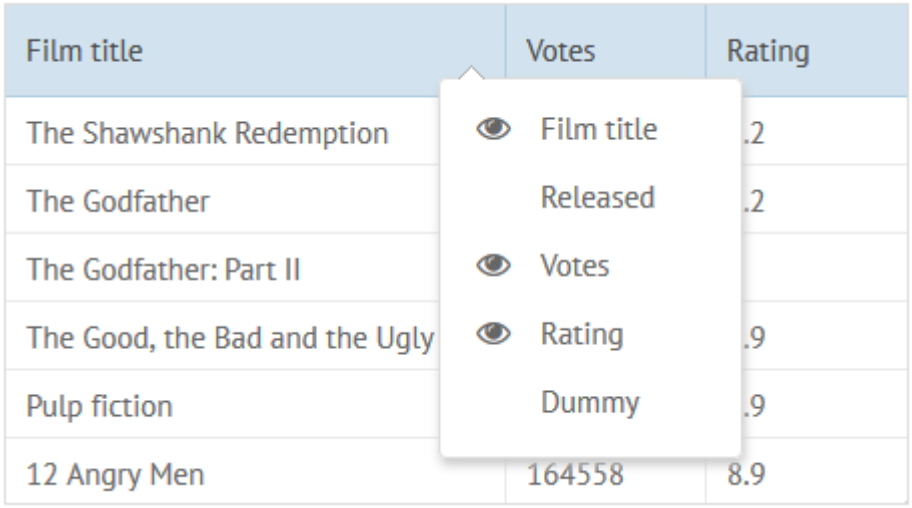
```
webix.ui.datafilter.customFilterName = {
    getValue:function(){}, //not used methods
    setValue: function(){},
    refresh: function(master, node, column){
        //event handlers
```

```
        node.onchange = function(){...};
        node.onclick = function(){...}
    },
    render:function(master, column){
        var html = "...";
        return html;
    }
};
```

RELATED SAMPLE: Custom content for header and footer

Available only in **PRO Edition**

Datatable Header Menu



Datatable **built-in header menu** is used to control column visibility with the help of a dedicated **popup list** with column names.

By default it is enabled by a **right click** over any column header while clicks on menu items change the state of the corresponding columns.

Simple Configuration

The simplest header menu is set with the help of a datatable **headermenu** property and has the following features:

- it is triggered by a right click over the header of **any** column;
- each menu item stands for a column;
- hidden columns are included into the menu without an "eye" (visible) icon;
- height of the menu is adjusted to the number of menu items;
- width of the menu is 150px.

```
{ view:"datatable", columns:[...], headermenu:true}
```

RELATED SAMPLE: Header menu in the DataTable

Extended Configuration

Headermenu can be presented as a JSON object with configuration options for the menu list. The options duplicate Webix list properties and may include:

- **width** (number) - width of the popup in pixels;
- **height** (number) - height of the popup in pixels;
- **autoheight** (boolean) - if *true*, adjusts popup height to the number of menu items. *True* by default;
- **yCount** (number) - the number of menu items to show;
- **scroll** (boolean) - if *true*, enables the scrollbar. *False* by default;
- **data** (object) - columns to show in the menu (if not set, all columns are reflected in the menu). Each item contains:
  - **id** - ID of the datatable column (the same one it has in the *columns* configuration);
  - **value** - title of the datatable column (its header, as a rule).
- **template** (string) - template of menu items (if you need to show a property other than *value* in the list).

Scrollable header menu with extra width

```
view:"datatable",
columns:[...],
headermenu:{
    width:250,
    autoheight:false,
    scroll:true
}
```

RELATED SAMPLE: Extended Header menu in the DataTable

Header menu with limited item quantity

```
view:"datatable",
columns:[...],
headermenu:{
    data:[
        { id:"year", value:"Released"},
        { id:"rating", value:"Rating"}
    ]
}
```

RELATED SAMPLE: Extended Header menu in the DataTable

Excluding a column from the menu

To exclude any column from the header menu, one should define a *headermenu* property for it and set its value to *false*:

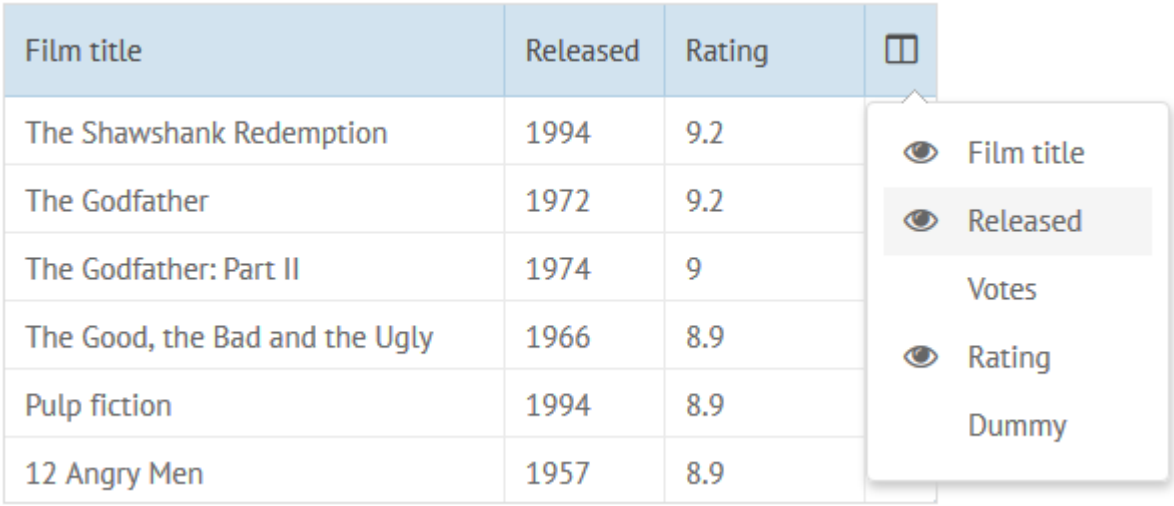
```
{
    view:"datatable",
    headermenu:true, //enables the header menu
    columns:[
        //excluding "id" column from the menu
```

```
        {id:"id", header:"#", headermenu:false},
        {id:"title", header:"Title"}, ...
    ]
}
```

Attaching Headermenu to Other Elements

Extra "headermenu" column

Header menu can as well be shown by a **left mouse click** over the special datatable column that features a "headerMenu" **content**.




```
view:"datatable",
columns:[
    { header:{ content:"headerMenu" }, headermenu:false, width:35 }
]
```

RELATED SAMPLE: Header menu icon in the DataTable

If there is such a column in the datatable configuration, the **headermenu** property for it can be omitted.

Content property of a header line is also used to set built-in filters and define column groups.

Showing header menu on demand

The headermenu unique ID can be accessed through datatable **config** object. If you know this ID, you can work with the headermenu like with any other popup window, e.g. show it when needed.

```
webix.ui({
    view:"datatable", id:"dt2", columns:[...], headermenu:true
});

//show the headermenu by API
function showMenu(node){
```

```
    var menu = $$("dt2").config.headermenu;
    $$(menu).show(node);
}
```

RELATED SAMPLE: Header menu icon in the DataTable

Available only in **PRO Edition**

Datatable Rowspans and Colspans

Datatable API allows for setting **rowspans** and **colspans** with the help of **span** configuration that is passed together with datatable data.

Country	Film title	Released	Votes	Rating
USA	The Shawshank Redemption	1994	678790	9.5
	The Godfather	1972	511495	9.2
	The Godfather: Part II	1974	319352	9
Highest Rating				**9.5**
Italy	The Good, the Bad and the Uc	1966	213030	8.9
	The Star Maker	1995	164558	7.9
Highest Rating				**8.9**

The data you want to display in the datatable is passed in the **data** object that contains:

- **data/url** - data for loading or parsing;
- **spans** - array of rowspans and colspans for the datatable;

```
webix.ui({
    view:"datatable",
    columns:[...],
    data:{
        data:grid_data,
        spans:[
            //span for 3 rows
            [1, "country", 1, 3],
            //span for 4 columns, styled
            ["sub1", "country", 4, 1, null, "highlight"],
        ]

    }
});
```

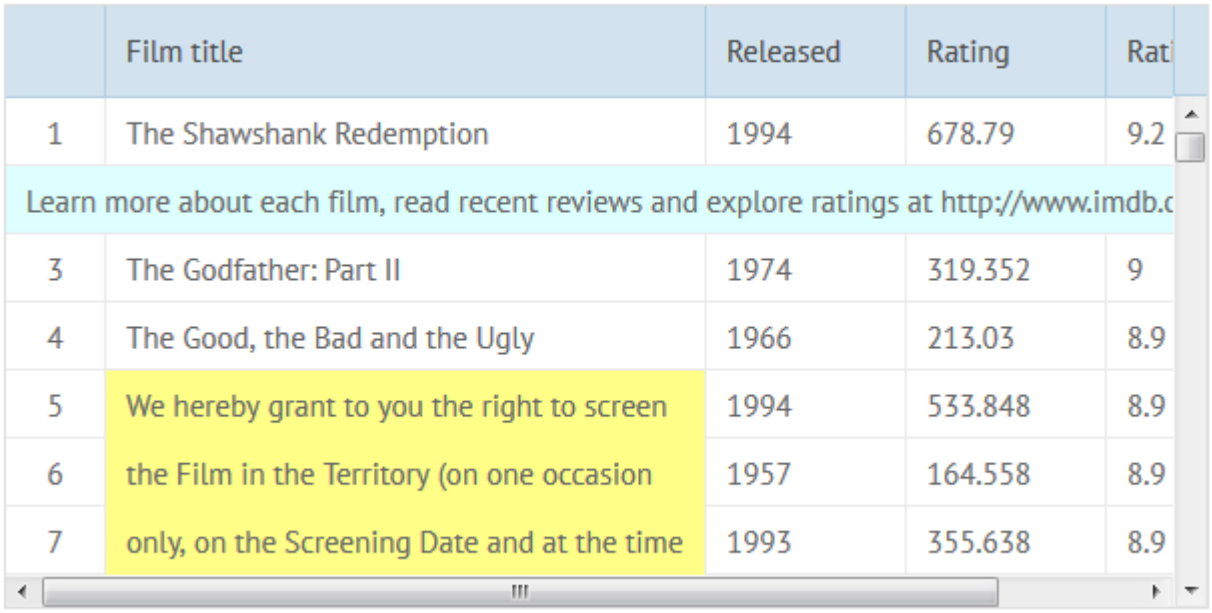
RELATED SAMPLE: Colspans in the DataTable

Each definition of a rowspan or colspan includes:

- **row id** - id of the row that starts colspan/rowspan;

- **column id** - id of the column that starts colspan/rowspan;
- **width** - how many columns should the span include. Default value is 1;
- **height** - how many rows should the span include. Default value is 1;
- **value** (optional) - value that the span area should display. If the value of the start cell is required, omit this field or pass an empty value ("", null);
- **css** (optional) - css class that should be applied to the span area.

CSS for Rowspan and Colspans



The simplest way to add styling to the rowspan or colspan is to mention its **css class** in the configuration (as shown above).

Extra rules for span elements should be applied to properly position long texts in case of a row span:

```
Span CSS

<!-- ensuring proper display-->
.webix_span_layer{
    background: transparent;
    position:absolute;
    left:0px;
    top:0px;
    height:0px;
    width:0px;
    overflow: visible;
}
.webix_dtable_span{
    position:absolute;
    background: #fff;
    white-space: normal;
```

```
}
<!--colouring-->
.hrow{
    background: #DFF;
}
```

Datatable configuration:

Styled Spans

```
var text2 = "Learn more about each film, read recent reviews and explore
ratings...";

webix.ui({
    view:"datatable",
    columns:[...],
    data:{
        data:grid_data,
        spans:[
            //span for 3 columns
            [ "2", "rank",  6, 1, text2, "hrow"]
        ]

    }
});
```

RELATED SAMPLE: Colspans in the Big DataTable

Data loading

DataTable can load data of various formats, such as:

- JSON;
- XML;
- JSArray;
- CSV.

Examples of data formats

In addition to the mentioned above, you can define any custom format (read how to do this in article Data drivers).

There are two main ways of loading data in DataTable:

To define datasource in the object constructor;

To use parse or load method.

Both ways lead to one and the same result and don't have any specificity.

Loading data from inline dataset

```
grid = new webix.ui({
    view:"datatable",
    ...
    data:[
        { id:1,name:"The Shawshank Redemption",year:1994},
        { id:2,name:"The Godfather",year:1972}
    ]
});
```

RELATED SAMPLE: Loading from inline datasource

Loading data from a data file

```
grid = new webix.ui({
    view:"datatable",
    ...
    url:"data/data.json"
});
```

RELATED SAMPLE: Loading from an external data file

Loading from XML, CSV, jsArray

When loading of non-json data, you need to specify the datatype property in config, or set the second parameter of the parse/load method to the name of the expected data type.

Also, you need to show in the columns configuration, which data will go to each column.

```
Loading from XML

grid = new webix.ui({
    view:"datatable",
    columns:[
        { map:"#cells[0]#", header:"Will use value from first cell sub-tag" }
        { map:"#details#", header:"Will use value from details attribute or sub-tag" }
    ],
    datatype:"xml"
    url:"data/data.xml"
});
```

RELATED SAMPLE: Loading from an external data file

If you use xml format with tag names equal to column ids, you may skip mapping, DataTable will be able to auto-map values

```
Loading from CSV

grid = new webix.ui({
    view:"datatable",
    columns:[
        { map:"#data1#", header:"First column" }
        { map:"#data2#", header:"Second column" }
    ],
    datatype:"csv"
    url:"data/data.csv"
});
```

RELATED SAMPLE: Loading from an external data file

For more details, check Data mapping

Converting HTML table to DataTable

To turn an HTML table into DataTable you should call method parse with next parameters:

- *data* - the id of the table;

- *datatype* - the format of data. Must be set to value 'htmltable'.

```
Loading data from an html table

<table id="films">
    <tr>
        <td width="40">id</td>
        <td width="200">Film title</td>
        <td width="80">Release year</td>
    </tr>
    ...
</table>
<script>
 grid = new webix.ui({
     view:"datatable",
     ...
 });

 grid.parse("films", "htmltable");
</script>
```

RELATED SAMPLE: Loading from HTML table

Loading from database

To load data from database table(s) you should deal with both client- and server-sides.

On the client side you must define parameter url (or call method load()) and specify a server-side script there.

On the server side you should write the script realizing server-client 'communication' that returns data in the XML or JSON format:

```
Static loading from db. Server-side code

<?php
   require_once("../../common/config.php");
   $data = new JSONDataConnector($conn, $dbtype);
   $data->dynamic_loading(30);
   $data->render_table("packages_plain","id","package, size, architecture,
section");
?>
```

```
Static loading from db. Client-side code.

grid = new webix.ui({
    ...
    url:"data/table_data.php"
});
//or
```

```
grid.load("data/table_data.php");
```

RELATED SAMPLE: Loading from DB

Loading big datasets (dynamic)

Generally, data source in case of dynamic loading is database.

On the client side you must define parameter url (or call method load()) with a file realizing server-side 'communication'.

On the server side, if you don't write server-side code by yourself and use WebixConnector you should just call the **dynamic_loading()** method:

```
Dynamic loading from db. Client-side code

grid = new webix.ui({
    ...
    url:"data/data_dyn.php"
});
```

```
Dynamic loading from db. Server-side code

require_once("../../common/config.php");//including the connector file
$data = new JSONDataConnector($conn, $dbtype);//initializing the connector
object
$data->dynamic_loading(30);//enabling dynamic loading
$data->render_table("packages_plain","id","package, size, architecture,
section");//loading data from the specified database table
```

RELATED SAMPLE: Dynamic loading from DB

Note, inside **dynamic_loading()** you should specify a number of records that will be loaded at once.

Dynamic related parameters

The library offers you different properties and methods that can help you to set when, how often and which data will be loaded.

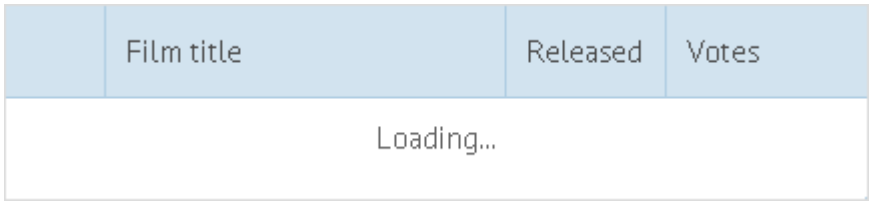
Table 1 Dynamic related parameters

Parameter	Description	Usage example
datafetch	Defines the start position DataTable will load data from	You want to load records starting from the fiftieth one, then you set **datafetch:50**. RELATED SAMPLE: Setting loading position

Parameter	Description	Usage example
datathrottle	Sets polling interval that can be used to decrease the number of requests to server	As a rule, data are getting loaded when a user starts to scroll DataTable. If he scrolls DataTable slowly, the app pushes a big amount of requests to the server, a major part of which is unnecessary. To exclude unnecessary requests and reduce server overloading, you can set the **datathrottle** property, for example to 5, and requests start to be sent to server at 5 seconds intervals. RELATED SAMPLE: Setting polling interval
loadahead	Allows you to avoid DataTable from possible 'blank spots' while scrolling	When you start scrolling, DataTable detects where you scroll (up or down) and loads next records in this direction (the number of records is defined by the parameter). If you set **loadahead:50**, then on each scrolling movement DataTable will load next 50 records. RELATED SAMPLE: Dynamic loading. Load Ahead

Loading screen

When you deal with large data size, it's useful to display the loading screen showing users that the app is actually doing something.



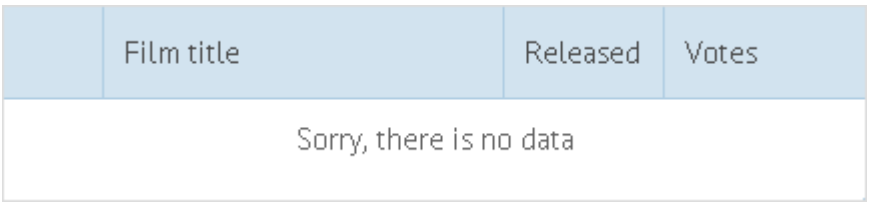
To enable the loading screen for DataTable you should call method showOverlay(), to disable it after - hideOverlay().

```
Using the loading screen in DataTable

grid = new webix.ui({
    view:"datatable",
    ...
    on:{
        onBeforeLoad:function(){
            this.showOverlay("Loading...");
        },
        onAfterLoad:function(){
            this.hideOverlay();
        }
    },
    url:"data/table_data.php"
});
```

RELATED SAMPLE: Loading screen

In case you don't call method hideOverlay() , instead of the loading screen you will set the overlay text. It can be useful in some situations. For example, when your DataTable doesn't contain any data, you can display the overlay text to inform users about it.




```
Defining the overlay text for DataTable

on:{
    onAfterLoad:function(){
        if (!this.count())
        this.showOverlay("Sorry, there is no data");
    }
}
```

RELATED SAMPLE: Adding overlay text

Loading configuration

You can load configuration from server side along with data. In such case your data will look as:

```
{
    config:{
        columns:[ ... columns configuration here... ],
        ... any extra options here, same as in constructor...
    },
    data:[ ... array of data objects here, as in above samples... ]
}
```

Read more on the topic in article External configuration.

Examples of data formats

Table 1 Initial data

Country	Population	Estimate
China	1 341 990 000	2011
India	1 193 290 000	2011
United States	311 960 000	2011
Indonesia	237 556 363	2010

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<data>
  <item id="1">
    <country> China </country>
    <population> 1 341 990 000 </population>
    <year> 2011 </year>
  </item>
  ...
  <item id="4">
    <country>  Indonesia </country>
    <population> 237 556 363 </population>
    <year> 2010 </year>
  </item>
</data>
```

you are not limited to the above xml format, check Data mapping for instruction how any xml file can be loaded in the datatable.

JSON

```
[
    { id:"1", country: "China", population: "1 341 990 000", year: "2011"
},
    { id:"2", country: "India", population: "1 193 290 000", year: "2011"
},
    { id:"3", country: "United States", population: "311 960 000", year:
"2011" },
    { id:"4", country: "Indonesia", population: "237 556 363", year:
"2010" }
]
```

JSArray

```
[
    [ "China", "1 341 990 000", "2011" ],
    [ "India", "1 193 290 000", "2011"],
    [ "United States", "311 960 000", "2011"],
    [ "Indonesia", "237 556 363", "2010"]
]
```

CSV

```
China, 1 341 990 000, 2011
India, 1 193 290 000, 2011
United States, 311 960 000, 2011
Indonesia, 237 556 363, 2010
```

Data mapping

It's possible to define mapping between loaded data and columns of datatable.

By default, DataTable will assume that incoming json|xml has attribute with the same name as columns id. So for well formed xml|json you need to define only column id.

In case of xml|json in custom format, you can use "map" attribute of column, it defines which property of loaded data object need to be used foir that column in grid

```
Loading from XML

grid = new webix.ui({
    view:"datatable",
    columns:[
        { id:"size",    map:"#cells[0]#", header:"Will use value from
first cell sub-tag" }
        { id:"url",     map:"#details#", header:"Will use value from
details attribute or sub-tag" }
    ],
    datatype:"xml"
    url:"data/data.xml"
});
```

The code above will correctly show data from XML as in:

```
<rows>
    <item
details="http://mysite.com"><cell>15x24</cell><cell>19.02.2008</cell></ite
m>
    <item
details="http://mysite.com"><cell>10x12</cell><cell>22.02.2008</cell></ite
m>
    ...
</rows>
```

Specific of CSV and jsArray data types

During csv(jsarray) parsing, its columns are named as data1 .. dataN You can use that values as column's ids

```
grid = new webix.ui({
    view:"datatable",
    columns:[
        { id:"data1"  },
        { id:"data2"  }
    ],
```

RELATED SAMPLE: Loading from inline datasource

or you can use mapping, to define more meaningfull names for columns

```
grid = new webix.ui({
    view:"datatable",
    columns:[
        { id:"url",   map:"#data1#"  },
        { id:"size",  map:"#data2#"  }
    ],
```

Converting values to the native type

When you loading data from xml, all values will be loaded as string (in json - string or numbers), but it is not very useful as we need to work with different types of objects, date objects for example.

Mapping logic allows to define the native type of value, and during data mapping it will be converted from the string to the defined type

```
webix.ui({
    view:"datatable",
    columns:[
        { id:"start", map:"(date)#start#" }
    ]
```

RELATED SAMPLE: Converting strings to dates

If id of columns and name of mapped field is equal you can shorten it to

```
webix.ui({
    view:"datatable",
    columns:[
        { id:"start", map:"(date)" }
    ]
```

Event more complex data mapping

If all above doesn't work you can use scheme property to define custom data mapping function

```
webix.ui({
    view:"datatable",
    scheme:{
         $init:function(obj){
         //obj - data object from incoming data
             obj.count = obj.cells[]; //set value based on some data in
incoming dataset
             obj.price = obj.cells[1];
             obj.total = obj.price * obj.count;   //or calculate values on
the fly
         }
```

```
    },
    columns:[
        { id:"count" },
        { id:"price" },
        { id:"total" }
    ],
    ...
```

Sizing and Resizing

In this chapter you will learn how to set sizes for different DataTable elements and then change them, if it's needed.

We will consider 3 elements:

DataTable container;

Column;

Row.

and 3 possible situations (scenarios) for them:

Setting the fixed size;

Autosizing;

Resizing.

Setting the fixed size

DataTable container

By default, DataTable adjusts to the parent container size that gives us 2 ways to set the fixed size for it:

To set the size of the parent '*div*' container;

To use the width and height parameters within DataTable constructor.

```
Setting the fixed size for DataTable

//way 1
<div id="box" style="width:300px;height:400px;"></div>

grid = new webix.ui({
    container:"box",
    view:"datatable",
    ...
});
//way 2
<div id="box"/>

grid = new webix.ui({
    container:"box",
    view:"datatable",
    height:400,
    width:300,
    ...
});
```

RELATED SAMPLE: Adjusting to the parent container

RELATED SAMPLE: Sizing through the 'width', 'height' parameters

Column

Fixed column width

By default, datatable columns feature **fixed size** of **100px**. To change it, you should either:

- use **width** while configuring the column to set width to a column individually, or
- use **columnWidth** while configuring the datatable to set common width for all the columns. Like default common width, it can be overriden by individual *width* value, or
- use **minWidth** property in the column configuration to define minimal width for it. If there's more space initially or after resizing, column width will br increased, but it can never be less than *minWidth* value, or
- use **minColumnWidth** property in datatable configuration to set common minimal width for all columns. It can be overriden by inidividual *minWidth* value.

If you set the width in such a way, it will save its value regardless of any other enabled size parameters (e.g. autowidth). BTW, you can set different widths for different rows.

```
Setting different widths for columns

grid = new webix.ui({
    view:"datatable",
    ...
    columns:[
        { id:"title", header:"Film title" ,   width:80},
        { id:"title", header:"Release year" , minWidth:50}
    ]
});
```

```
Altering common width for all columns

grid = new webix.ui({
    view:"datatable",
    columnWidth:200,
    columns:[
        { id:"title", header:"Film title"},
        { id:"title", header:"Release year"}
    ]
});
```

Relative Sizing

If you set widths for columns and their sum is less than the width of the parent container, you can use the **fillspace** attribute to force some of columns to widen for filling the unused space.

	Film title	Year	Votes
1	The Shawshar	1994	678790
2	The Godfather	1972	511495

	Film title	Year	Votes
1	The Shawshank Redemption	1994	678790
2	The Godfather	1972	511495

```
Using attribute fillspace

grid = new webix.ui({
    view:"datatable",
    ...
    columns:[
        { id:"title", header:"Film title", fillspace:true},
        ...
    ]
});
```

RELATED SAMPLE: Adjusting columns to the parent container

There can be more than one **fillspace** in the datatable; in this case width will be calculated on the base of a proportion defined by numeric values:

```
grid = new webix.ui({
    view:"datatable",
    ...
    columns:[
        { id:"id", header:"ID", fillspace:1},
        { id:"title", header:"Film title", fillspace:4}
    ]
});
```

In the code above *title* column is 4 times bigger then *id* column, which is 20 to 80 percent relation.

Adjusting column to its content

If you want to adjust the width of a column to the related content size, you can use attribute **adjust**. It is defined for the needed column individually and features two modes:

- **data** - adjusts column width to the content of the widest item in it;
- **header** - adjusts column width to its header content

```
Adjusting the column width to fit the content size
```

```
grid = new webix.ui({
    view:"datatable",
    ...
    columns:[
        { id:"title", header:"Film title", adjust:"data"},
        { id:"rank", header:"Rank", adjust:"header"}
        ...
    ]
})
```

RELATED SAMPLE: Adjusting columns to the content

Note that resulting column width after adjusting won't be lesser than **minWidth** for this column provided that the latter is set.

Row

To set the fixed height for a row you should use **$height** while defining the data you will load into DataTable. If you set the height in such a way, it will save its value regardless of any other enabled size parameters (e.g. - autoheight).

```
Setting different heights for rows

grid = new webix.ui({
    view:"datatable",
    ...
    data:[
        { $height:20, id:1, title:"The Shawshank Redemption", year:1994},
        { $height:40, id:2, title:"The Godfather",            year:1972},
        ...
    ]
});
```

RELATED SAMPLE: Setting custom size for rows

Autosizing

You can enable width, height (or both of them) autosizing to adjust DataTable to the parent container size horizontally or vertically. Autosizing is enabled by the following parameters:

- autowidth;
- autoheight.

```
DataTable autosizing
```

```
grid = new webix.ui({
    view:"datatable",
    ...
    autoheight:true,
    autowidth:true
});
```

RELATED SAMPLE: Height and width autosizing

Resizing

By resizing we mean 2 possible scenarios:

- resizing columns, rows by hovering, dragging mouse over some border;
- setting other sizes for DataTable elements programmatically.

RELATED SAMPLE: Row auto-height

Mouse resizing

To enable the possibility to resize columns (rows) by mouse you should use one of the following parameters (or both of them at once):

- resizeRow;
- resizeColumn.

```
Enabling the possibility to resize columns and rows by mouse

grid = new webix.ui({
    view:"datatable",
    ...
    resizeColumn:true,
    resizeRow:true
});
```

RELATED SAMPLE: Column and row resizing

For these resizing options there exist relevant events onRowResize and onColumnResize.

Dynamic resizing

For dynamic resizing DataTable elements, the library offers method resize(). When you change size of some element and call this method, child(parent) container of the calling element will adjust to his new size.

```
Adjusting the parent container to the DataTable size

grid = new webix.ui({
    view:"datatable",
    ...
```

```
})

grid.define("width", 700);
grid.resize();
```

RELATED SAMPLE: Dynamical adjusting to the parent container

In some situations you may need to adjust an element to the size of the outer parent HTML container. In such the situation you may use method adjust() instead of resize():

```
Adjusting DataTable to the size of the parent container

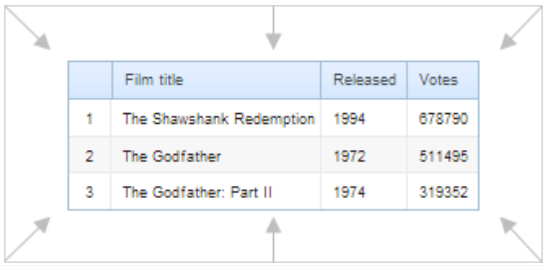
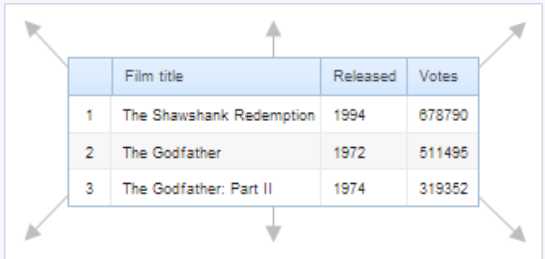
grid = new webix.ui({
    container:"box",
    view:"datatable",
    ...
})

webix.toNode("box").style.width = "700px";
grid.adjust();
```

RELATED SAMPLE: Dynamical adjusting to the parent container

The resize() and adjust() methods can lead to one and the same effect. For example, you have DataTable placed into a 'div' container, named as 'box'. The initial height of DataTable is 50 px. You want to increase it to 80 px. Possible solutions can look as shown in the table below:

Table 1 Use of adjust() and resize() methods

Image	Related code
	`document.getElementById('box').style.height="80px";` `grid.adjust();`
	`grid.define("height", 80);` `grid.resize();`

You can use events to call the mentioned above methods:

```
Adjusting DataTable to the size of a window, it□s placed into

webix.event(window, "resize", function(){ grid.adjust(); })
```

RELATED SAMPLE: Sizing and events

Dynamic Adjusting Columns and Rows

To dynamically adjust column width of row height to the size of their content, you can you the corresponding methods:

- **adjustColumn(id)** - adjusts the with of a chosen column specified by its ID;
- **adjustRowHeight(id)** = adjusts height of a chosen tow specified by its ID. The method can slow down the application.

Scrolling

By default, scrolling (vertical and horizontal) is enabled in DataTable. It can be disabled by parameters scrollY, scrollX.

```
Disabling vertical scrolling

grid = new webix.ui({
    view:"datatable",
    ...
    scrollY:false
});
```

The size of the scroll bars (vertical and horizontal at the same time) can be managed by parameter scrollSize (by default, 18px).

```
Managing the size of the scroll bars

grid=webix.ui({
    view:"datatable",
    ...
    scrollSize:20
})
```

You can also force DataTable to be scrolled just by whole rows, i.e. you won't be able to scroll along the full length of rows. The related parameter you should enable is scrollAlignY:

```
Scrolling DataTable by whole rows

grid = new webix.ui({
    ...
    scrollAlignY:true
});
```

RELATED SAMPLE: Different scrolling behaviours

Internationalization

DataTable provides support for numbers, currency and date internationalization.

The **standard package** of the Webix library includes 9 locales, namely:

- **'en-US'** - North American (used by default);
- **'ru-RU'** - Russian;
- **'fr-FR'** - French;
- **'ja-JP'** - Japanise;
- **'be-BY'** - Belorussian;
- **'de-DE'** - German;
- **'es-ES'** - Spanish;
- **'it-IT'** - Italian;
- **'zh-CN'** - Chinese.

Webix Pro edition includes over **300 locales** to match a great variety of cultures.

To localize your app into some other language, you should do the following:

Create a locale file.

Activate the locale.

Creating locales

Locale is created in a separate *js* file named as **"ll-CC"**, where

- **ll** - a two-letter language code;
- **CC** - a two-letter country code.

```
Creating Spanish (Spain) locale for DataTable. The 'es-ES.js' file

webix.i18n.locales["es-ES"]={   //"es-ES" - the locale name, the same as
the file name
  groupDelimiter:" ",         //a mark that divides numbers with many
digits into groups
  groupSize:3,                //the number of digits in a group
  decimalDelimiter:",",       //the decimal delimiter
  decimalSize:2,              //the number of digits after the decimal
mark

  dateFormat:"%d/%m/%Y",      //applied to columns with
'format:webix.i18n.dateFormatStr'
  timeFormat:"%H:%i",         //applied to columns with
'format:webix.i18n.dateFormatStr'
  longDateFormat:"%d %F %Y",  //applied to columns with
```

```
'format:webix.i18n.longDateFormatStr'
  fullDateFormat:"%d.%m.%Y %H:%i",//applied to cols with
'format:webix.i18n.fullDateFormatStr'

  price:"{obj} EUR",//EUR - currency name. Applied to cols with
'format:webix.i18n.priceFormat'
  calendar:{
        monthFull:["Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio",
"Julio", "Agosto",
                "Septiembre", "Octubre", "Noviembre", "Diciembre"],
        monthShort:["En", "Feb", "Mar", "Abr", "Mayo", "Jun", "Jul",
"Ago", "Sep", "Oct",
                "Nov", "Dic"],
        dayFull:["Domingo", "Lunes", "Martes", "Miercoles", "Jueves",
"Viernes", "Sabado"],
        dayShort:["Dom", "Lun", "Mar", "Mier", "Jue", "Vier", "Sab"]
  }
};
```

- **monthFull** - the full names of months starting from January;
- **monthShort** - the short names of months starting from January;
- **dayFull** - the full names of week days starting from Sunday;
- **dayShort** - the short names of week days starting from Sunday.

Activating the locale

To activate locale:

Include the related locale file on the page;

Call *webix.i18n.setLocale('locale_name')* (**!before DataTable initialization**).

```
<script src="../locale/es-ES.js" type="text/javascript"
charset="utf-8"></script>

<script>
    webix.i18n.setLocale("es-ES");
    grid = new webix.ui({
        view:"datatable",
        columns:[
            { header:"Date",  id:"start",
format:webix.i18n.dateFormatStr},
            { header:"LongDate", width:170,  id:"start",
format:webix.i18n.longDateFormatStr},
            { header:"Price",  id:"number", format:webix.i18n.priceFormat
},
        ],
        ...
```

```
    });
</script>
```

RELATED SAMPLE: Internationalization

External configuration

You have the possibility to load not only data but also DataTable configuration from external file or the server side.

Film title	Released	Votes
The Shawshank Redemption	1994	678790
The Godfather	1972	511495

Providing configuration as part of data

Configuration from a JSON file

To load both data and configuration from JSON file, initialize DataTable as shown below:

```
dtable = new webix.ui({
    container:"box",
    view:"datatable",
    url:"data/data-config.json" //path to json file. See the file content
below
});
```

The specified **data-config.json** file should have the following structure:

```
'data-config.json' file

{
    "config":{
        columns:[
            { id:"title",   header:"Film title", width:200},
            { id:"year",    header:"Released",   width:80},
            { id:"votes",   header:"Votes",      width:100}
        ],
        height:100,
        autowidth:true
    },
    "data":[
        {id:"1", title:"The Shawshank Redemption", year:"1994",
votes:"678790"},
        {id:"2", title:"The Godfather",           year:"1972",
votes:"511495"}
    ]
}
```

Configuration from an XML file

To load both data and configuration from XML file, initialize DataTable as shown below:

```
dtable = new webix.ui({
    container:"box",
    view:"datatable",
    url:"data/data-config.xml"// path to XML file. See the file content
below
});
```

The specified **data-config.xml** should have the following structure:

```
'data-config.xml' file

<?xml version='1.0' encoding='utf-8' ?>
<data>
    <config>
        <columns>
            <column id="title"   header="Film title"
width="200"></column>
            <column id="year"     header="Released"
width="80"></column>
            <column id="votes"    header="Votes"
width="100"></column>
        </columns>
        <height>100</height>
        <autowidth>true</autowidth>
    </config>
    <item id='1' title='The Shawshank Redemption' year='1994'
votes='678790'></item>
    <item id='2' title='The Godfather'           year='1972'
votes='511495'></item>
</data>
```

RELATED SAMPLE: Loading configuration from external url

Loading configuration from a separate file

Configuring from the server side

To load DataTable configuration from the server side, initialize DataTable as shown below:

```
webix.ajax("data/config.json", function(text){
    var config = webix.DataDriver.json.toObject(text);
    new webix.ui({
        container:"box",
        view:"datatable",
        columns:config,
```

```
        data: [
            { id:1, title:"The Shawshank Redemption", year:1994,
votes:678790},
            { id:2, title:"The Godfather",           year:1972,
votes:511495}
        ]
    });
});
```

The specified **config.json** file should have the following structure:

```
'config.json' file

[
    { id:"title",   header:"Film title",  width:200},
    { id:"year",    header:"Released" ,   width:80},
    { id:"votes",   header:"Votes",       width:100}
]
```

Configuring and populating from the server side

To load both data and configuration from the server side, initialize DataTable as shown below:

```
webix.ajax("data/data-config.json", function(text){
    var obj = webix.DataDriver.json.toObject(text);
    new webix.ui({
        container:"box",
        view:"datatable",
        columns:obj.config,
        data:obj.rows
    });
});
```

The specified **data-config.json** file should have the following structure:

```
'data-config.json' file

{
    "config":[
        { id:"title",   header:"Film title", width:200},
        { id:"year",    header:"Released",   width:80},
        { id:"votes",   header:"Votes",      width:100}
    ],
    "rows":[
        { "id":1, "title":"The Shawshank Redemption", "year":1994,
"votes":678790},
        { "id":2, "title":"The Godfather",           "year":1972,
"votes":511495}
    ]
}
```

RELATED SAMPLE: Loading configuration from server side

Look and Feel

- Cells Templates
- Number and Date Formatting
- Styling
- Split Mode or 'Frozen Columns'

Cells Templates

String templates

To customize data content and specify which data will be presented in DataTable and how, you can use templates. Data templates are set in the related column by attribute **template**.

By #value# you can refer to the data property specified for the column. It can be especially useful when you define a common template for several columns.

You can read about templates syntax in article Templates.Syntax.

```
Specifying data template for a specific column

grid = new webix.ui({
    view:"datatable",
    ...
    columns:[
        { id:"title", header:"Film title",
template:"<strong>#title#</strong>",
        { id:"year",  header:"Release year" , template:"at #year#",}
    ]
});
```

RELATED SAMPLE: Using string templates

Templates can be used for presenting almost any content in DataTable (anything which can be done through HTML - can be placed in DataTable's cell) such as: images, links, numbers, string, dates.

Table 1 Templates for different types of content

Content	Example of template
strings	`template:"<strong>#title#</strong>"`
images	`template:"<img src='.imgs/#id#.jpg'/>"`
links	`template:"<a href='http://google.com?q=#title#'>#title#</a>"`

Complex templates

You can set custom template for a column by presenting the **template** attribute as a function (function accepts raw object and returns the result string).

```
grid = new webix.ui({
    view:"datatable",
    columns:[
```

```
      {id:"title", header:"Film title"},
      {id:"votes", header:"Votes", template:function(obj){ return
obj.votes/1.2547}};
    ],
    ...
})
```

RELATED SAMPLE: Converting strings to dates

For more on styling, read article Styling

Combining templates and formats

You can use both template and format attributes for the same column. In such case column will format data first, and will use it in template after that.

```
grid = new webix.ui({
    view:"datatable",
    columns:[
      {id:"votes", header:"Votes",  format:webix.i18n.numberFormat,
            template:"no more than #value#"}
    ],
    ...
})
```

Adding buttons in templates

You can insert any custom html to the row elements, which gives an easy way for adding buttons or similar controls to Datatable's rows

```
grid = new webix.ui({
    view:"datatable",
    columns:[
      {id:"votes", header:"Votes",  template:"#value#
            <input type='button' value='Show details'
class='details_button'>'"}
    ],
    on_click:{
        details_button:function(id, ev){
            //will be called on button click
            some_custom_method(id.row, id.column);
        }
    }
    ...
})
```

RELATED SAMPLE: Datatable: Custom Handler

Built-in templates

Through the template you can define common elements for the cells belonging to one and the same column, namely:

- checkboxes;
- radio buttons;
- "edit" cons;
- "trash" icons.

You can define additional common elements within the component type.

```
webix.ui({
    view:"datatable",
    ...
    columns:[
        { id:"ch1", header:"", template:"{common.checkbox()}"},
        { id:"ra1", header:"", template:"{common.radio()}"}
    ]
});
```

RELATED SAMPLE: Checkbox and radio in DataTable

Learn more about checkboxes and radios in a separate article - Checkbox and Radio in DataTable

```
webix.ui({
    view:"datatable",
    ...
    columns:[
        { id:"edit", header:"", template:"{common.editIcon()}"},
        { id:"trash", header:"", template:"{common.trashIcon()}"}
    ]
});
```

You can set built-in template via a function:

```
webix.ui({
    view:"datatable",
    ...
    columns:[
        { id:"ra1", template:function(obj, common, value, config){
             return common.radio(obj, common, value, config);
        }},
        { id:"trash", header:"", template:function(){
            return common,editIcon();
        }}
```

```
    ]
});
```

As you can see, **common.checkbox()** and **common.radio()** functions take four parameters:

- **item object** with its properties from the dataset;
- **common** object with four methods:
  - common.checkbox(*obj, common, value, config*);
  - common.radio(*obj, common, value, config*);
  - common.editIcon();
  - common.trashIcon();
- **value** - current checkbox/radio state;
- **config** - column configuration object.

Templates.Syntax

There are several ways to define a template:

HTML text

```
grid = new webix.ui({
    view:"datatable",
    ...
    columns:[
        { id:"col1", header:"Column1", template:"#Package# :
#Version#<br/>#Maintainer#"},
    ]
});
```

In the code snippet above, values within sharps are replaced with the related properties from data objects that can look like shown below:

```
//for next data
[
 {Package:"Webix Connector", Version:1.1, Maintainer:"Server side Team"},
 {Package:"Webix Scheduler", Version:3.5, Maintainer:"Scheduler Team"},
...
]
//Above template will generate text as
var template ="#Package# : #Version#<br/>#Maintainer#"
var text =    "Webix Connector : 1.1 <br/>#Maintainer#"
```

Function

```
grid = new webix.ui({
    view:"datatable",
    ...
    columns:[
        { id:"col1", template:function(obj){ return obj.Package
+"<br/>"+obj.Maintainer; }},
    ]
});
```

HTML container

```
<textarea id="template_container" style="display:none">
 #Package# : #Version# <br/>#Maintainer#</textarea>
// instead of 'textarea' can be any appropriate HTML tag
```

```
...
grid = new webix.ui({
    view:"datatable",
    ...
    columns:[
        { id:"col1", header:"Column1",
template:"html->template_container"},
    ]
});
```

External file

```
grid = new webix.ui({
    view:"datatable",
    ...
    columns:[
        { id:"col1", header:"Column1",
template:"http->../common/template.html"},
    ]
});
```

Number and Date Formatting

Number formatting

To set the required format for number fields you should use attribute **format**.

```
Using number templates

grid = new webix.ui({
    view:"datatable",
    columns:[
        { id:"votes", format:webix.Number.numToStr({
            groupDelimiter:",",
            groupSize:3,
            decimalDelimiter:".",
            decimalSize:2})
        },
        ...
    ]
});
```

RELATED SAMPLE: Using number templates

Method **numToStr()** allows you to set the following properties for fractional numbers:

- *groupDelimiter* - a mark that will be used to divide numbers with many digits into groups;
- *groupSize* - the number of digits in a group;
- *decimalDelimiter* - a mark that will be used as the decimal delimiter;
- *decimalSize* - the number of digits after the decimal mark.

There are two default presets - numberFormat and priceFormat which provides number and price formatting according to the current locale

```
grid = new webix.ui({
    view:"datatable",
    columns:[
        { id:"votes", format:webix.i18n.numberFormat },
        { id:"price", format:webix.i18n.priceFormat }
    ]
```

For details on number formatting methods, see Number formatting methods.

Date formatting

Dates in DataTable can be specified by both **DateTime** and **string** values.

But formatting can be applied only to **DateTime** objects. Therefore, to format string values you should convert them to DateTime format at first.

Common date formatting

To set the required format for date and time values, specify the **format** attribute.

As a value for the attribute you can use:

- Members of the **i18n** object;
- The **dateToStr()** method of the Date object.

```
Setting date formats

grid = new webix.ui({
    view:"datatable",
    columns:[
        { template:"#start#", header:"Date",
format:webix.i18n.dateFormatStr},
        { template:"#start#", header:"Y-m-d",
format:webix.Date.dateToStr("%Y-%m-%d")}
    ],
    data:[
        { text:"Finish", start:new Date(2012,11,12) },
        { text:"Start",  start:new Date(1988,1,29) }
    ]
});
```

RELATED SAMPLE: Using date templates

For details on date formatting methods, see article Date formatting methods.

Converting string values to DateTime format

To correctly work with dates, you need to have data in the grid as valid Date object, but in all common formats data is presented as string. DataTable provides a way to mark columns for auto-conversion from Date string to Date objects

```
grid = new webix.ui({
    view:"datatable",
    columns:[
        { map:"(date)#start#", header:"Date",
format:webix.i18n.dateFormatStr},

});
```

RELATED SAMPLE: Converting strings to dates

the *(date)* marker at start of map declaration forces data conversion from string to object. By default

datatable will assume that data will be in "%Y-%m-%d" (mysql date format), but you can change it as

```
webix.i18n.parseFormat = "%m.%d.%Y"
webix.i18n.setLocale();
```

For more details, see Date formatting methods.

Custom formatting

You can apply custom formatting to a column by presenting the **format** attribute as a function (function accepts raw value and returns the formatted one).

```
grid = new webix.ui({
    view:"datatable",
    columns:[
        {id:"title", header:"Film title"},
        {id:"votes", header:"Votes", format:function(value){
            value = some_custom_logic(value);
            return value; }
        }
    ],
    ...
})
```

Styling

You can use CSS in conjunction with DataTable content to achieve the desired look-and-feel. This article will help you to cover support of cascading style sheets (CSS) in DataTable.

You can fine-tune the style of:

- A whole table;
- Columns;
- Rows;
- Cells;
- The header.

A whole table

To apply some style to a whole table you can use the css parameter. As the value of the parameter you must specify the name of the appropriate css class.

Film title	Released	Votes
The Shawshank Redemption	1994	678790
The Godfather	1972	511495
The Godfather: Part II	1974	319352

Film title	Released	Votes
The Shawshank Redemption	*1994*	*678790*
The Godfather	*1972*	*511495*
The Godfather: Part II	*1974*	*319352*

```
Styling the whole table

<style>
     .my_style .webix_hcell{
          background:#009966;
          color:white;
          font-weight:bold;

     }
     .my_style .webix_column{
          font-style:italic;
          background:#ddFFdd;
     }
     .my_style .webix_column > div{
          border-color:#ddd;
     }

</style>

<script>
```

```
webix.ui({
    view:"datatable",
    css:"my_style",
    ...
})
</script>
```

RELATED SAMPLE: Basic styling

Columns

For applying some style to a specific column, you should specify **css** attribute within columns parameter while configuring the column.

As the value of the attribute you can specify:

- the name of the appropriate css class;
- an object containing the css properties.

Film title	Released	Votes
The Shawshank Redemption	1994	678790
The Godfather	1972	511495
The Godfather: Part II	1974	319352

Film title	Released	Votes
The Shawshank Redemption	1994	678790
The Godfather	1972	511495
The Godfather: Part II	1974	319352

```
Styling of a single column

<style>
    .my_style {
        font-weight:bold;
        color:#B51454;
    }

</style>

<script>
webix.ui({
    view:"datatable",
    columns:[
        // a separate css class
        { id:"title", header:"Film title", css:"my_style" },
        // an object with css properties
        { id:"votes", header:"Votes",      css:{"text-align":"right"}}
    ],
...
})
```

```
</script>
```

RELATED SAMPLE: Columns styling

Rows

Generally, to apply some style to a specific row, you may use the **$change** key of the scheme parameter. The key is a function that runs each time data is changed in the table. As a parameter, the function takes the data item object.

Film title	Released	Votes
The Shawshank Redemption	1994	678790
The Godfather	1972	511495
The Godfather: Part II	1974	319352

Film title	Released	Votes
The Shawshank Redemption	1994	678790
The Godfather	1972	511495
The Godfather: Part II	1974	319352

```
General styling of rows

<style>
    .highlight{
        background-color:#FFAAAA;
    }
</style>

<script>
webix.ui({
    view:"datatable",
    scheme:{
        $change:function(item){
            if (item.year > 1980)
                item.$css = "highlight";
        }
    },
    columns:[
        { id:"title", header:"Film title"},
        { id:"votes", header:"Votes"}
    ],
...
})
</script>
```

If you specify data directly inside the DataTable constructor, you have one more way to set the desired style for a row:

- set parameter **rowCss** to value **#css#**;
- use **$css** attribute inside the data parameter to set the desired style.

We don't recommend to dynamically change attributes with the '**$**' prefix (e.g. $css, $cellCss). Please, use the addCellCss, addRowCss methods instead.

```
Setting rows style directly in the dataset

<style>
    .my_style{
        background-color:#FFAAAA;
    }
</style>

<script>
grid = new webix.ui({
    view:"datatable",
    rowCss:"#css#",
    columns:[
        { id:"title",   header:"Film title"},
        { id:"votes",   header:"Votes"}
    ],
    data:[
        { id:1,
        $css:"my_style",
        title:"The Shawshank Redemption",
        votes:678790
        },
        { id:2,
        $css:{ "text-align":"right" },
        title:"The Godfather",
        votes:511495,
        }
    ],
...
});
</script>
```

RELATED SAMPLE: Rows styling

Applying hover styles

You can specify a custom style for selection of a row when the mouse pointer is over it.

Film title	Released	Votes
The Shawshank Redemption	1994	678790
The Godfather	1972	511495
The Godfather: Part II	1974	319352

Film title	Released	Votes
The Shawshank Redemption	1994	678790
The Godfather	1972	511495
The Godfather: Part II	1974	319352

For this purpose, you should define your css style in the **hover** property.

```
<style>
    .myhover{
        background: #F0DCB6;
    }
</style>

<script>
grid = webix.ui({
    view:"datatable",
    columns:[
        { id:"title",   header:"Film title",    width:200},
        { id:"year",    header:"Released",      width:80},
        { id:"votes",   header:"Votes",         width:100}
    ],
    hover:"myhover"
    ...
});
</script>
```

RELATED SAMPLE: Row hover

Cells

Applying cssFormat method

Generally, to apply some style to specific cells in a column, you should use the **cssFormat** attribute inside the columns parameter. The attribute sets a function that will define how cells of the column must be coloured. The function takes the following parameters:

- **cell value** (compulsory) - current cell value;
- **row item object** - all values for the current row;
- **row id** - ID of the row in question;
- **column id** - ID of the column in question.

(*function*).

Film title	Released	Votes
The Shawshank Redemption	1994	678790
The Godfather	1972	511495
The Godfather: Part II	1974	319352

Film title	Released	Votes
The Shawshank Redemption	1994	678790
The Godfather	1972	511495
The Godfather: Part II	1974	319352

```
General styling of cells
```

```
function mark_votes(value, config){
    if (value > 400000)
        return { "text-align":"right" };
    return value;
}

grid = new webix.ui({
    view:"datatable",
    columns:[
        { id:"title",   header:"Film title"},
        { id:"votes",   header:"Votes",   cssFormat:mark_votes }
    ],
    ...
})
```

Cell styling within configuration

If you specify data directly inside the DataTable constructor, you have one more way to set the desired style for a cell - property **$cellCss**:

We don't recommend to dynamically change attributes with the **'$'** prefix (e.g. $css, $cellCss). Please, use the addCellCss, addRowCss methods instead.

```
Setting cells style directly in the dataset
```

```
<style>
    .my_style{
        background-color:#FFAAAA;
    }
</style>

<script>
grid = new webix.ui({
    view:"datatable",
    columns:[
        { id:"title",   header:"Film title"},
        { id:"votes",   header:"Votes"}
    ],
```

```
    data:[
        { id:1,
          title:"The Shawshank Redemption",
          votes:678790,
          $cellCss:{
              votes:"highlight"
          }
        },
        { id:2,
          title:"The Godfather",
          votes:511495,
          $cellCss:{
              votes:{ "text-align":"right" }
          }
        }
    ],
...
})
</script>
```

RELATED SAMPLE: Cells styling

The header

To set style for a specific cell of the header you should use the following technique:

Film title	Released	Votes
The Shawshank Redemption	1994	678790
The Godfather	1972	511495
The Godfather: Part II	1974	319352

Film title	Released	Votes
The Shawshank Redemption	1994	678790
The Godfather	1972	511495
The Godfather: Part II	1974	319352

```
General styling of the header

<style>
    .my_style{
        background-color:#FFAAAA;
    }
</style>

<script>
grid = new webix.ui({
        view:"datatable",
        columns:[
            { id:"title",   header:{ text:"Film title", css:{
"background":"#AFA"}} },
            { id:"votes",   header:{ text:"Votes",      css:"my_style"} }
        ]
```

```
...
})
</script>
```

RELATED SAMPLE: Header styling

CSS maps

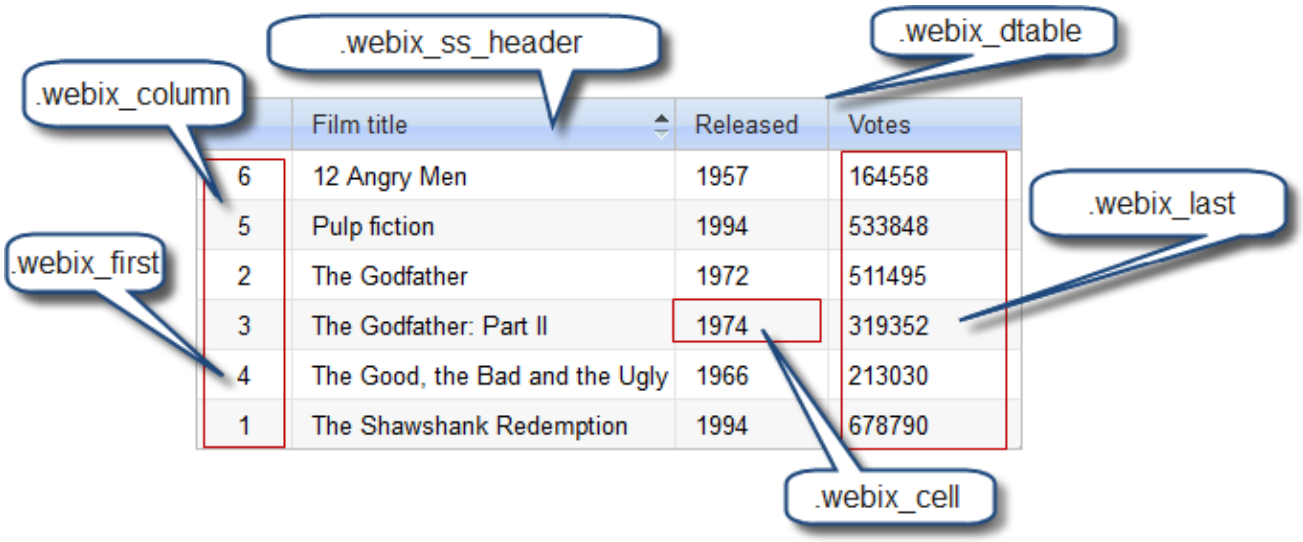
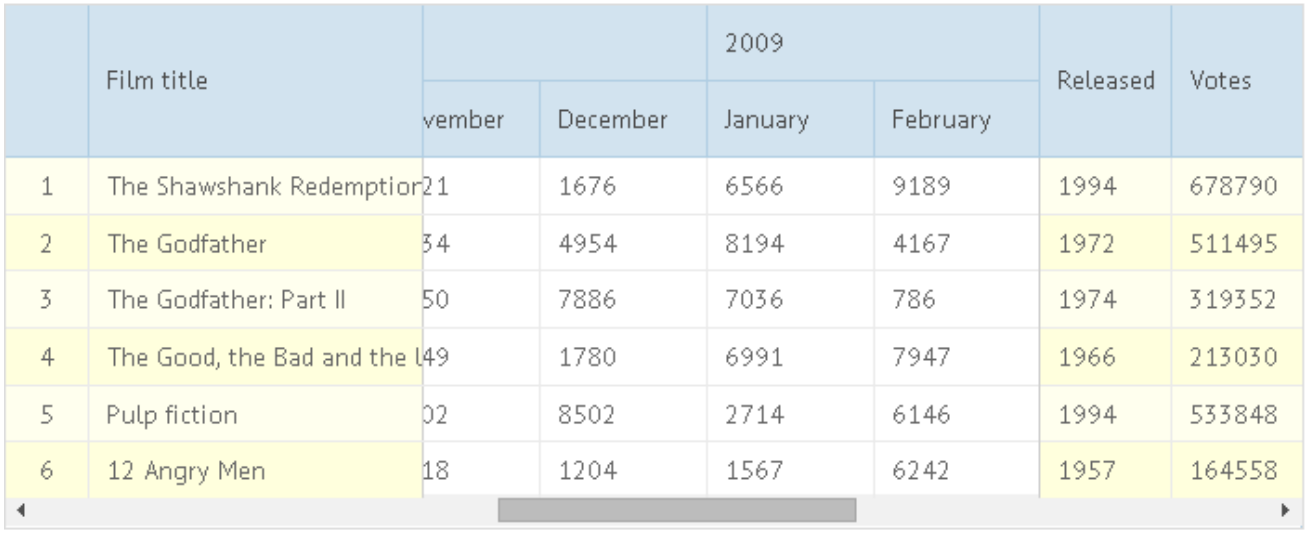


Table 1 CSS image maps

CSS class	Description
.webix_dtable	container with DataTable
.webix_ss_header	container with the header
.webix_ss_header .webix_hcell	a cell of the header
.webix_ss_header .webix_first	the first cell of the header
.webix_ss_header .webix_last	the last cell of the header
.webix_ss_footer	container with the footer
.webix_ss_footer .webix_hcell	a cell of the footer
.webix_ss_footer .webix_first	the first cell of the footer
.webix_ss_footer .webix_last	the last cell of the footer
.webix_ss_body	container with the table body
.webix_column	a column of the table
.webix_column .webix_last	the last column of the table
.webix_column .webix_first	the first column of the table
.webix_column .webix_cell	a cell of the column

Split Mode or 'Frozen Columns'

Split mode allows to have DataTable separated into two parts: 'frozen' and 'scrolled'. The frozen part is fixed. It can contain the last columns on the right(left) side or on the both sides at a time. The scrolled part is movable and can be scrolled horizontally.



	Film title		2009		Released	Votes
		vember	December	January	February		
1	The Shawshank Redemption	21	1676	6566	9189	1994	678790
2	The Godfather	34	4954	8194	4167	1972	511495
3	The Godfather: Part II	50	7886	7036	786	1974	319352
4	The Good, the Bad and the U	49	1780	6991	7947	1966	213030
5	Pulp fiction	02	8502	2714	6146	1994	533848
6	12 Angry Men	18	1204	1567	6242	1957	164558

To 'froze' columns, you should use parameters leftSplit and rightSplit.

```
Enabling the 'split' mode

dtable = new webix.ui({
    view:"datatable",
    ...
    leftSplit:2, // 2 columns will be frozen on the left side
    rightSplit:2 // 2 columns will be frozen on the right side
});
```

RELATED SAMPLE: Frozen columns. Basic implementation

Operations with data

- Copy-pasting data
- DataTable Filtering
- Math Operations
- Paging with Datatable
- Selection with Datatable
- DataTable Sorting
- DataTable Editing
- Checkbox and Radio in DataTable

Copy-pasting data

To enable the possibility to copy/paste DataTable data by the CTRL+C/CTRL+V keyboard shortcuts you should use parameter clipboard.

The parameter can have one of 3 values:

'*block*' (or *true*. It's the default value);

'*selection*';

'*repeat*'.

Each of the types has its specificity and define its behavior of copying.

```
Setting the desired behavior of copying

grid = new webix.ui({
        view:"datatable",
        ...
        clipboard:"selection",
});
```

RELATED SAMPLE: Copying between grids

'Block' copying

It's the default type and can be also specified by setting the **true** value.

It's can be characterized as follows:

- When the copy area is less than the one selected, copied cells are not repeated and fill only the related selected cells.
- When the copy area is greater than the one selected, copied cells fill not only the selected area but also the related cells nearby.

RELATED SAMPLE: 'block' copying

'Selection' copying

- When the copy area is less than the one selected, copied cells are not repeated and fill only the related selected cells.
- When the copy area is greater than the one selected, copied cells fill only the selected area and don't 'touch' the cells nearby.

RELATED SAMPLE: 'selection' copying

'Repeat' copying

- When the copy area is less than the one selected, copied cells are repeated a specific number of times to fill the selected cells.
- When the copy area is greater than the one selected, copied cells fill only the selected area and don't 'touch' the cells nearby.

RELATED SAMPLE: 'repeat' copying

DataTable Filtering

The library provides ability to filter data on client-side by one or several criteria using for this built-in or customly created filters.

Filters in the header

You can define a built-in filter in the header or footer of a column. Webix offers 8 filter types:

- in a standard free version:
  - **textFilter**;
  - **serverFilter** (text-based);
  - **serverSelectFilter** (select-based);
  - **selectFilter**;
  - **numberFilter**;
  - **dateFilter**;
- additionally, in the Webix Pro edition:
  - **richSelectFilter**;
  - **multiSelectFilter**;
  - **datepickerFilter**.

All of them are described in detail below.

Built-in filter is set by property **content** of the **header/footer** attribute. Note, to add a filter to the header(footer), the header(footer) must be specified as array.

```
columns:[
    { id:"title", header:[{content:"{filtername}Filter"}, "Title"] }
]
```


If you need to create a **custom filter** (or smth) for header content, or you need to **redefine** the behaviour of an already existing filter, go here for the instructions.

Text filter

Film title go	Released	Votes
The Godfather	1972	511495
The Godfather: Part II	1974	319352

Retrieves values that contain mask defined through text field.

```
{ id:"title",header:[ "Film title",{content:"textFilter"}] }
```

RELATED SAMPLE: Filtering. Built-in Text Filter, Select Filter

Server Filter

A text filter that works with backend. Retrieves values that contain mask defined through text field and sends a request to server to return filtered dataset.

```
{ id:"title",   header:["Film title", {content:"serverFilter"}] }
```

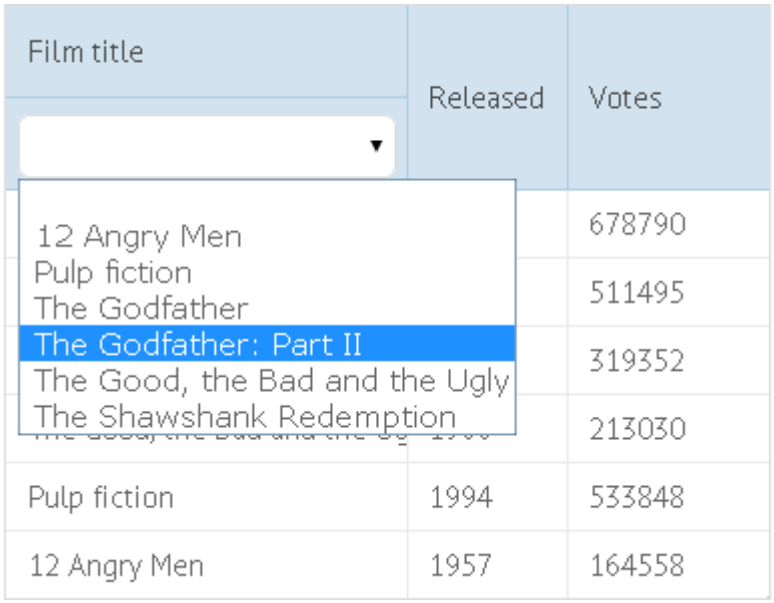
RELATED SAMPLE: Datatable: Serverside Filtering and Sorting

Request parameters include:

- *count* - the number of data records to return. Its value depends on dynamic loading parameters, if any;
- *start* - ID of the data record to start from (0 - beginning). Its value depends on dynamic loading parameters, if any;
- *filter[column_name]* - name of the column (in brackets) for which filtering is performed. Its value is a text value you've entered in the filter input.

If **serverside sorting** is enabled, data is both filtered and sorted on server.

Select filter

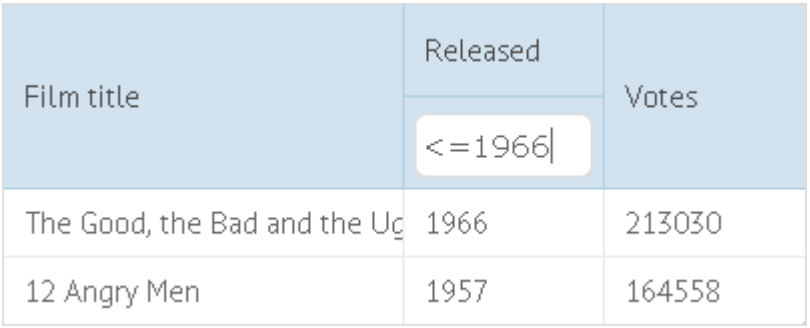


Retrieves values that contain mask defined through a dropdown list of possible values. Based on a standard HTML select input.

```
{ id:"title", header:["Film title",{content:"selectFilter"}] }
```

RELATED SAMPLE: Filtering. Built-in Text Filter, Select Filter

Number filter



Text filter used for number columns. Retrieves values which contain mask defined through text field. Allows users to use the following comporison operators in it:

- '=' - equal to;
- '>' - greater than;
- '<' - less than;
- '<=' - less or equal;
- '>=' - greater or equal.

```
{ id:"year", header:[ "Released",{content:"numberFilter"}] }
```

RELATED SAMPLE: Filtering. Built-in Numeric Filter

Date filter

Film title	Released
	<10.10.1980
The Godfather	02/05/1973
The Godfather: Part II	11/03/1975

A text filter used for date columns. Retrieves values that contain mask defined through text field. Allows users to use the following comporison operators in it:

- '>' - greater than;
- '<' - less than;
- '<=' - less or equal;
- '>=' - greater or equal.

There are 3 ways you can input data to the dateFilter:

'*yyyy*' - 4-digits year;

'*mm.yyyy*' - 2-digits month and 4-digits year separated by point;

'*dd.mm.yyyy*' - 2-digits day, 2-digits month and 4-digits year separated by points

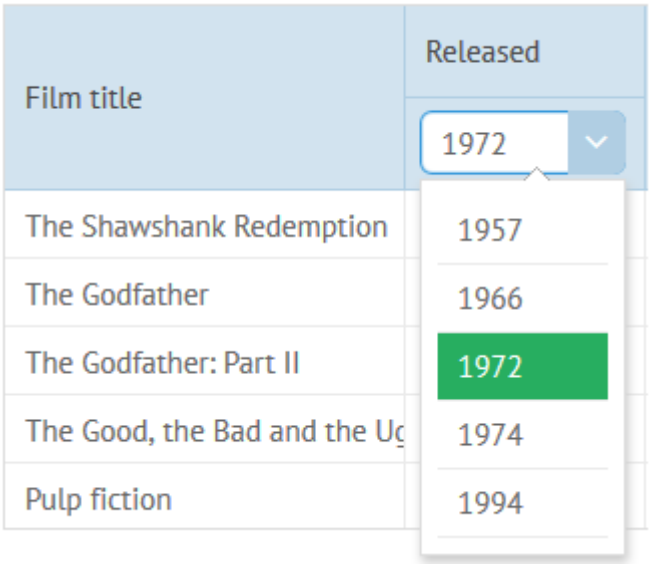
```
{ id:"year", header:[ "Released",{ content:"dateFilter"}],
    format:webix.i18n.dateFormatStr}
```

RELATED SAMPLE: Filtering. Built-in Date Filter

Richselect Filter



Available in **Webix Pro** edition only.



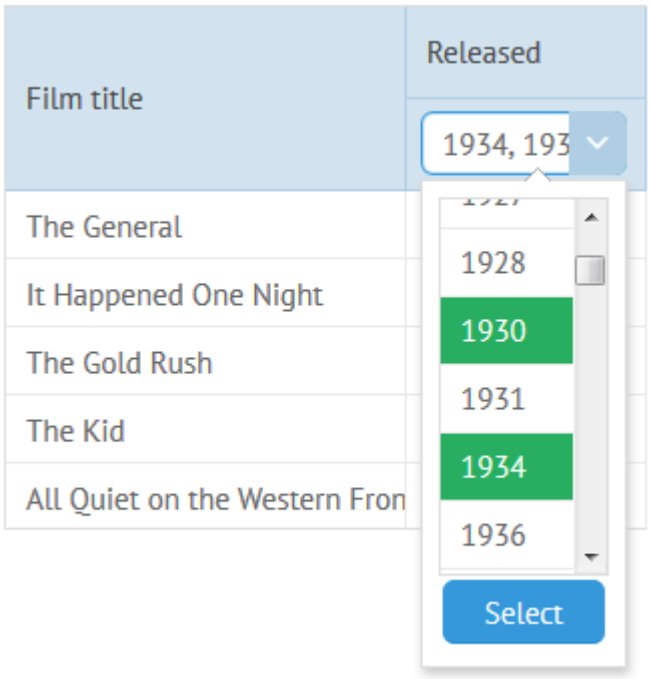
Retrieves values that contain mask defined through a popup list of possible values. Based on Webix richselect control.

```
{ id:"year", header:["Released", { content:"richSelectFilter" }] }
```

RELATED SAMPLE: Multi-select filter in the DataTable

Multiselect Filter

Available in **Webix Pro** edition only.



Retrieves values that contain mask defined through a popup list of possible values while **miltiple values** can be selected at once. Based on Webix multiselect control.

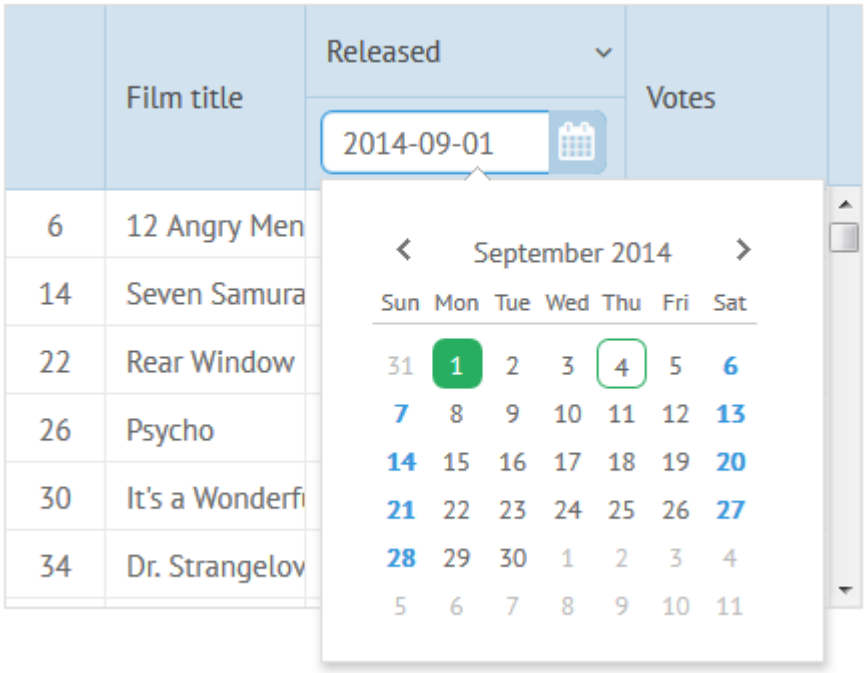
```
{ id:"year",header:["Released", { content:"multiSelectFilter" }]
```

RELATED SAMPLE: Multi-select filter in the DataTable

Datepicker Filter



Available in **Webix Pro** edition only.



Retrieves values that contain mask defined through the popup calendar. Based on Webix datepicker control.

```
{ id:"date", header:[ "Released", { content:"datepickerFilter" }],
        format:webix.i18n.dateFormatStr}
```

RELATED SAMPLE: Date filters in the DataTable

Filtering Logic

Note, each time you start to type text in such a filter, DataTable invokes the filterByAll() method. Each time the method is called, all data is re-filtered (previous results aren't preserved).

AND logic

By default, when you specify filters in several columns, DataTable applies AND logic to them, i.e. the table will display just records that meet all criteria at once.

RELATED SAMPLE: Filtering by several criteria (AND logic)

OR logic

Film title	Released	Votes
The Shawshank Redemption	1994	678790
The Godfather	1972	511495
The Godfather: Part II	1974	319352

Film title	Released	Votes
shaw	1972	
The Shawshank Redemption	1994	678790
The Godfather	1972	511495

In case you want to apply OR logic (to display records that meet at least one of criteria) you should redefine the filterByAll() method. For example, it can look like this:

```
Implementing OR logic for filters

grid = new webix.ui({
    view:"datatable",
    ...
    columns:[
        { id:"title", header:["Film title", {content:"textFilter"}] },
        { id:"year",  header:["Released",  {content:"selectFilter"}] },
        { id:"votes", header:"Votes" }
    ]
});

grid.filterByAll=function(){
//gets filter values
    var title = this.getFilter("title").value;
    var year = this.getFilter("year").value;
//unfilters the table if values were not selected
    if (!title  && !year) return this.filter();
//filters using OR logic
    this.filter(function(obj){
        if (obj.year == year) return true;
        if (obj.title.toLowerCase().indexOf(title)!=-1) return true;
        return false;
    });
};
```

RELATED SAMPLE: Filtering by several criteria (OR logic)

Custom filtering rules

A filter is a set of filtering rules applied to specific content. When you specify a filter you have the possibility to set additional filtering rules for it. To set additional filtering rule for the filter you must create a custom function implementing those rule and set property **compare** of the **header**(**footer**) attribute to this function.

Custom function specified as a filtering rule takes 2 parameters:

value - a column value (the function checks values by turn);

filter - the entered in the filter value.

For example, to make DataTable filtered just by the start letter of column values, you may specify the filtering rule shown below:

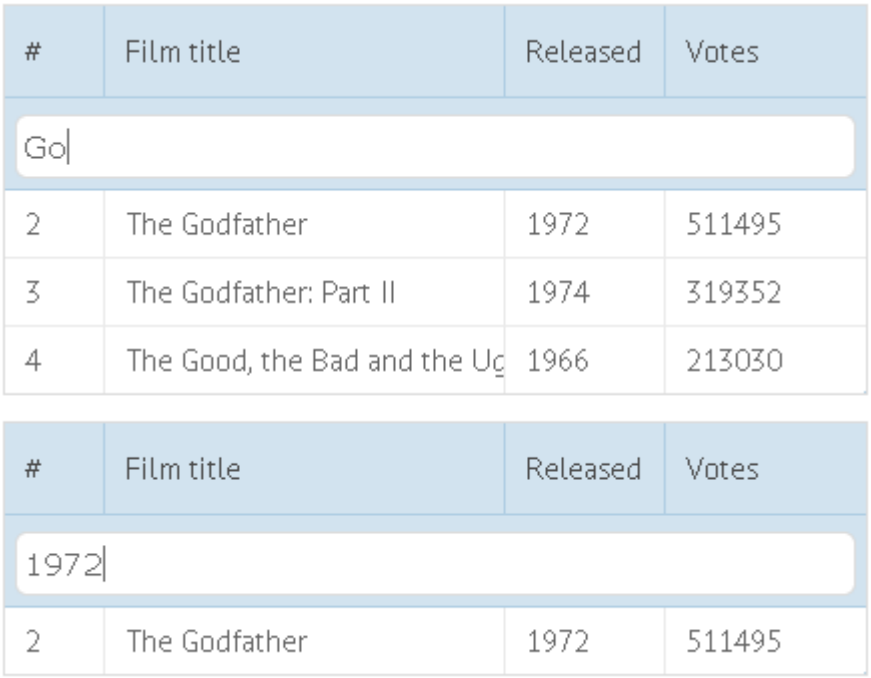
```
Filtering by the start letter of the column values

function startCompare(value, filter){
    value = value.toString().toLowerCase();
    filter = filter.toString().toLowerCase();
    return value.indexOf(filter) === ;
}

grid = new webix.ui({
    view:"datatable",
    ...
    columns:[
        { id:"title", header:["Film title",{content:"textFilter",
compare:startCompare}]},
        ...
    ]
});
```

RELATED SAMPLE: Using different filtering rules

Filtering by multiple criteria entered via one input



There is the possibility to filter DataTable by different columns using one input for this.

This can be done by:

Redefining method filterByAll();

Specifying an additional filtering rule.

```
Filtering by multiple criteria (by using method filterByAll)

grid = new webix.ui({
    view:"datatable",
    ...
    columns:[
        { id:"rank",    header:["#", { content:"textFilter", colspan:3}]
},
        { id:"title",   header:["Film title",""] },
        { id:"year",    header:["Released",""] },
    ]
});

grid.filterByAll=function(){
//gets filter values
    var text = this.getFilter("rank").value.toString().toLowerCase();
//unfilters for empty search text
    if (!text) return this.filter();

//filters using OR logic
    this.filter(function(obj){
        if (obj.year == text) return true;
        if (obj.title.toLowerCase().indexOf(text)!=-1) return true;
        return false;
    });
```

```
};
```

RELATED SAMPLE: Filtering by multiple criteria entered via one input

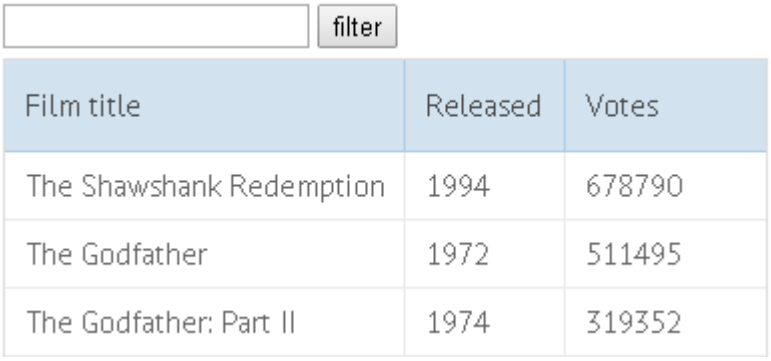
```
Filtering by multiple criteria (by specifying an additional filtering rule)

grid = new webix.ui({
    view:"datatable",
    columns:[
        { id:"rank",    header:["#", {content:"textFilter",
compare:oneForAll, colspan:3}]},
        { id:"title",   header:["Film title",""] },
        { id:"year",    header:["Release year",""]}
    ],
});

function oneForAll(value, filter, obj){
    if (obj.year.toString().toLowerCase().indexOf(filter) !== -1;) return
true;
    if (obj.title.toLowerCase().indexOf(text)!=-1) return true;
    return false;
};
```

RELATED SAMPLE: Custom filtering by multiple criteria entered via one input

HTML inputs as filters for DataTable



Setting an HTML input as datatable filter

```
Setting a Custom Filter

<input type="text" id='myfilter'> //input

grid = new webix.ui({  //component
    view:"datatable",
    columns:[
        {id:"title", ...}
    ]
});
```

```
grid.registerFilter(document.getElementById("myfilter"),
    { columnId:"title" }, //column to filter
    {
        getValue:function(node){
            return node.value;
        },
        setValue:function(node, value){
            node.value = value;
        }
});
```

RELATED SAMPLE: Filtering by multiple criteria

Custom Filtering with filter() method

In addition, the library gives you method filter() to provide fully custom filtering.

For example, if you add an input and button to the page and want to filter DataTable by clicking on it, you code may look like this:

```
Implementing a custom filter

<input type="text"><input type="button" value='filter'
onclick='filterText(this);'>

<script>
    function filterText(node){
        var text = node.previousSibling.value;
        if (!text) return grid.filter();

        grid.filter(function(obj){  //grid is a dataTable instance
            return obj.year == text;
        })
    }
</script>
```

RELATED SAMPLE: Filtering via a separate input element

Note, in the DataTable constructor you need to specify no parameters.

Find API

Datatable API allows searching for the needed records easily with the help of its find method.

Unlike filtering, it preserves the records visible but returns an array of rows that match the given criterion for further usage.

For instance, it allows highlighting the filtered data:

```
//res - array of found rows
var res = table.find(function(obj){
    return obj.title.toLowerCase().indexOf(value) != -1;
});

table.clearCss("marker", true)
for (var i = ; i < res.length; i++)
    table.addCss(res[i].id, "marker", true);
table.refresh();
```

RELATED SAMPLE: "Find" API

TreeTable specific configuration

In case of TreeTable some extra configurations can be defined to define how filters will be applied

The **filterMode** property is an object that can contain 2 attributes:

- **showSubItems** - (*boolean*) defines whether the tree must display childs of the filtered items (*true*) or the filtered items would be displayed as childs of the root node (*false*). The default value - *true*.
- **openParents** - (*boolean*) defines whether the tree will expand branches to show the found items (*true*) or stay as is (*false*). The default value - *true*
- **level** - (*number*) sets the nesting level to filter items from (one-based numbering).

```
Using the filterMode parameter

webix.ui({
    view:"treetable",
    filterMode:{
        showSubItems:false,
        level:2
    }
});
```

RELATED SAMPLE: Filtering

Math Operations

DataTable gives you an opportunity to load data stored as mathematical expressions. DataTable parses, evaluates such expressions and presents result in the related cells.

Note that this functionality won't work with the webix.env.strict enabled. By default, the flag is disabled and math is supported.

Common (storing data as formulas)

To start parse data as mathematical expressions - just set parameter math to *true* (by default, it's disabled). Columns are specified as usual.

	Country	Revenue	Expenditures	Deficit/surplus	Date
1	United States	2092000	3397000	-1305000	2010
2	Japan	1839000	2252000	-413000	2009
3	Germany	1398000	1540000	-142000	2009
4	France	1229000	1445000	-216000	2009

Activating math support and parsing data as math expressions

```
var budget = [
    ...
    {id:4, name:"France", rev:"1229000", exp:"1445000", dif:"=[:3,:2]-
[:3,:3]",
    date:"2009"}
];

grid = new webix.ui({
    view:"datatable",
    container:"box",
    columns:[
        { id:"id",   header:"#"},
        { id:"name", header:"Country"},
        { id:"rev",  header:"Revenue"},
        { id:"exp",  header:"Expenditures"},
        { id:"dif",  header:"Deficit/surplus"},
        { id:"date", header:"Date"}
    ],
    math: true, //this parameter enables math support
    data: budget
```

```
})
```

RELATED SAMPLE: Using formulas

Syntax:

Formula must start from the '=' sign.

Within formulas can be used:

- basic mathematical operators, which are: +, -, /, *, (,);
- cell referencies;
- custom functions.

You can refer to a cell in 2 ways:

- **[id, field]** - takes the value from the specified field of an item with the specified id.

```
var data = [
    { id:1, num1:"67000", num2:"15000", difference:"=[1,num1]-[1,num2]"},
    { id:2, num1:"56000", num2:"11000", difference:"=[2,num1]-[2,num2]"},
    { id:3, num1:"45300", num2:"10000", difference:"=[3,num1]-[3,num2]"}
];
```

- **[:row_ind, :col_ind]** - the indices of the row and column (zero-based numbering).

```
var data = [
    { id:1, num1:"67000", num2:"15000", difference:"=[:0,:1]-[:0,:2]"},
    { id:2, num1:"56000", num2:"11000", difference:"=[:1,:1]-[:1,:2]"},
    { id:3, num1:"45300", num2:"10000", difference:"=[:2,:1]-[:2,:2]"}
];
```

You can also combine 2 variants and use **[id, :col_ind]** or **[:row_ind, field]**.

```
var data = [
    { id:1, num1:"67000", num2:"15000", difference:"=[:0,num1]-
[:0,num2]"},
    { id:2, num1:"56000", num2:"11000", difference:"=[:1,num1]-
[:1,num2]"},
    { id:3, num1:"45300", num2:"10000", difference:"=[:2,num1]-[:2,num2]"}
];
```

You can call any **custom function** within a formula in case provided operations are not enough:

```
function mysum(a, b){ return a+b };
...
{ id:3, difference"=mysum([:0, :1], [:0,:2]) + 1 - [:0, :3]" }
```

Assigning formulas while columns configuration

To apply some formula to a column you may also use attribute **math** while column configuring. Any formula that will be set as the value of this attribute will be applied to the entire column.

```
Specifying formulas while columns configuring

var budget = [
    ...
    { id:4, name:"France ", rev:"1229000", exp:"1445000", date:"2009"}
];
grid = new webix.ui({
    view:"datatable",
    container:"box",
    columns:[
        { id:"id",   header:"#"},
        { id:"name", header:"Country"},
        { id:"rev",  header:"Revenue"},
        { id:"exp",  header:"Expenditures"},
        { id:"dif",  header:"Deficit/surplus", math:"[$r,:2] - [$r,exp]"},
        { id:"date", header:"Date"}
    ],
    math: true, //this parameter enables math support
    data: budget
})
```

RELATED SAMPLE: Applying formulas to an entire column

Syntax:

Formula is written without the '=' sign.

The **$r** keyword refers to the current row id.

The **$c** keyword refers to the name of the current column.

Within formulas can be used:

- basic mathematical operators, which are: : +, -, /, *, (,).
- cell referencies;
- custom functions.

You can refer to a cell in 2 ways:

- **[id, field]** - takes the value from the specified field of an item with the specified id (*see usage examples above*).
- **[:row_ind, :col_ind]** - the indices of the row and column (zero-based numbering).

  You can also combine 2 variants and use **[id, :col_ind]** or **[:row_ind, field]**.

You can call any **custom function** within a formula in case provided operations are not enough:

```
function mysum(a, b){ return a+b };
...
{ id:"dif", header:"", math:"mysum([$r,:2] - [$r,exp]) + 1 - [$r, :3]" }
```

Sum counter

To get the total value of some column you can use the **summColumn** counter (instead of a math formula), created especially for this need.
The counter is added to the footer of a column (can't be used in the header).

```
columns:[
    { id:"title", header:"Film title"},
    { id:"year",  header:"Year"},
    { id:"votes", header:"Votes", footer:{content:"summColumn"}}
]
```

Editing Formulas Directly in Datatable

Math formulas are editable directly in the datatable provided that:

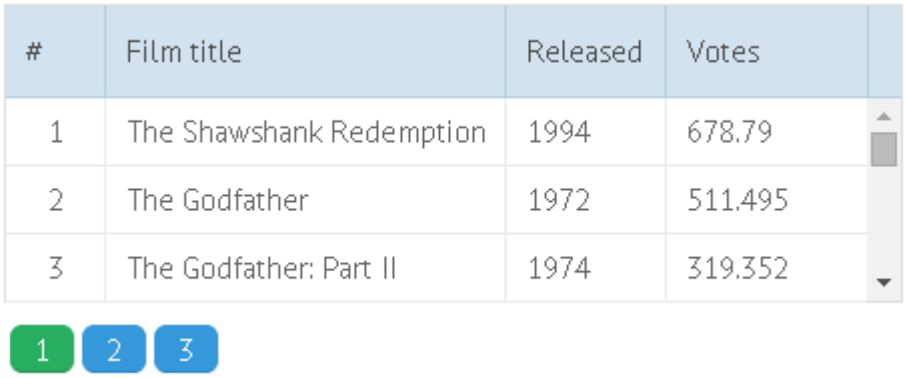
- **editMath** property is set to **true** value;
- datatable is **editable** itself;
- the column where math is used features "text" editor;
- **editAction** is specified;

```
view:"datatable",
columns:[
    { id:"diff", header: "Math", math:"[$r,:2] - [$r,exp]",
editor:"text"},
],
math: true,
editable:true,
editAction:"click",
editMath:true
```

When this property is omitted, you can still open the editor for the calculated cell value and see the result of the formula in the editor input. However, changes won't be saved, since math features higher priority.

Paging with Datatable

DataTable supports paging over the items in the data source. You enable and configure paging through the pager parameter.



Paging doesn't depend on the way you load data into DataTable: static or dynamic, external data file or database table. Datatable can display direction controls (that provide forward and backward navigation) as well as numeric controls that allow a user to move to a specific page.

Generally, you should just specify parameter pager with the required attributes.

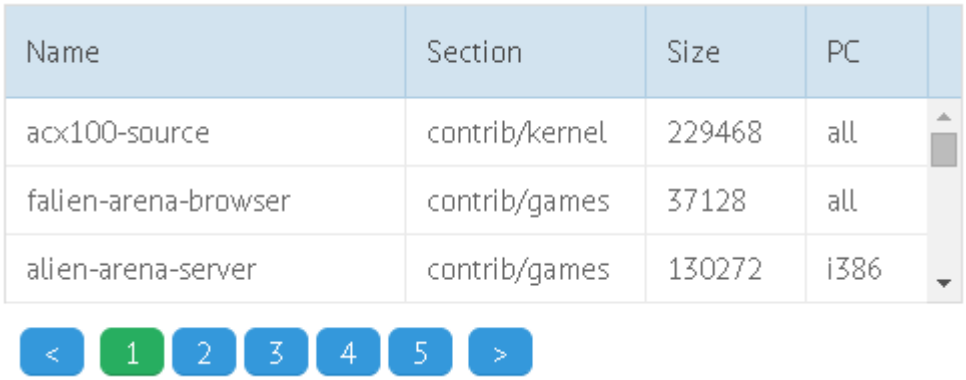
```
Basic pager configuration

grid = new webix.ui({
    view:"datatable",
    ...
    pager:{
        container:"paging_here",// the container where the pager controls
will be placed into
        size:100, // the number of records per a page
        group:5   // the number of pages in the pager
    }
})
```

RELATED SAMPLE: Static paging. Loading from an external data file

Pager controls

If you set the pager parameter, DataTable automatically adds controls for page navigation. You can customize those controls by specifying a certain template for them.



The template is set by the **template** attribute of the pager parameter and can contain the following values:

{common.first()} - the button to move to the first page.

{common.prev()} - the button to move to the previous page.

{common.next()} - the button to move to the next page.

{common.last()} - the button to move to the last page.

{common.pages()} - numeric controls to move to a specific page.

Depending on what you want to display on the paging area, add the related items to your template.

```
Using templates in pager

grid = new webix.ui({
    view:"datatable",
    ...
    pager:{
        template:"{common.prev()} {common.pages()} {common.next()}",
        container:"paging_here",
        size:10,
        group:5
    }
});
```

RELATED SAMPLE: Using templates for pagers

Generally, using the mentioned values, you can set the following combinations for DataTable:

- Numeric (default)
- Next/Previous
- Next/Previous/First/Last
- Numeric/First/Last
- Numeric/Next/Previous/First/Last

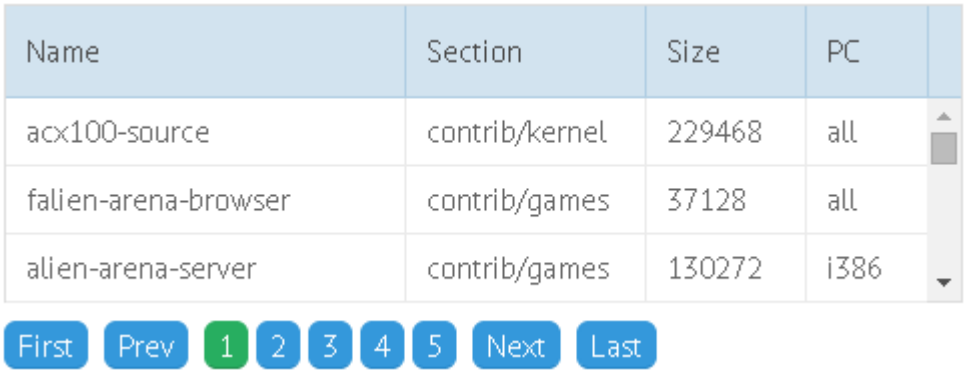
The first combination is set by default. Other available ones can be achieved with the help of templates (see the table below).

Table 1 Pager controls combinations

Mode	Image	Related template
Numeric		default mode, doesn't require specifying a separate template
Next/Previous		`template:" {common.prev()} {common.next()} "`
First/Previous/Next/Last		`template:" {common.first()} {common.prev()} {common.next()} {common.last()}"`
Numeric/First/Last		`template:" {common.prev()} {common.pages()} {common.next()}"`
Numeric/Next/Previous/First/Last		`template:" {common.first()} {common.prev()} {common.pages()} {common.next()} {common.last()}"`

Customizing pager labels

You can set some text instead of arrows for the Next, Previous, First and Last buttons.




```
Specifying custom labels for pager controls

webix.locale.pager = {
    first: "First",// the first button
    last: "Last",// the last button
    next: "Next",// the next button
    prev: "Prev"// the previous button
};

grid = new webix.ui({
    view:"datatable",
```

```
    ...
    pager:{
        template:"{common.first()} {common.prev()} {common.pages()}
                {common.next()} {common.last()}",
        container:"paging_here",
        size:10,
        group:5
    }
})
```

It's not obligatory to redefine all the mentioned buttons at once, you can redefine just a part of them. In this case, **webix.locale.pager** will contain just the required buttons.

RELATED SAMPLE: Pager localization

The size of a page

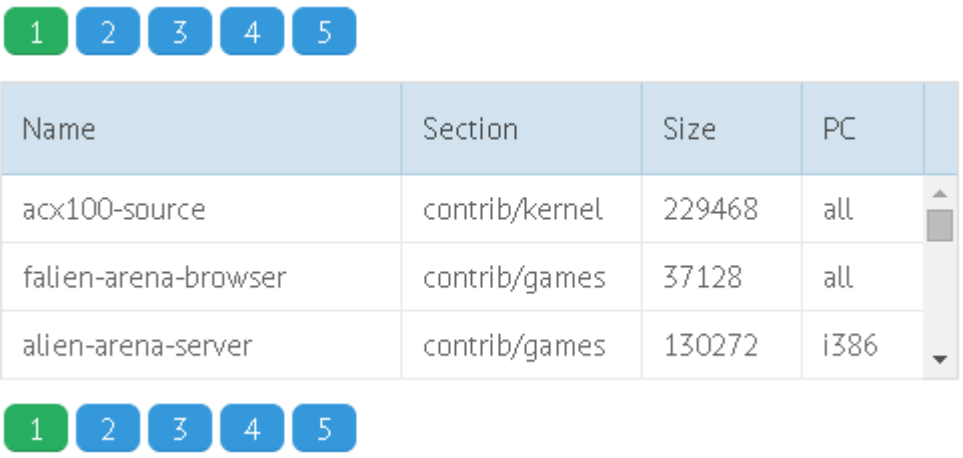
The size of a page is a number of items to display at once on a page. This number is set by the **size** attribute:

```
Setting the number of records on a page

pager:{
    size:100,  // the number of records on a page
    ...
}
```

Multiple pagers on a page

Several pagers can be on one and the same page.



To add an additional pager to a page you should call method clone() and set there configuration of your new pager using the same attributes you use while defining the pager parameter:

```
Specifying 2 pagers on a page

grid = new webix.ui({
```

```
    view:"datatable",
    ...
    pager:{..}
});

grid.getPager().clone({
    container:"paging_here",
    size:10,
    group:5
});
```

RELATED SAMPLE: Defining several pagers on a page

Selection with Datatable

In DataTable you can specify different types of selection depending on your needs.

Available modes are:

- *cell*;
- *row*;
- *column*;
- *multi cell*;
- *multi row*;
- *multi column*;
- *block selection*.

The desired type is set through parameters select, multiselect and blockselect.

The table below shows which combination of parameters you should use for each type.

Table 1 Types of selection

Type of selection	Related parameters
single cell	`select:"cell"`
single row	`select:"row"`
single column	`select:"column"`
multiple cells	`select: "cell", multiselect: true`
multiple rows	`select: "row", multiselect: true`
multiple columns	`select: "column", multiselect: true`
visual block selection	`select: "cell", multiselect: true, blockselect: true`

For example, to set in your DataTable multi row selection, you should use the following code:

```
Enabling multi row selection

grid = new webix.ui({
    view:"datatable",
    ...
    select:"row",
    multiselect:true
})
```

RELATED SAMPLE: Multi-row selection

Related methods

Table 2 Selection methods

Method	Description
clearSelection	clears selection
getSelectedId	returns the selected cells
mapSelection	executes some custom method for all cells in the currently selected block
moveSelection	moves selection to the specified direction
select	selects the specified element
selectRange	selects the specified range of elements
unselect	cancels selection of the specified element

Related events

Table 3 Selection events

Event	Description
onAfterSelect	fired after a cell is selected
onAfterUnselect	fired after a cell is unselected
onBeforeSelect	fired before a cell is selected
onBeforeUnSelect	fired before a cell is unselected
onSelectChange	fired when selection is changed in DataTable

DataTable Sorting

DataTable allows you to sort data rows on the client side. There are 2 ways to invoke sorting in the table:

- By a single click on the header of a column with the enabled **sort** attribute;
- By api call (can be called from some event or action, i.e button click or page load) of the **sort()** method.

Sorting by a click on the header

When you click on the header, DataTable starts to display a special control indicating which column the table is currently sorted by and the direction of this sorting (ascending or descending). Each next click on the same header will reverse the sorting direction.

	Film title	Released	Votes
6	12 Angry Men	1957	164558
5	Pulp fiction	1994	533848
2	The Godfather	1972	511495
3	The Godfather: Part II	1974	319352
4	The Good, the Bad and the Ug	1966	213030
1	The Shawshank Redemption	1994	678790

Columns can have different type of content (numbers, strings, dates) and each type of content requires its specific way of sorting.

Sorting Modes for Different Data Types

For this purpose, DataTable provides 4 sorting types to ensure correct sorting of columns:

- *int*;
- *date*;
- *string*;
- *string_strict* (case-sensitive 'string').

To enable sorting and assign the appropriate sorting type to a column, you should specify an attribute **sort** among the column's parameters and set it to some of types.

```
Sorting configuration

grid = new webix.ui({
        view:"datatable",
```

```
        ...
        columns:[
            { id:"title", header:"Film title",    sort:"string"},
            { id:"year",  header:"Release year", sort:"int"}
        ]
});
```

RELATED SAMPLE: Sorting. Using built-in means

The above mentioned sorting modes work with column values defined by their **ID** attributes and ignore values either defined by column templates or provided by column options.

In the sample column below column titles are sorted, yet categories are displayed (column template overrides column ID during rendering):

```
{ id:"title",   template:"#cat_id#", header:"Category ID" }
```

In the sample column below option IDs are sorted, yet option values are displayed:

```
{ id:"cat_id",  header:"Category", collection:[
    { id:1, value:"Crime"}, { id:2, value:"Thriller" }]
}
```

Sorting by visible text

Sorting by visible text is enabled width the help of **"text"** sorting mode that switches on **string** comparison for the values actually displayed in datatable columns.

It takes into account column values defined by **templates** and collection **values** rather than IDs:

```
columns:[
    { id:"title",template:"#cat_id#", header:"Category ID", sort:"text" },
    { id:"cat_id",  header:"Category",sort:"text", collection:[
        { id:1, value:"Crime"}, { id:2, value:"Thriller" }]
    }
]
```

RELATED SAMPLE: Sorting. Using built-in means

Triggering Server-side Sorting on Client-side

It's possible to issue a request to server to sort data in backend and reload the sorted dataset to the datatable:

The option is enabled by **server** sorting mode:

```
view:"datatable",
columns:[
    { id:"package", header:"Name", sort:"server"},
    {..}
],
url:"data.php"
```

RELATED SAMPLE: Datatable: Serverside Filtering and Sorting

Now, header clicking will trigger a serverside GET request with the following parameter: *sort[package]=desc* (data.php?sort[package]=desc), which allows sending:

- **operation name** (sort);
- **data property** to sort by (here: package);
- **sorting order** (here: desc).

The new data will be loaded to the datatable and replace the initial dataset.

If serverFilter is enabled for this column, the data will be both filtered and sorted on serverside before returning to client.

Custom sorting functions

If you want to apply custom sorting behaviour, you can define the related logic in a function and set this function as the value of attribute **sort**.

This function will be called for each pair of adjacent values and return 1,-1 or 0:

- **1** - an object with the first value in pair must go before the second one;
- **-1** - the second object goes before the first one;
- 0 - the order of both objects doesn't change.

The function can be defined in a general way:

```
Using custom functions for sorting DataTable

function sortById(a,b){
        a = a.rank;
        b = b.rank;
        return a>b?1:(a<b?-1:);
}

webix.ui({
        view:"datatable",
        ...
```

```
        columns:[
            { id:"title",   header:"Film title", sort:sortByParam },
            ...
        ]
})
```

RELATED SAMPLE: Sorting Datatable. Using custom sorting functions

Programmatic sorting

You should use method sort() to invoke sorting on some event or action, i.e button click or page load. You can't use this method if you want to sort DataTable by a click on the header.

```
Sorting DataTable on the button click

<input type='button' onclick='grid.sort("#title#", "asc", "string");'>

<script type="text/javascript" charset="utf-8">
        grid = webix.ui({ view:'datatable', ...})
</script>
```

RELATED SAMPLE: Sorting Datatable. Using sorting methods

Controlling header sorting mark

You can show and hide the sorting sign (^/V) by calling the markSorting() method with the following parameters:

- **column_id** - id of the column you want to draw a sorting sign for;
- **dir** - sorting direction.

When used with no arguments, the method removed all the sorting signs from the datatable headers.

```
Cancelling sorting

grid = webix.ui({ view:"datatable", ...})

grid.markSorting("title", "asc");
```

RELATED SAMPLE: Sorting Datatable. Using sorting methods

DataTable Editing

DataTable is an editable component, i.e. users have the possibility to edit it manually.

Generally, to make the table editable you should set parameter editable to true.

```
Making DataTable editable

dtable = new webix.ui({
  view: "datatable",
  ...
  editable:true
});
```

RELATED SAMPLE: Basic use of editors

2 main points related to editing are:

- The types of available editors.
- The action on which the cell(s) editor(s) will be opened.

Editors

The library provides 10 predefined editors:

- **text**, **inline-text**, **password**;
- **checkbox**, **inline-checkbox**;
- **select**, **combo**, **richselect**;
- **multiselect** (**Webix Pro** edition only);
- **date**;
- **color**;
- **popup**

All of them are described in detail in the corresponding article.

To assign the needed editor type to a column, you should specify attribute **editor** of the columns parameter and set it to some of types.

```
Specifying the editor for a column

columns: [
  { id:"title", header:"Film title", editor:"text"}
]
```

RELATED SAMPLE: Basic use of editors

Creating a custom editor

To create a custom editor, you should set the following methods to it:

- **focus()** - sets the focus to the input with the editor.
- **getValue()** - gets the value of the editor.
- **setValue()** - sets the value of the editor.
- **render()** - renders the editor.

webix.editors = { "myeditor": { focus: function () {...} getValue: function () {...}, setValue: function (value) {...}, render: function () {...} } };

The functions are defines with the help of the following inner properties:

- **this.node** - HTML tag of the editor. Appears after render() execution;
- **this.value** - the initial value of an input. Appears after setValue(); execution;
- **this.config** - editor configuration;
- **this.popup** - popup ID (is used for editors with popup windows)

After your editor is in the collection, you can set it for a column using the familiar technique:

```
Adding a new editor and setting it for the column

webix.ui({
  view: "datatable",
  columns: [
    ...
    { id: "title", header: "Film title", editor: "myeditor"}
  ]
});
```

Find more info on Data Editors in the dedicated chapter of the manual.

Setting the action to open editors on

Generally, to set the action on which editors will be opened you should use the editaction property.

Default actions

By default, editors are opened on a single click. To make opening editors occured on a double click, you should set editaction to *dblclick* as in:

```
Opening editors on a double click

webix.ui({
  view: "datatable",
  editable: true,
  editaction: "dblclick"
```

```
  ...
});
```

Custom actions

Using some other action/keyboard key for opening requires some more treat cause you should provide all 'open-edit-close' logic by yourself.

To open editors on some keyboard key press you should do the following:

- set the editaction property to value 'custom'.
- use the UIManager mixin (method addHotKey) to add a hot key and define the processing logic.

```
Opening editors on the Enter key press

var mytable = new webix.ui({
  view: "datatable",
  ...
  editable: true,
  editaction: "custom"
});


webix.UIManager.addHotKey("enter", function(view){
    var pos = view.getSelectedId();
    view.edit(pos);
}, mytable);
```

RELATED SAMPLE: Basic use of editors

To open editors on some action you should do the following:

- set the editaction property to value 'custom'.
- specify the action (through the related event) and define the processing logic (in the event handler function). You can do this with the help of the on parameter or the attachEvent method.

```
Opening editors on cell selecting

var mytable = new webix.ui({
  view: "datatable",
  ...
  editable:true,
  editaction: "custom",
  select: "cell",
  on:{
    onAfterSelect: function (data,prevent) {
      this.editCell(data.row, data.column);
    }
  }
```

```
})
```

Use case: editing the entire row, column

To open for editing all cells of the specified row/column make the following:

Set the editaction property to value 'custom'.

Use the on parameter to specify the action and define the processing logic:

- as an event to handle - onItemClick.
- editRow, editColumn enables the edit mode for the specified row/column (opens all editors)

```
Opening editors in all cells of the row, column

//editing the entire row
var table1 = new webix.ui({
  view: "datatable",
  ...
  editable: true,
  editaction: "custom",
  on: {
    onItemClick: function (id) {
      this.editRow(id);
    }
  }
});

//editing the entire column
var table2 = new webix.ui({
  view:"datatable",
  ...
  editable: true,
  editaction: "custom",
  on: {
    onItemClick: function(id){
      this.editColumn(id);
    }
  }
})
```

RELATED SAMPLE: Opening multiple editors

Apart from row and column editing, datatable API offers the possibilities to

- programmatically open editor in the next cell of the row (provided that it is editable) - **editNext()** method;
- move focus to the active editor (if any) - **focusEditor()** mehtod;
- close the editor without saving changes - **editCancel()** method;

- close the editor while saving changes - **editStop()** method.

Editing via a Bind Form

Webix components including datatable can be bound to another component to ensure **synchronous changing** of their **data**. For instance, a simple function can be used to bind a form to a grid, which allows to edit datatable data:

- clicking the datatable row will trigger form filling;
- then you edit data in the form;
- form saving will send changed data back to the grid.

```
$$('form1').bind('datatable1');
```

RELATED SAMPLE: Editors. Bind Form

Note that **name** attributes of form fields coincide with **ids** of datatable columns.

More info about data binding.

The Tab key navigation

By default, the Tab(Tab+Shift) key(s) allows navigating within editors defined in the table.

Main points:

- The Tab key press closes the current editor and opens the next one. If the current editor is last in a row, the next one will be opened in the next row (from the beginning).
- The Tab+Shift keys press closes the current editor and opens the previous one. If the current editor is first in a row, the next one will be opened in the previous row (from the end).
- If a cell doesn't have the editor specified, it won't be included to navigation.
- The opened editor always resides in the visible area. It means that if you click Tab and the cell is out of the visible area then the table will be scrolled to show it (regardless of the mode set).
- When multiple editors are opened in the table at the same time, Tab/Tab+Shift navigates within the opened editors.

RELATED SAMPLE: Tab navigation and scrolling

RELATED SAMPLE: Opening multiple editors

Checkbox and Radio in DataTable

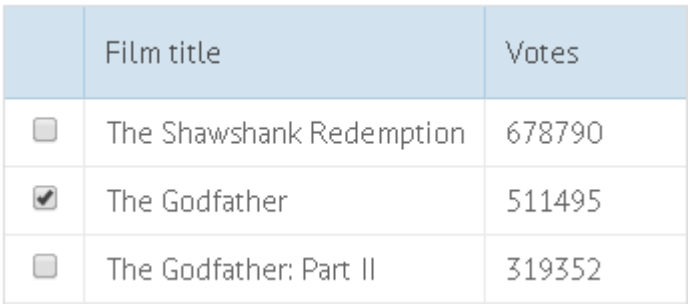
DataTable allows you to add any HTML elements inside of it (using the **template** attribute of the columns parameter).
Besides, the library provides ready-to-use solutions for popular checkbox and radio controls to save your time.

The controls are specified through the same **template** attribute of the columns parameter.

```
webix.ui({
    view:"datatable",
    ...
    columns:[
        { id:"ch1", header:"", template:"{common.checkbox()}"},
        { id:"ra1", header:"", template:"{common.radio()}"}
    ]
});
```

checkbox

A standard checkbox.



	Film title	Votes
☐	The Shawshank Redemption	678790
☑	The Godfather	511495
☐	The Godfather: Part II	319352

To define a checkbox column, set the **template** attribute to "*{common.checkbox()}*":

```
Creating a checkbox column

columns:[
        { id:"ch1", header:"", template:"{common.checkbox()}"}
        ...
]
```

RELATED SAMPLE: Checkbox and radio in DataTable

radio

A radio button.

	Film title	Votes
○	The Shawshank Redemption	678790
◉	The Godfather	511495
○	The Godfather: Part II	319352

To define a radio column, set the **template** attribute to "*{common.radio()}*":

```
Creating a radio column

columns:[
        { id:"ch1", header:"", template:"{common.radio()}"}
        ...
]
```

RELATED SAMPLE: Using 'inline' editors

Custom checkbox

Shortly,a technique of creating a custom checkbox is the following:

Create a function defining a new checkbox.

Set it as the value of the **template** attribute.

Set the **checkboxRefresh** parameter to *true* to force the table getting refreshed after each change of the checkbox value.

Take a look at the image below and learn how to create the checkboxes displayed in the first column.

	Film title	Votes
NO	The Shawshank Redemption	678790
YES	The Godfather	511495
NO	The Godfather: Part II	319352

1. Create a function that will define your custom checkbox.

The function will accept 3 parameters and be called for each data item:

- **obj** - (*object*) the data item object
- **common** - (*object*) the predefined checkbox template
- **value** - (*boolean*) the value of the checkbox. *true*, if the box is checked. Otherwise, *false*. Initially, the parameter is *null* and take the boolean value just when you check the checkbox.

Please, pay your attention at a very important thing - the 'webix_table_checkbox' css class.

This very class forces a standard DIV to behave as a checkbox. Therefore, defining a look of your checkbox don't forget to apply the 'webix_table_checkbox' css class to it.

```
Function defining a custom checkbox

function custom_checkbox(obj, common, value){
    if (value)
        return "<div class='webix_table_checkbox checked'> YES </div>";
    else
        return "<div class='webix_table_checkbox notchecked'> NO </div>";
};

<style type="text/css">
    .checked{
        color:green;
        font-weight: bold; cursor:pointer;
    }
    .notchecked{
        color:red;
        font-weight:bold; cursor: pointer;
    }
</style>
```

2. Set the 'template' attribute of the related column to the name of the function.

Once you do this, the column will display data returned by the function defined above.

```
Setting custom template for the column

columns:[
        { id:"ch1",     header:"",  template:custom_checkbox}
        ...
]
```

3. Specify the 'checkboxRefresh' parameter and set it to *true*.

The checkboxRefresh parameter answers for repainting the related data record. When it set to *true*, each time the checkbox value is changed DataTable will repaint the related record.

```
Making DataTable getting refreshed after each change of the checkbox value

webix.ui({
    view:"datatable",
    ...
    columns:[
        { id:"ch1",     header:"",  template:custom_checkbox},
        ...
    ],
    checkboxRefresh:true
```

```
});
```

RELATED SAMPLE: Custom radio and checkbox in DataTable

Custom radio

Shortly,a technique of creating a custom radio is the following:

Create a function defining a new radio.

Set it as the value of the **template** attribute.

Take a look at the image below and learn how to create the radio buttons displayed in the first column.

	Film title	Category	Votes
	The Shawshank Redemption	Thriller	678790
	The Godfather	Crime	511495
	The Godfather: Part II	Crime	319352

1. Create a function that will define your custom radio.

The function will take 3 parameters and be called for each data item:

- **obj** - (*object*) the data item object
- **common** - (*object*) the predefined radio template
- **value** - (*boolean*) the value of the radio. *true*, if the radio is checked. Otherwise, *false*. Initially, the parameter is *null* and take the boolean value just when you check the radio.

Please, pay your attention to a very important thing - the 'webix_table_radio' css class.
This very class forces a standard DIV to behave as a radio. Therefore, defining a look of your radio don't forget to apply the 'webix_table_radio' css class to it.

```
Function defining a custom radio

function custom_radio(obj, common, value){
    if (value)
        return "<div class='webix_table_radio checked'></div>";
    else
        return "<div class='webix_table_radio notchecked'></div>";
};

<style type="text/css">
    .checked{
        background:green;
        width:16px; height:16px; margin-top:3px;
    }
```

```
    .notchecked{
        background:orange;
        width:16px; height:16px; margin-top:3px;
    }
</style>
```

2. Set the 'template' attribute of the related column to the name of the function.

As you specify the template attribute in such a way, the column will display data returned by the function defined above.

```
Setting custom template for the column

webix.ui({
    view:"datatable",
    ...
    columns:[
        { id:"ch1",     header:"",  template:custom_radio},
        ...
    ]
});
```

RELATED SAMPLE: Custom radio and checkbox in DataTable

Integration

In this chapter you will see how to integrate DataTable with jQuery.

jQuery integration

DataTable can be easy integrated with jQuery.

We want to show you 2 most 'interesting' situations of such integration:

- Initializing of a standard DataTable;
- Creating DataTable from an HTML table.

Initializing jQuery DataTable

A standard DataTable using jQuery can be initialized as follows:

```
Integration with jQuery

<div id="box" class="cont"></div>
...

$(".cont").webix_datatable({
    columns:[{...}],
    select:"cell",
    data:some_data
});
```

RELATED SAMPLE: Integration with jQuery

Creating DataTable from an HTML table

To turn an HTML table into DataTable you should call method **datatable()** for it:

```
Creating DataTable from an HTML table

<table>
        <tr>
            <td width="40">id</td>
            ...
        </tr>
        ...
</table>

<script>
 $("table").webix_datatable();
</script>
```

RELATED SAMPLE: Integration with jQuery. Turning HTML Table into DataTable

Keyboard navigation

We can single out 2 types of navigation in DataTable:

- Navigating the selection
- Navigating the active editor

When one of the types is available, the other is blocked, i.e. when you open some cell editor you won't navigate the selection any more but can navigate the active editor within cells available for editing. Once no editors is open, you can again navigate the selected cell (row, column).

Selection navigation

Initially, the selection navigation is disabled in DataTable.

To activate it, set the navigation parameter to *true*.

```
Activating the selection keyboard navigation in DataTable

webix.ui({
    view:"datatable",
    navigation:true,
    ...
})
```

Table 1 Keyboard commands for navigating the selection

Command	Description
Up arrow key	moves one cell (row) to the up
Down arrow key	moves one cell (row) to the down
Left arrow key	moves one cell (column) to the left
Right arrow key	moves one cell (column) to the right
Home	moves to the top left cell(row, column) in the grid
End	moves to the bottom right cell(row, column) in the grid
PGUP	scrolls the table up
PGDN	scrolls the table down

RELATED SAMPLE: Navigation in DataTable

Active editor navigation

Active editor navigation is navigating within cells available for editing: closing the currently active editor and opening the editor in the next/previous cell.

The navigation is activated once you enable editing in the table.

```
dtable = new webix.ui({
        view:"datatable",
        ...
        editable:true
});
```

Table 2 Keyboard commands for navigating the active editor

Command	Description
Tab	closes the current editor and openes the next one
Tab + Shift	closes the current editor and openes the previous one

You can read more about editing and active editor navigation in article DataTable Editing.

RELATED SAMPLE: The Tab key navigation

Drag-and-drop with Datatable

The library supports drag-and-drop within a table or several tables on the page.

Moving rows

To enable the drag-and-drop support for rows, set the drag parameter to *true*.

```
Enabling the drag-and-drop support for rows

webix.ui({
        view:"datatable",
        ...
        drag:true
});
```


To drag/drop rows between several tables you should enable the drag parameter in each of them.

RELATED SAMPLE: Drag-and-Drop between tables

The 'order' mode

There is a special drag-and-drop mode - 'order'.

In such a mode, rows can't be dragged out of the table boundaries. Also, while dragging an item the remaining items are automatically rearranged so that when you drop the item to the final destination the remaining items won't need to do any rearrangements.

To enable the d-n-d support and activate the 'order' mode, just set the drag parameter to *order*.

```
Activating the 'order' mode for rows

webix.ui({
        view:"datatable",
        ...
        drag:'order'
});
```

RELATED SAMPLE: The 'order' mode for rows

Moving columns

To enable the drag-and-drop support for columns, set the dragColumn parameter to *true*.

```
Enabling the drag-and-drop support for columns

webix.ui({
    view:"datatable",
    ...
    dragColumn:true
});
```


To drag/drop columns between several tables you should enable the dragColumn parameter in each of them.

RELATED SAMPLE: Drag-and-Drop support for columns

The 'order' mode

There is the 'order' mode for the table columns.

In that mode, columns can't be dragged out of the table boundaries. Also, while dragging a column the remaining columns are automatically rearranged so that when you drop the column to the final destination the remaining columns won't need to do any rearrangements.

To enable the d-n-d support and activate the 'order' mode for columns, just set the dragColumn parameter to *order*.

```
Activating the 'order' mode for columns

webix.ui({
    view:"datatable",
    ...
    dragColumn:'order'
});
```

RELATED SAMPLE: The 'order' mode for columns

Advanced

Information stated in the Common part is enough in most cases. The current part should be used just if you want to customize existing drag-and-drop behaviour.

The part covers the following topics:

- Custom text of the dragging items;
- Denying dragging of specific items;
- Denying dropping to specific positions;
- Switching between 'move' and 'copy' behaviors;

- Limiting the draggable area to a certain element;
- Auto scrolling during drag-n-drop.
- Related events.

Custom text of the dragging items

To redefine the text displaying for the dragging item(s), you should use the onBeforeDrag event.
The desired template is set through the **context.html** property. You can use any HTML specifying the value for the property.

```
Displaying the number of the dragging items

dtable.attachEvent("onBeforeDrag", function(context, ev){
    context.html = context.source.length+" item(s)";
});
```

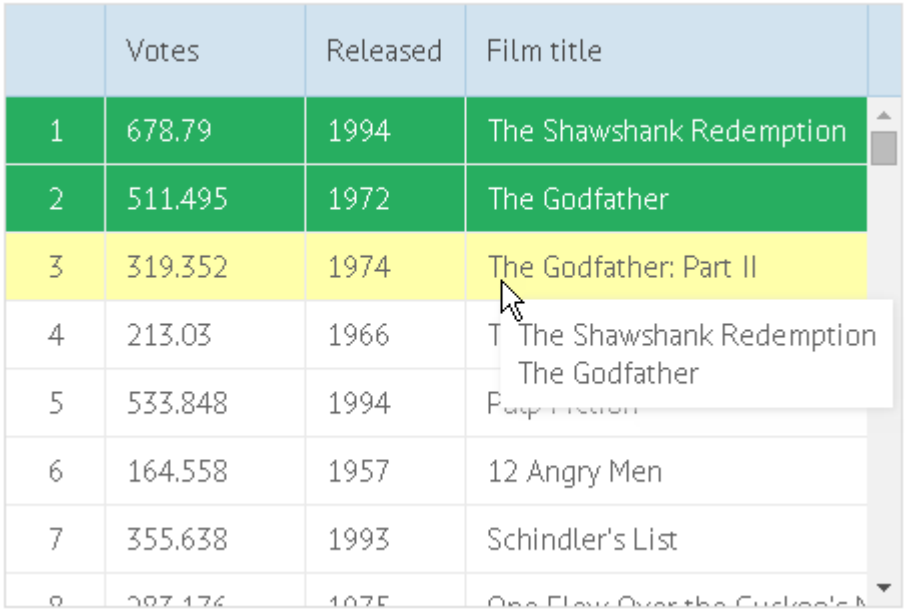
Some more use cases:

- A fixed image and the id of the dragged row:

```
Displaying the dragged row id and some image

dtable.attachEvent("onBeforeDrag", function(context, ev){
    context.html = "<img src='some.gif'></img> " + context.start;
});
```

- The text of the dragged items (in case of multiselect selection):




```
Displaying the title property of dragged items

dtable.attachEvent("onBeforeDrag", function(context, ev){
    context.html = "";
    for (var i=; i< context.source.length; i++){
        context.html += context.from.getItem(context.source[i]).title +
```

```
"<br>" ;
    }
});
```

RELATED SAMPLE: Using events for customizing dnd

Denying dragging of specific items

To deny dragging of specific items you can use the onBeforeDrag event.

```
Denying dragging even items

dtable.attachEvent("onBeforeDrag", function(context, ev){
    if(dtable.getIndexById(context.start)%2==){
        return false;      //denies dragging if the item index is even
    }
    return true;           //allows dragging in any other case
});
```

From now on, each time d-n-d is started, the item index will be checked, and the row can be dragged only if its index isn't even.

You can use the addCss method to mark items and hasCss to check whether an item has a certain marking.

Denying dropping to specific positions

To deny dropping to some positions, you can use the onBeforeDrop event and return false each time you want to block the operation.

Let's assume you specify a table on the page as below and want to deny dropping to the position of an item with 'id=2':

```
Denying dropping to a specific position

var dtable = new webix.ui({
    view:"datatable",
    ...
    data:[
        { id:1, title:"The Shawshank Redemption", year:1994},
        { id:2, title:"The Godfather",            year:1972},
        { id:3, title:"The Godfather: Part II",   year:1974}
    ]
});
dtable.data.addMark(2,"type", false,"b"); // adds a flag (type='b') to an
item with id=2

dtable.attachEvent("onBeforeDrop", function(context, ev){
```

```
//if an item has flag type='b', denies dropping to the position of this
item
        if(dtable.data.getMark(context.target,"type")=='b') return false;
        return true;
});
```

Switching between 'move' and 'copy' behaviors

By default, rows are moved (not copied) while d-n-d.

To set the 'copy' behaviour, use the onBeforeDrop event. Specify inside the copy logic and return *false* to prevent drag-and-drop processing:

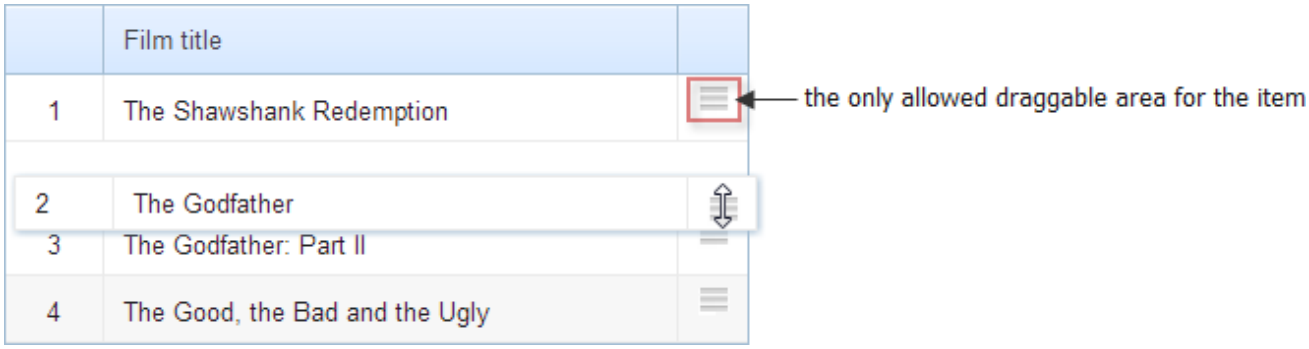
```
Setting the 'copy' behaviour for d-n-d

dtable.attachEvent("onBeforeDrop", function(context, ev){
    for (var i=; i< context.source.length; i++){
context.from.copy(context.source[i],context.start,this,webix.uid());
    }
    return false;
});
```

RELATED SAMPLE: Using events for customizing dnd

Limiting the draggable area to the specified element

You can provide a behaviour when the user could drag items only by the certain specified areas.



To achieve such a behaviour, just follow this plan:

- Specify an element you want an item to be dragged by.
- Set this element as the target object for the onBeforeDrop event (the target object is an object on which the event will occur).

```
.webix_drag_handle{
    background-image:url(./handle.png);
    background-repeat: no-repeat;
}
```

```
var dtable = new webix.ui({
    view:"datatable",
    //...
    columns:[
        { id:"rank",    header:"", css:"rank" },
        { id:"title",   header:"Film title" },
        { id:"year",    header:"Released"},
        { id:"drag",    header:"", template:"<div
class='webix_drag_handle'></div>" }
    ],
    on:{
        onBeforeDrag:function(data, e){
                return (e.target||e.srcElement).className ==
"webix_drag_handle";
        }
    }
});
```

RELATED SAMPLE: Limiting the draggable area for items (dragging rows)

RELATED SAMPLE: Limiting the draggable area for items (dragging columns)

Scrolling during Drag And Drop

By default datatable will scroll self automatically when drag is near the top or bottom border. This behavior can be disabled by using dragScroll configuration property

```
webix.ui({
    view:"datatable",
    dragScroll:false
});
```

By default drag scroll works only for vertical scroll, you can enable it for both x and y directions by using

```
webix.ui({
    view:"datatable",
    dragScroll:"xy"
});
```

Related events

The following events are generated while d-n-d process:

- onBeforeDrag - fires before the mouse button is pressed and moved over a draggable item
- onBeforeDragIn - fires before a dragged element is moved over the droppable area
- onDragOut - fires when a dragged element is moved outside of the droppable area
- onBeforeDrop - fires before a dragged element is released over the droppable area
- onAfterDrop - fires after drag-n-drop was finished

In the default scenario there is no need to use any of the mentioned events, because DataTable will process all the operations on its own. But sometimes default behavior needs customizing - this is the time when the mentioned events can be used.

Data export to PDF, Excel

Datatable and Treetable data can be exported to the following formats:

- PDF;
- Excel.

The components don't require any special configurated to be exported.

For the treetable it is possible to add indentations for tree nodes:

```
// "value" column will be indented in the exported file
view:"treetable",
columns:[
    { id:"id"},
    { id:"value", exportAsTree:true },
    { id:"chapter"}
]
```

RELATED SAMPLE: Export to PDF, Excel

After that all you need is to apply an appropriate export method to this datatable or treetable.

Export to PDF

To export data from datatable or treetable into a PDF document you need to call method exportToPDF. For example, if you add a button by clicking on which the grid will start exporting, then its code will look like this:

```
<input type="button" value="Get as PDF" style='width:400px;margin:25px;'
onclick="grid.exportToPDF();">
```

RELATED SAMPLE: DataTable Export to PDF

Above solution uses online data export service, but you can configure a local copy as well - you can download and install the Export package. It must be unpacked inside the root of your web server.

The latest packages for installation can be found here:

- PHP package
- Java package
- ASP .NET package

The code will look like this:

```
grid.exportToPDF("generate.php"); //provide the correct relative path
```

Export to Excel

To export data from datatable/tretable into an Excel document you need to call method exportToExcel. For example, if you add a button by clicking on which the grid will start exporting, then its code will look like this:

```
<input type="button" value="Get as PDF" style='width:400px;margin:25px;'
onclick="grid.exportToExcel();">
```

RELATED SAMPLE: Export to Excel

Above solution uses online data export service, but you can configure a local copy as well - you can download and install the Export package. It must be unpacked inside the root of your web server.

- PHP package
- Java package
- ASP .NET package

The code will look like this:

```
grid.exportToExcel("generate.php"); //provide corrent relative path
```

Customizing Export

Both exportToPDF and exportToExcel return data from **all currently visible columns** of your datatable without any data formatting (as it is in the dataset). However, you may need to get data from other fields as well as restrict the number of fields in the export file, or apply some format to data values.

All customizing happens within the needed export function:

```
//in case you use online service
grid.exportToExcell(null, {..setting custom values..});

//in case you use local version of export script
grid.exportToExcell("generate.php", {..setting custom values..});
```

Export API allows

- setting **columns** you'd like to see in export (including **ID** column). Here column **names** should be specified:

```
grid.exportToExcel(null, {
    id:true, // note where id export is set
    columns:{
        customer:true,
        region_id:true,
        country_id:true
    }
```

```
});
```

- defining a custom **template** and **width** for data from specified column:

```
grid.exportToPDF(null, {
    id:true,
    columns:{
        customer:{
            template:"#customer#",
            width:500
        }
    }
});
```

The column will be rendered with the stated template and width, which may differ from datatable parameters.

- **including extra field** into export by forming them right within the export function:

```
grid.exportToPDF(null, {
    columns:{
        custom1:{
            template:function(obj){
                return "Year " + obj.year;
            },
            width:50,
            header:[{ text:"Date" }]
        },
        //other columns
        title:true
    }
});
```

"Custom1" (any name can be used) receives data from **year** field even if it is not seen in this datatable but is present in its dataset. The field is rendered with **width**, **template** and **header** that will be the header of the corresponding column in the export table.

RELATED SAMPLE: DataTable - Export Configuration

Configuration

You can define how datatable will look in PDF or Excel document after it is exported, configuring available options in the server side code. For these needs, please download the local script version.

In addition, you gain extra security with local scripts as well as get rid of a text that shows download source.

Events Handling with DataTable

You can assign a custom behavior to different events of DataTable by using its event handling system.

There are 2 ways to add a handler to the event:

- through the method attachEvent();
- through the parameter on.

Event names are case-insensitive

Method attachEvent()

You can attach several handlers to the same event and detach them using two respective methods:

```
A general way to attach/detach the event handler

// to attach event
var myEvent = $$("dataTableId").attachEvent("onItemClick", function () {
  // event handler code
})

// to detach event
$$("dataTableId").detachEvent(myEvent);
```

Parameter 'on'

With the help of parameter on you can attach any event(s) to DataTable. But you can't detach them later.

```
Attaching the event handler through parameter 'on'

webix.ui({
  view: "dataTable",
  ...
  on: {"itemClick": function () {alert("item has just been clicked");}}
);
```

Cancelable Events

All events with subword '**onbefore**' can be cancelled.

To cancel some event you should return **false** within the appropriate event handler.

```
Cancelling the event handler
```

```
var myEvent = $$("dataTableId").attachEvent("onBeforeTabClick", function
() {
  ... // some event handler code
  return false;
})
```

Accessible objects and data

Inside the event handler you can refer to the holder component through keyword **this**. Besides, most event handlers get incoming argument(s), like the **id** of DataTable's row (see ui.datatable to know exactly what arguments are passed inside event handler).

Btw, using the **id** of a sub-element you can access a data item associated with it and all its properties, even if they were not used to draw the element. For example:

```
Referring within the event handler

$$("myTable").attachEvent("onafterselect",function (id) {
  $$("top_label").setValue(this.getItem(id).name)
})
```

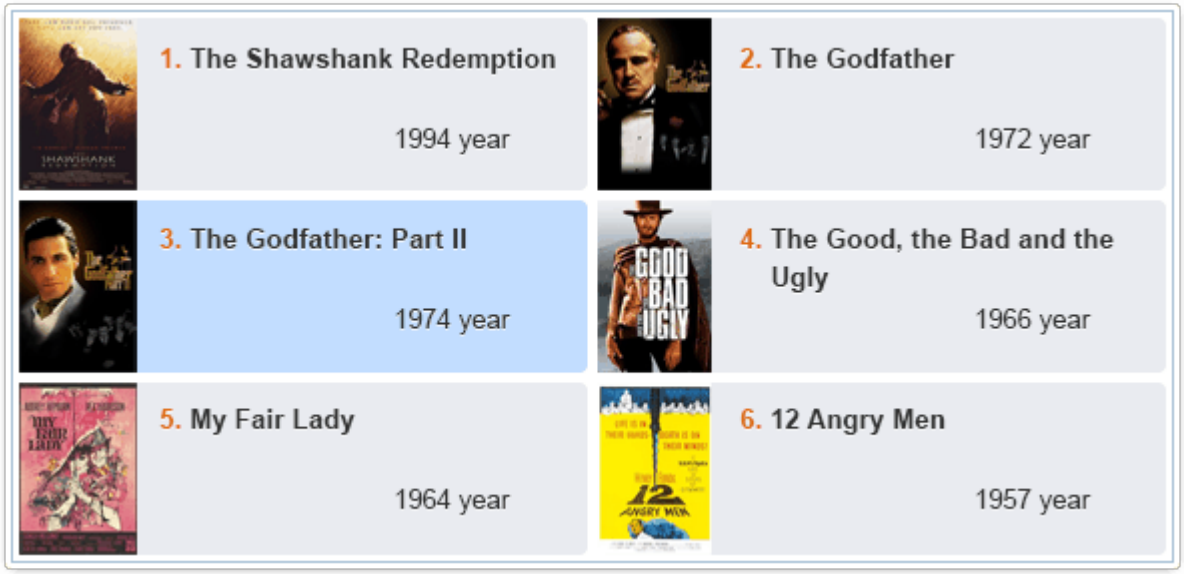
Dataview

API Reference

- Methods, Properties and Events
- Samples

Overview

Dataview displays objects with data according to a predefined template. It inherits from view.



Initialization

Specify the component's ID if you are going to work with the dataview later. Optionally, you can place dataview into an HTML container.

```
webix.ui({
    view:"dataview",
    container:"dataA",
    template:"<div class='webix_strong'>#title#</div> Year: #year#, rank:
#rank#",
    data:...,
    datatype:"...",
    xCount:3, //the number of items in a row
    yCount:4 //the number of items in a column
});
```

RELATED SAMPLE: Basic Initialization

Data property defines datasource for the view. Data can be loaded in one of supported data types.

Working with Dataview

- Data Loading
- Defining Data Templates
- Selection
- Adding/Deleting Items
- Editing Data
- Paging
- Adding Active Elements to Dataview Items

Note that there's a built-in possibility to edit data with dataview. You should create a prototype *dataview-edit* object beforehand.

Related Articles

- Data Object Operations
- Setting Dimensions for the Components
- Dynamic Resizing and Adjustment
- Redefinition of the Components
- Dataview CSS Image Map

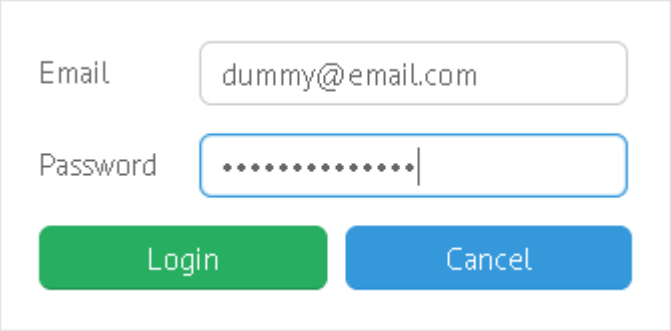
Form

API Reference

- Methods, properties and events
- Samples

Overview

Any form is a means of getting an information from users. It contains input fields and select components like checkboxes, radiobuttons alongside with submit/cancel buttons depending on the situation. Find them all in our Controls Guide.



Initialization

Ui-related form inherits from view. It resembles layout very much as it is divided into columns and rows where controls are put.

- **elements** - the form's specific property, an array of vertically arranged controls and control groups;
- **columns** - an array of horizontally arranged controls and control groups;
- **rows** - an array of vertically arranged controls and control groups.

```
Login form

webix.ui({
    view:"form",
    id:"log_form",
    width:300,
    elements:[
        { view:"text", label:"Email"},
        { view:"text", type:"password", label:"Password"},
```

```
        { margin:5, cols:[
            { view:"button", value:"Login" , type:"form" },
            { view:"button", value:"Cancel" }
        ]}
        ]
    });
```

RELATED SAMPLE: Basic Initialization

There exists a possibility to specify common configuration for all controls included in this or that form.

Grouping Controls in Form

Dividing form into sections

Form **elements** can be divided into columns and rows with any level of complexity.

```
{ view:"form", elements:[
    {cols:[
        { rows:[
            {view:"text"},
            {view:"datepicker"}
        ]},
        {view:"checkbox"},
        {view:"button"}
    ]},

]}
```

To add user-friendliness to form interface, separated blockes can be set within it by:

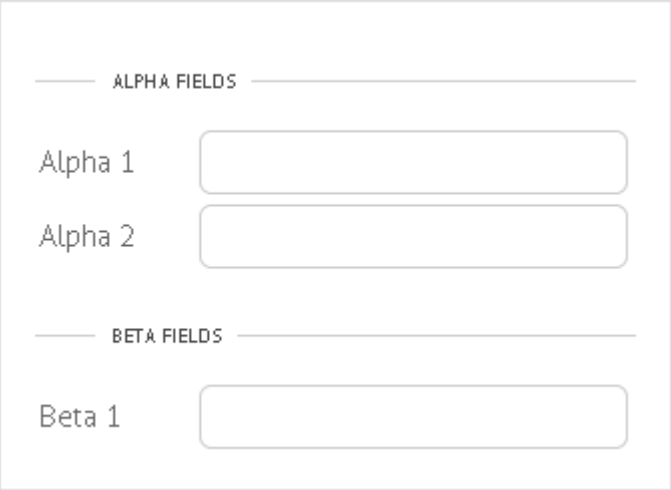
- Adding **headers** with **section** type before control blocks;
- Dividing a form into **fieldsets**.

Sections

Form elements are placed into different **row arrays** where the first row is a **section** header. Its template define text value of the header:

```
{view:"form", elements: [
    { rows:[
        { template:"Alpha fields", type:"section"},
        { view:"text", label:"Alpha 1", value:'' },
        { view:"text", label:"Alpha 2", value:'' }]
```

```
    },
    { rows:[
        { template:"Beta fields", type:"section"},
        { view:"text", label:"Beta 1", value:'' }]
    }
]}
```

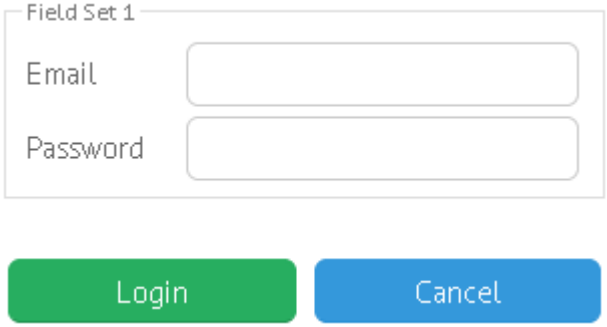


RELATED SAMPLE: Splitting Form into Sections

Fieldsets

Fieldset is a collection of controls surrounded by a border from all the sides. Webix fieldset features text **label** and **body** where **rows** or **cols** array of controls are stored:

```
{view:"form", elements:[
    { view:"fieldset", label:"Field Set 1", body:{
        rows:[
           { view:"text", label:"Login"},
           { view:"text", label:"Email"}]
        }
    }
]}
```



RELATED SAMPLE: Fieldset

Multiview Form

Form field can be placed into different tabs thus forming a multiview layout. Switching between tabs there is implemented with tabbar and segmented buttons as well as with the help of the dedicated tabview.

The switching controls are placed directly into the array of form **elements**.

```
{view:"form", elements:[
    {view:"tabview",
        tabs:["A","B","C"],
        cells:[
            { view:"text", name:"value1", label:"value1" },
            { view:"text", name:"value2", label:"value2" }
        ]
    }]
}
```

RELATED SAMPLE: Form with Tabs

Retrieving Form Values

Getting value of a single element

Since form elements comprise an array you can refer to each of them by its **number** starting from 0 and get/set the value of any item.

```
var form1 = [
    { view:"text", id:"log", label:'Login', name:"login"},
    { view:"text", id:"em" label:'Email', name:"email" },
    { view:"button", id:"sub", name:"submit", value:"Sumbit"}
];

webix.ui({
    view:"form",
    id:"my_form",
    elements: form1
});

var login = form1[].login;// returns current input value;
var button = form1[2].submit; //returns "Submit";
```

Or, you can apply **getValue()** method directly to the needed control:

```
$$("log").getValue();
```

Getting values of all form elements

To get an associative array of all elements (*name:value* pairs) you can use getValues method. To get to the necessary value, you should specify the **name** of the needed control.

```
$$("my_form").getValues().login; //returns current value of the text input field
```

Additionally, you can get only **changed** and **unchanged** form values with the following methods respectively:

- getDirtyValues - returns a hash of changed values;
- getCleanValues - return hash of unchanged values.

Disabling Form Elements

Any form element can be **disabled** or switched to **readonly** mode.

```
var form2 = [
        { view:"header" }
        { view:"text", value:'..', label:".."},
        { view:"text", type:'..', value:'..', label:".."},
]

form2[i].readonly = true; //where "i" takes the number value of the element staring from 0.
form2[i].disabled = true;
```

RELATED SAMPLE: Attributes

See also Form and HTMLform Treatment article to learn more about the most important form methods.

Reacting on Changes in Input Fields

If you want to track the moment when the value in an input changes from one to another and attach some event to it, make use of the dedicated **"onChange"** event.

The event is attached to element object that can be retrieved by its **ID** or **name**.

The attached function can be any you wish. Here is an example of logging the changes. The old value comes as **oldv** parameter while the new one is referred as **newv**

```
$$("form1").elements["login"].attachEvent("onChange", function(newv, oldv){
            webix.message("Value changed from: "+oldv+" to: "+newv);
```

RELATED SAMPLE: 'onChange' Event

Getting Parent Form for the Input

The easiest way to get to a parent form from any of its elements is to call the getFormView:

```
{view:"text", on:{"onChange":function(){
    var form = this.getFormView();
}}}
```

Loading and Parsing Initial Data

Webix form can load or parse data like any data management component. All the data loading rules are true for it.

Data **attributes** should coincide with **names** of form fields:

```
Parsing

var data = {id:1, fname:"Ann", lname:"Brown"};

webix.ui({
    view:"form", id:"myform", elements:[
        {view:"text", name:"fname"},
        {view:"text", name:"lname"}
    ],
    data:data
});
//or
$$("myform").parse(data);
```

```
Loading

$$("myform").load("data.php");
```

Sending Form Data

Form data can be sent to server in either of the following ways:

1 . Using Webix Ajax helper:

```
webix.ajax().post("some.php", form.getValues());
//with callback
webix.ajax().post("some.php", form.getValues(), function(text, data, xhr){
});
```

2 . Using webix.send method that emulates HTML form submitting:

```
webix.send("come.php", form.getValues());
```

3 . Indirectly, via the **bound master component** or DataCollection:

The method is good when the form is used for editing the data of the main component (datatable, tree, list, etc.). In this case not the form data matters but the data of the main component. Form saves the data to the master component while the master handles the serverside part.

```
//data from selected list item is pushed to the form
form.bind(list);

//pushes the data back to list item
form.save()
```

Data binding concept is described in a separate documentation article.

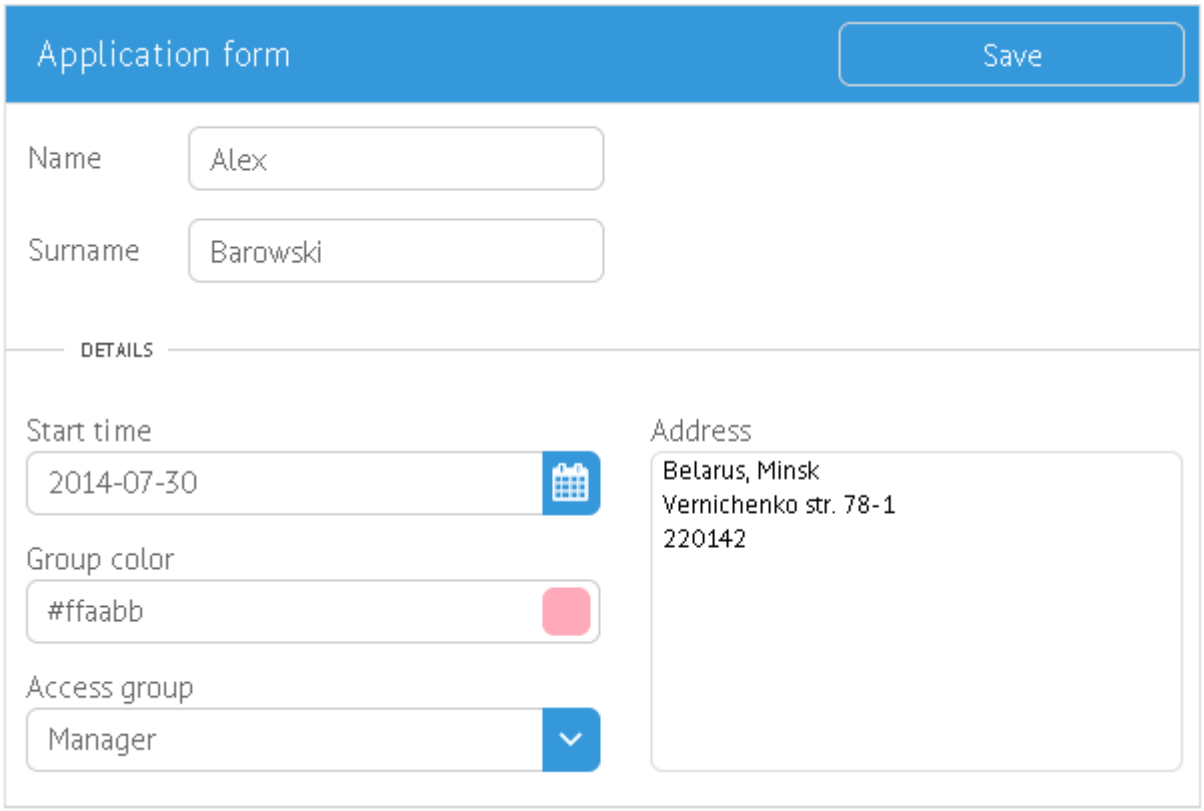
Related Articles

- Form and HTMLform Treatment
- Uploading Files with Form
- HTMLform
- Controls List
- Controls Common Functionality Guide
- Data Validation in Forms
- Event Handling
- Data Binding
- Creating Message Boxes

Form Controls

Webix delivers a variety of controls that can be placed on the form. Depending on the functionality they provide, these controls can be used for entering text, selecting an option, date or color, switching between multiview cells, etc.

You can learn more about common controls functionality from the article Controls Guide. The current article describes each form control in details.



Button

Standard Buttons




```
{ view:"button", id:"my_button", value:"Button", type:"...",
inputWidth:100 }
//types are descriped below
```

RELATED SAMPLE: Button ('button')

Icon Buttons




```
{ view:"button", type:"icon", icon:"mail", label:"Mail", width:80 }
```

RELATED SAMPLE: Icons

Image Buttons




```
{ view:"button", type: "image", image:".../img.gif", label: "Image Button"
}
```

RELATED SAMPLE: Buttons: Type 'image'

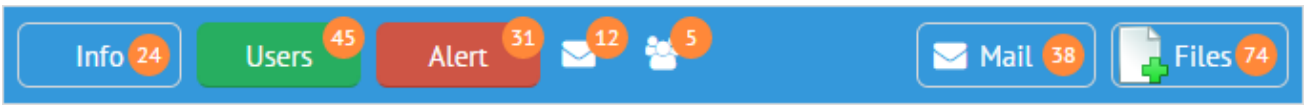
HTML Buttons




```
{ view:"button", id:"my_button", value:"Button", type:"htmlbutton" }
```

RELATED SAMPLE: Buttons: Styling

Buttons with Badges




```
{ view:"button", width:40, badge:12, type:"icon", icon:"envelope" }
```

RELATED SAMPLE: Buttons: Type 'icon' with Badges

Main Button Types

Type of a button defines the way is looks like:

- base (**no type**) - default button;
- **'form'** - a button for a form (darker than standard);
- **'danger'** - a red button to attract user attention;
- **'prev'** - a button with left arrow;
- **'next'** - a button with right arrow;
- **'htmlbutton'** - a button than can be defined by HTML markup;
- 'image' and its variations - a button with an image on it;

- 'icon' and its variations - a button with an icon in it.

Main properties

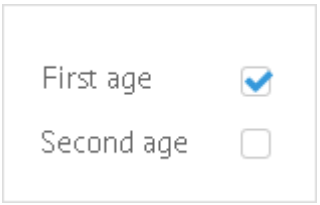
- **value** (string) - defines the text displayed on standard buttons (basic, *form*, *danger*, *prev*, *next*);
- **label** (string) - defines the text or/and HTML markup for the button of *imabeButton* and *iconButton* types and their valiarions as well as for *htmlbutton*;
- **width** (number) - sets the width of the button;
- **inputWidth** (number) - sets the width of the text on the button and adjusts button's width to it;
- **image** (string) - sets a path to the button icon;
- **align** (string)- positions a button with relation to a parent view;
- **popup** (string)- defines the ID of a popup window that will be shown on button click;
- **click** (function) - defines a function that will be executed on clicking the button (another possibility is to attach an event to it using the button's ID);
- **css** (string) - name of CSS class applied to a button;
- **badge** (number/string) - adds an orange circle that notifies about the number of new message for an icon button.

Specific button types as imagebutton and an icon button feature images / icons in their bodies and are described separately.

Controls Common Functionality

API Reference

Checkbox



RELATED SAMPLE: Check Box ('checkbox')

Checkbox enables zero, one and more choices from the defined list.

```
Checkbox Initialization

{ view:"checkbox", id:"field_a", label:"Second age", value:1}
```

Main properties

- **label** (string) - text label of a control. It can be customized by:

        - **labelAlign** (string) - label alignment towards its container. Possible values are "left" and "right". In any way, it's placed left to the control;
        - **labelWidth** - width of the label container;
    - **labelRight** (string) - set the text of right-hand label. Only checkbox features this functionality;
    - **value** (boolean) - defines whether the checkbox is selected by default or not (0 for false, 1 for true);
    - **align** (string)- positions button with relation to a parent view;
    - **checkValue** (string) - value of the checkbox in the checked state (1 by default);
    - **uncheckValue** (string) - value of the checkbox in the unchecked state (0 by default);

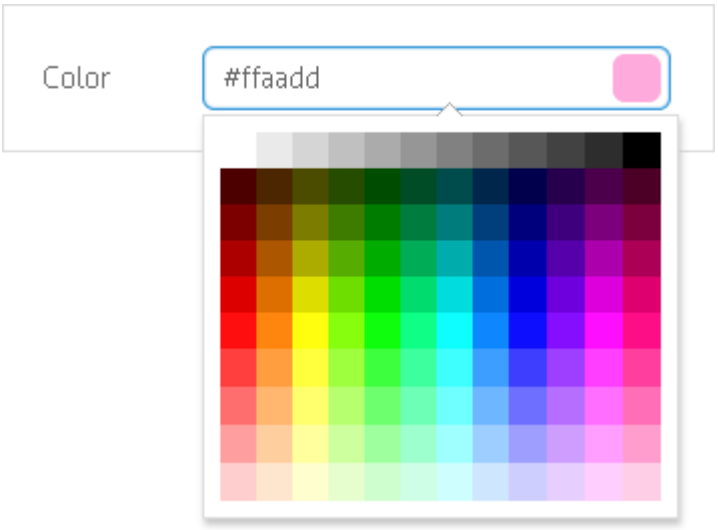
Webix checkbox control is based on standard HTML checkbox yet some skins use a FontAwesome icon to render it. To switch to a standard HTML checkbox, use **customCheckbox** property:

```
{ view:"checkbox", customeCheckbox:false}
```

Controls Common Functionality

API Reference

Colorpicker




```
Colorpicker Initialization

{ view:"colorpicker", label:"Front color", name:"color", value:"#ffaadd" }
```

RELATED SAMPLE: Colorpicker in Form

Colorpicker is used with ui-related colorboard that is initialized on clicking the control. Colorpicker allows selecting a color from the board for its further usage.

Main Properties

- **label** (string) - text label of a control. It can be customized by:
    - **labelAlign** (string) - label alignment towards its container. Possible values are "left" and "right".

In any way, it's placed left to the control;

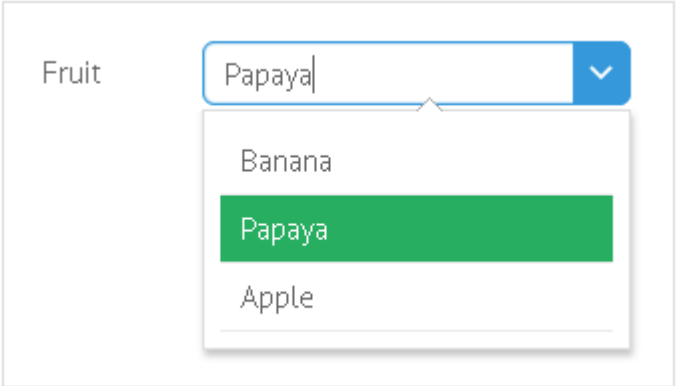
- **labelWidth** (number) - width of the label container;

- **value**(hex code) - the initial value of the picker that is changed by picking any color from the colorboard.

Controls Common Functionality

API Reference

Combo



RELATED SAMPLE: Editable Combo Box ('combo')

Ui-related combo is a text-like field with a popup window that appears as you click the field. A popup displays options for users to select. At the same time, you can edit the text field and the items will be filtered according to the input values.

```
Combo Initialization

//short form
{ view:"combo", id:'field_m', label: 'Combo', value:"One", yCount:"3",
    options:["One", "Two", "Three"]}

//full form
{ view:"combo", id:'field_t', label: 'Combo', value:"1", yCount:"3",
    options:[
    {id:1, value:"One"},
    {id:2, value:"Two"},
    {id:3, value:"Three"}
]}

//server options
{ view:"combo", options:"server/data.json"}
```

Main properties

- **value** (string, number)

        - within **options** array it sets text value for selectable items;
        - within Combo constructor it defines the initially selected item of the control (**option ID** in case of a full form, **option text** in case of a short form);
    - **placeholder** (string) - defines placeholder for combo input. The control should be inited without an initial value;
    - **yCount** (number) - the number of lines in a popup;
    - **options** (array, object, string) - sets the options to choose from;
    - **label** (string) - text label of a control. It can be customized by:
        - **labelAlign** (string) - label alignment towards its container. Possible values are "left" and "right". In any way, it's placed left to the control;
        - **labelWidth** (number) - width of the label container.

Advanced Combo Configuration

Combo box control can be customized in a number of ways, e.g.:

- options can be defined dynamically;
- options can be loaded from serverside;
- template can be changed;
- filtering rule can be changed..

To learn more, go to Advanced Combo and Richselect Settings article.

Pro Extensions for Combo

In the **Webix Pro** edition the control can be extended to show either a **dataview** or **datatable** in the popup:

- Dataview Suggest
- Grid Suggest

Getting Current Value

In case of a full initialization form, the **getValue()** method for combo will return ID of selected option rather than text. Use **getInput()** to get to input object and, hence, the text value.

```
$$('field_t').getValue(); // ->returns 1

$$('field_t').getInput().value // -> returns "One"
```

RELATED SAMPLE: Working with UnitList Items

Alternatives to Combo Box

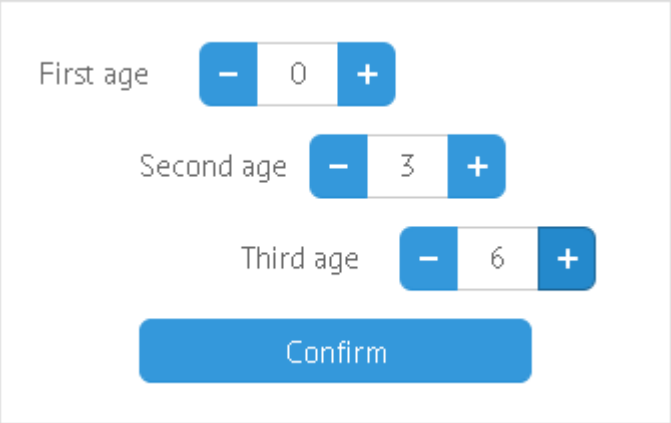
- richselect - identical to combo in look, but text field is non-editable.
- text coupled with suggest list. Indentical to combo in look, but while combo requires selection from a popup list, suggest offers values that can be either chosen or ignored.

Controls Common Functionality

API Reference

Counter

Counter is a spin-box helps select a number by clicking "+" and "-" buttons to reach the necessary value. Negative valuies are possible



RELATED SAMPLE: Spin Box ('counter')

```
Counter Initialization

{ view:"counter", label:"Age", step: 10, min:21, max:100}
```

Main properties

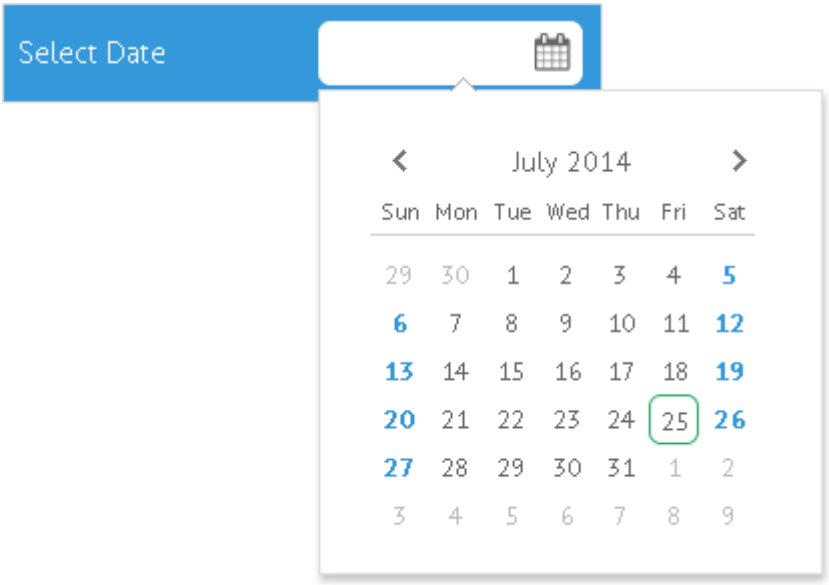
- **label** (string) - text label of a control. It can be customized by:
  - **labelAlign** (string) - label alignment towards its container. Possible values are "left" and "right". In any way, it's placed left to the control;
  - **labelWidth** (number) - width of the label container;
- **step** (number) - sets a step to the next value (1 by default);
- **align** (string)- positions a button with relation to the parent view;
- **value** (number) - the initial value of a control. 0 by default;
- **min** (number) - minimal value for a control. 0 by default;
- **max** (number) - maximum value for a control.

Note that if you set **min** and **max** parameters, the initial control value should be within the range.

Controls Common Functionality

API Reference

Datepicker



RELATED SAMPLE: Date Picker ('datepicker')

Datepicker is used in a combination with ui-related calendar that is initialized on clicking the control. It allows choosing a date and, optionally, time in the calendar for their further usage.

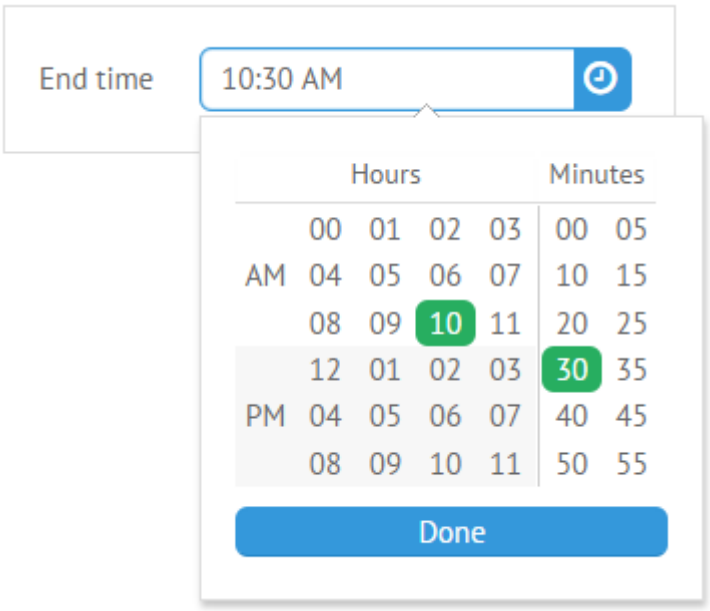
```
Datepicker initialization

{view:"datepicker", date: new Date(2012, 6, 8), label: 'Select Date',
timepicker:true}
```

Main properties

- **label** (string) - text label of a control. It can be customized by:
  - **labelAlign** (string) - label alignment towards its container. Possible values are "left" and "right". In any way, it's placed left to the control;
  - **labelWidth** (number) - width of the label container;
- **value** (string, obj) - initial value to display in the control input field;
- **placeholder** (string) - defines placeholder for datepicker input. The control should be inited without an initial value;
- **timepicker** (boolean) - enables a clock for time selection (*false* by default);
- **date** (function) - new Date() constructor to define the initial date of a calendar. If used without parameters, shows current date;
- **stringResult** (boolean) - sets the data type of value returned by **getValue()/get Values()** methods. If true, it returns date as string. By default the property is *false* and the method returns date object;
- **format** (string) - sets a date format to display in the datepicker field. By default it is "%Y-%m-%d".
- **type** (string) - (optional) sets the datepicker type:
  - "time" - configures datepicker for time selection only (uses webix.i18n.timeFormat as formatting pattern)

Datepickier type "time"



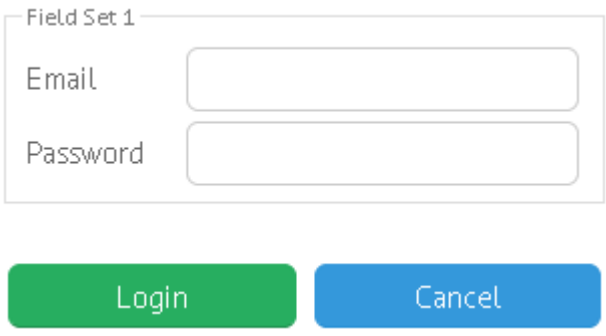

```
{ view:"datepicker", type:"time", stringResult:true}
```

Dates are formatted according to the Date formatting methods.

Controls Common Functionality

API Reference

Fieldset



A grouping control based on HTML < *fieldset* > tag. Creates a group to put controls into and sets a border around the group.

Can be used to neatly put controls into **layout**:

```
{view:"fieldset", body:{
    rows:[
        {view:"text", ..},
```

```
        {view:"button", ..}]
 }
}
```

RELATED SAMPLE: Multiple Fieldsets

Or, to create a control group within a **form**:

```
{view:"form", elements:[
    {view:"text", ..},
    {view:"fieldset", body:{
        rows:[
            {view:"text", ..},
            {view:"datepicker", ..}]}
    }
]}
```

RELATED SAMPLE: Fieldset

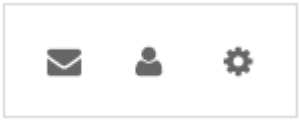
Main Properties

- **body** (object) - fieldset body object that contains **rows** or **cols** array with controls;
- **label** (string) - text label of the control group, integrated into border.

Controls Common Functionality

API Reference

Icon



```
Icon Initialization

{ view:"icon", icon: 'mail', align:"left"}
```

Main properties

- **align** (string) - the alignment of component inside designated area;
- **icon** (string) - a predefined name of an icon, its name. Icons and their names are listed on the Font Awesome.

Icons in this lib exist in the form of a separate **view** (as described above), as a **button type** and as **CSS class**. Read more about Icon types in the dedicated chapter of the documentation.

Controls Common Functionality

API Reference

Image button




```
Image Button Initialization

{ view:"button", type: "image", image:".../img.gif", label: "Image
Button"}
```

RELATED SAMPLE: Buttons: Type 'image'

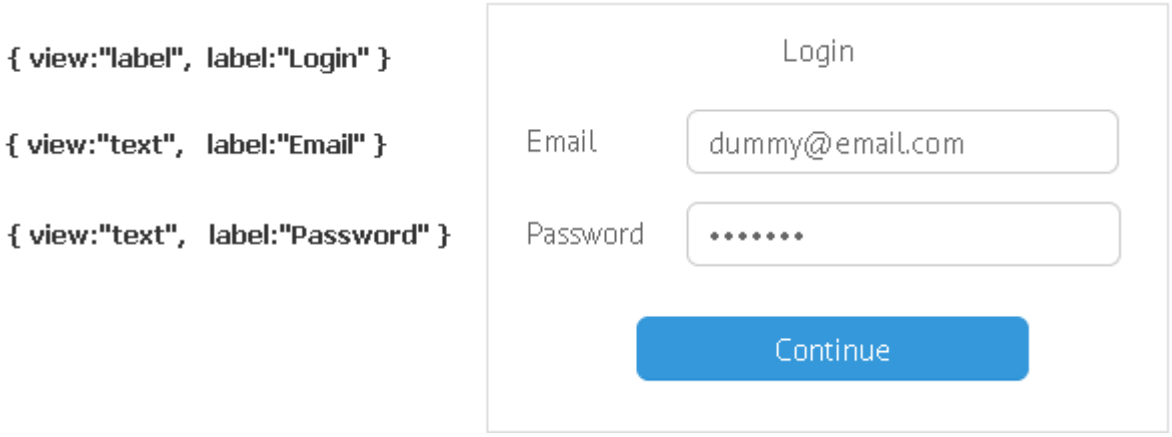
Main Properties

- **image** (path) - path to the desired image;
- **label** (string) - text on a button by the image;
- **type** (string) - defines the type of a button. Numbers indicate image sizes:
  - *image* - an image with a left-hand label (if any);
  - *imageTop* - an image with a bottom label;
  - *imageButton* - a button with an image and a left-hand label:
  - *imageButtonTop* - a button with an image and a label below the image;
- **width** (number) - sets the width of the entire button including its image and text label.

Controls Common Functionality

Manipulating Groups of Buttons

Label



Ui-related label is a non-scrollable placeholder for plain text. Like other components, it features specific **label** property that sets text value.

```
Label Initialization
```

```
{ view:"label", label: "Label", inputWidth:100, align:"center"}
```

RELATED SAMPLE: Attributes

Main properties

- **label** (string) - defines text of the label control;
- **inputWidth** (number) - defines the width of the control;
- **align** (string) - positions the label with relation to the parent view.

Label control can be used to render any user HTML or icons (as well as built-in in Webix FontAwesome icons):

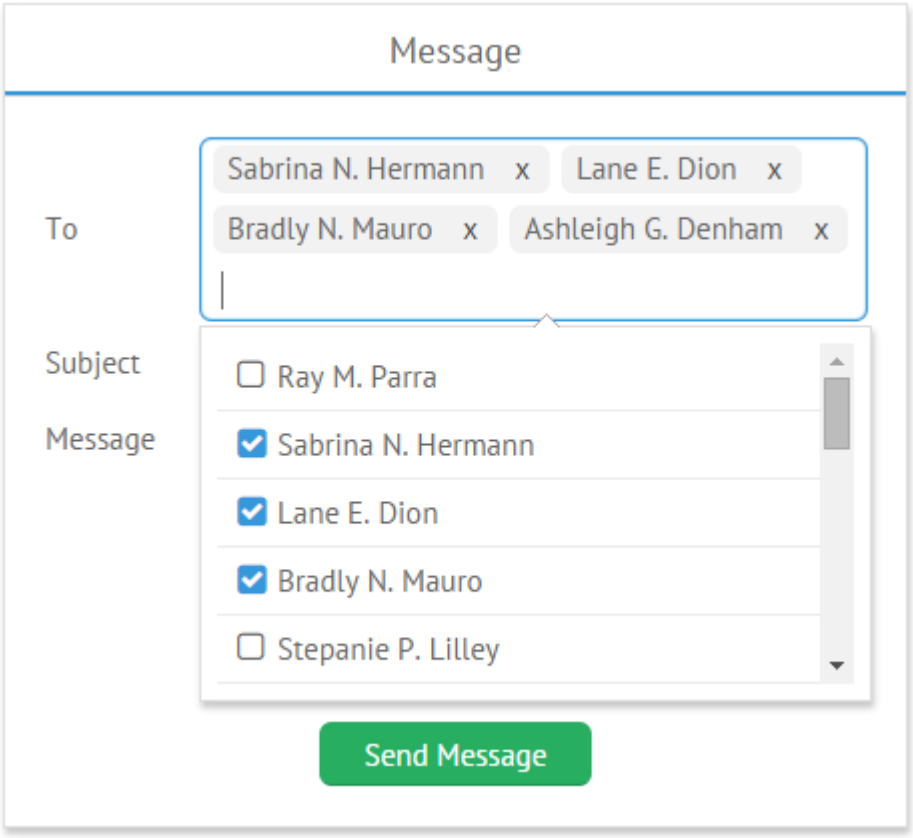
```
{ view:"label", label:"<span class='myCss'>My Label</span>"},
{ view:"label", label:"<span class='webix_icon fa-user'></span> My Label"
}
```

Controls Common Functionality

API Reference

Multicombo

Multicombo allows choosing several options from a suggest list.



RELATED SAMPLE: MultiSelect combobox

When a multicombo element in the form gets the focus, a suggest list opens. Each option has a checkbox next to it. To select an option, click on the corresponding row. Selected options will be added into one row.

To delete an option from multicombo, you can either click the "cross" sign or use the Backspace key.

```
Multicombo Initialization

{ view:"multicombo", label:"To", value:"2,3,4,13", suggest:
"data/names.js"}
```

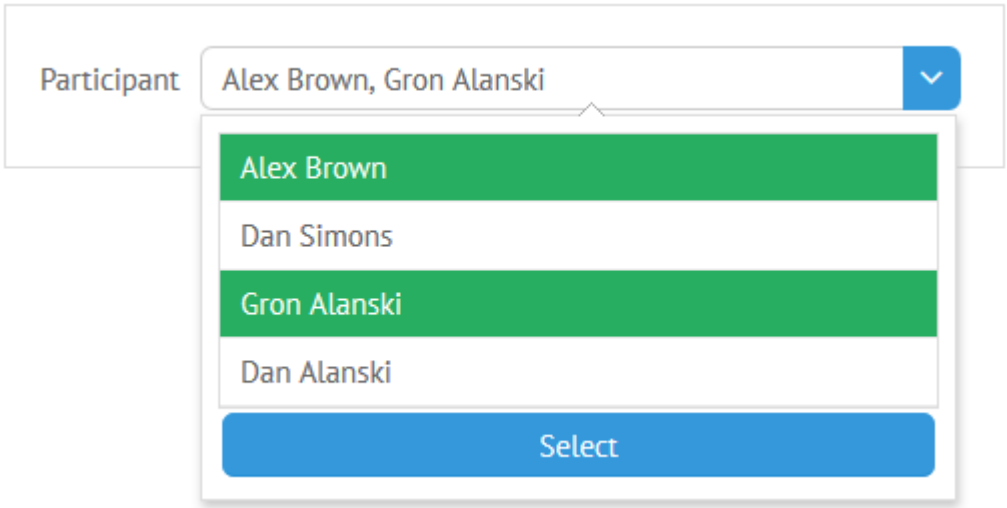
Main properties

- **label** (string) - specifies the text label of the control
- **labelPosition** (string) - the position of the label in relation to the control container
- **value** (array/string) - defines the initially selected items of the control
- **separator** (string) - a delimiter for separating options in a multicombo, comma by default
- **button** (boolean) - sets a button at the bottom of suggest list
- **suggest** (function/string) - specifies the data source of multicombo options

Controls Common Functionality

API Reference

Multiselect



RELATED SAMPLE: Multi-select input

UI-related multiselect is a select control based on Webix richselect and allows selecting multiple option at a time while displaying them in the dedicated non-editable input field.

The control's popup is supplied with a "select" button out of box.

```
Multiselect initialization

{ view:"multiselect", id:"multi", label:"Participant", options:[
```

```
    { id:1, value:"Alex Brown" },
    { id:2, value:"Dan Simons" },
    { ... }
], value:"1,3" }
```

Main properties

- **value** (string) - the initial value to display in the input field, its ID. The string can be separated by a comma (",") to define multiple values;
- **label** (string) - text label of a control. It can be customized by:
  - **labelAlign** (string) - label alignment towards its container. Possible values are "left" and "right". In any way, it's placed left to the control;
  - **labelWidth** (number) - width of the label container;
  - **labelPosition** (string) - defines label position related to the input field. The only possible value is "top";
- **placeholder** (string) - the initial text in the text field, disappears as you start typing in it;
- **options** (array, object) - defines the set of items to select from. Details;
- **optionWidth** (number) - defines the width of a popup list. By default, it is adjusted to the control width.

Changing "Select" button text

Multiselect control can be customized via extended configuration of its popup suggest, which is described in the Advanced Settings of Popup Selectors article.

Multiselect suggest features a **buttonText** property that allows defining any custom text for the button:

```
{view:"multiselect", label:"Participant",options:{
     buttonText:"Done",
     data:[
        { id:1, value:"Alex Brown" },
        { id:2, value:"Dan Simons" }
     ]
}}
```

Another way is to **modify the locale**.

"Select" button text is taken from the current locale (if other is not stated by **buttonText**, as shown above). The locale can be altered:

```
webix.i18n.controls.select = "Done";
webix.i18n.setLocale(); //apply the locale
```

Then all the multiselect instances in the app will be supplied with the "Done" word.

Accessing popup elements

Multiselect API allows getting to its constituent parts:

```
//button object
multiselect.getPopup().getButton();

//list object
multiselect.getpopup().getList();
```

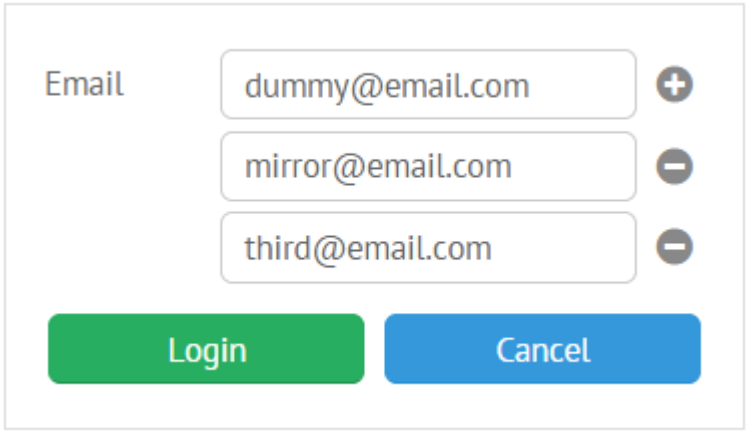
Controls Common Functionality

API Reference

Multitext



The control is available only in **Webix Pro** edition.



RELATED SAMPLE: Multi-Text input

UI-related multitext is a dynamic control based on standard Webix text input. It offers users the possibility to add and delete additional fields on the go.

```
Multitext initialization

{ view:"multitext", id:"multi", value:'dummy@email.com, mirror@email.com',
label:"Email" }
```

Main properties

- **value** (string) - the initial value to display in the input field. The string parts separated by a comma (",") will be put to additional input fields);
- **label** (string) - text label of a control. It can be customized by:
  - **labelAlign** (string) - label alignment towards its container. Possible values are "left" and "right". In any way, it's placed left to the control;

        - **labelWidth** (number) - width of the label container;
        - **labelPosition** (string) - defines label position related to the input field. The only possible value is "top";
    - **placeholder** (string) - the initial text in the text field, disappears as you start typing in it;
    - **icon** (string) - the control's icon.

Field Collection IDs

Each field of a multitext control features its own ID. It can be either a given (for the basic field) or an auto-generated one.

All the methods of a multitext object can be applied to any field object - the changes will be reflected for the whole control.

Adding and removing fields

Multitext API allows adding and removing additional fields dynamically:

```
var newFieldId = $$("multi").addSection();

//remove specific field
$$("multi").remove(newFieldId);
//remove all the additional fields
$$("multi").remove();
```

Notes:

- Only additional fields can be removed, basic one always remains.
- The methods can be called from any multitext object from the group.

Setting and getting values

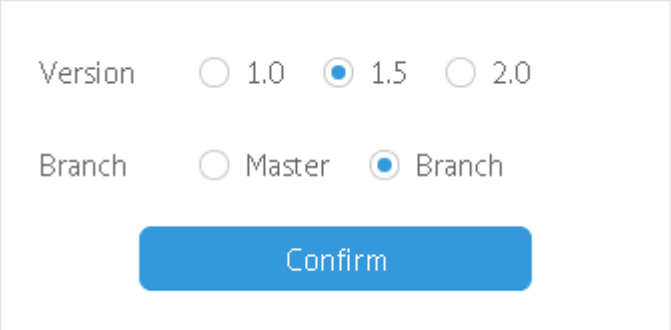
Multitext features an extended API for **setting** and **getting** its **values**.

- setValue and getValue methods for working with the value of the whole control;
- setValueHere and getValueHere methods for for working with the value of the first (basic) field.

Controls Common Functionality

API Reference

Radio



RELATED SAMPLE: Radio Button ('radio')

Ui-related radio is a block of severel items with the possibility to check only one of them at a time.

```
Radio Initialization

{view:"radio", label:"Branch", value:1, options:[
    { id:1, value:"Master" }, //the initially selected item
    { id:2, value:"Branch" }
]}
//short form for options array
 {view:"radio", options:["Master", "Branch"]}
```

Main properties

- **value** (string, number)
  - within **options** array it sets titles of radiobuttons;
  - within Radio constructor it defines the initially selected radiobutton;
- **id** (string, number) - used with the radio block as well as with each of its items;
- **options** (array, object) - defines the set of items to select from. Details.
- **label** (string) - text label of a control. It can be customized by:
  - **labelAlign** (string) - label alignment towards its container. Possible values are "left" and "right". In any way, it's placed left to the control;
  - **labelWidth** (number) - width of the label container;
- **align** (string)- positions button with relation to a parent view;
- **vertical** (boolean) - defines vertical arrangement of radiobuttons.

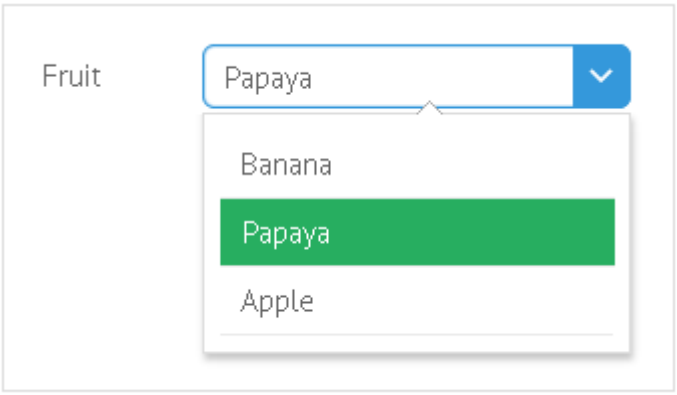
Webix radio control is based on standard HTML radio yet some skins use a FontAwesome icon to render it. To switch to a standard HTML radio, use **customRadio** property:

```
{view:"radio", customRadio:false}
```

Controls Common Functionality

API Reference

Richselect



RELATED SAMPLE: Non-editable Combo Box ('richselect')

Ui-related richselect is a non-editable combo-box.

```
Richselect Initialization
```

//full form

```
{view:"richselect",
    id: "list2", label: 'richselect',
    value:1, yCount:"3", options:[
        {id:1, value:"One"}, // the initially selected item
        {id:2, value:"Two"},
        {id:3, value:"Three"}
    ]
}
...
//short  form for options array
{view:"richselect", options:["One", "Two", "Three"]}

//server options
{ view:"richselect", options:"server/data.json"}
```

Visually, the **richselect** control consists of a text input and popup window that contains a standard list of options in its body.

Another possibility to create a component with a similar functionality is attaching a suggest list to a text field, but in this case, text field will be editable.

Main Properties

- **value** (string, number)
  - within **options** array it sets text value for selectable items;
  - within richselect constructor it defines the initially selected item of the control(**ID** in case of a long form, **text value** in case of a short form of initialization);
- **placeholder** (string) - defines placeholder for richselect input. The control should be inited without an

initial value;

- **label** (string) - text label of a control. It can be customized by:
  - **labelAlign** (string) - label alignment towards its container. Possible values are "left" and "right". In any way, it's placed left to the control;
  - **labelWidth** (number) - width of the label container;
- **options** (array, object, string) - sets the options to choose from in a long or short form. Details.

Advanced Richselect configuration

Richselect control can be customized in a number of ways, e.g.:

- options can be defined dynamically;
- options can be loaded from serverside;
- template can be changed..

To learn more, go to Advanced Settings of Popup Selectors article.

Pro Extensions for Richselect

In the **Webix Pro** edition the control can be extended to show either a **dataview** or **datatable** in the popup:

- Dataview Suggest
- Grid Suggest

Getting Current Value

In case of a full initialization form, the **getValue()** method for combo will return ID of selected option rather than text. Use **getInput()** to get to input object and, hence, the text value.

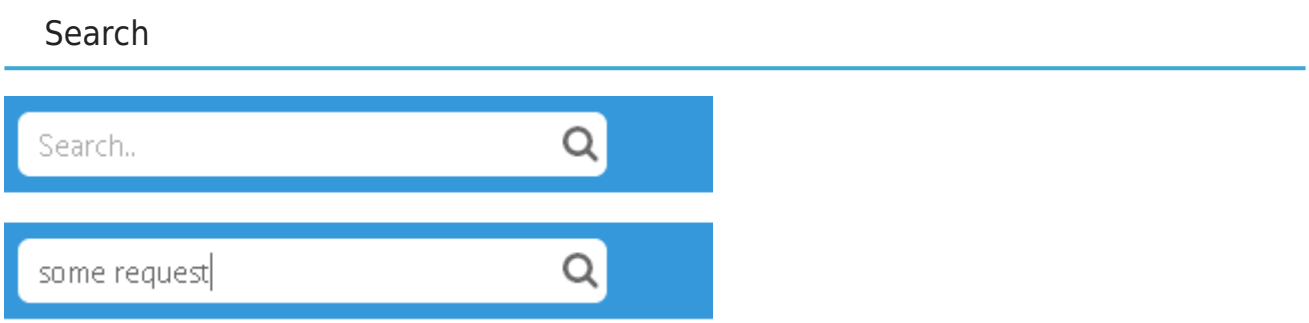
```
$$('field_t').getValue(); // ->returns 1

$$('field_t').getInput().value // -> returns "One"
```

RELATED SAMPLE: Working with UnitList Items

Controls Common Functionality

API Reference



RELATED SAMPLE: Search Field

Main Properties:

- **placeholder** (string) - a prompt text displayed in the search field.
- **label** (string) - text label of a control. It can be customized by:
  - **labelAlign** (string) - label alignment towards its container. Possible values are "left" and "right". In any way, it's placed left to the control;
  - **labelWidth** (number) - width of the label container.

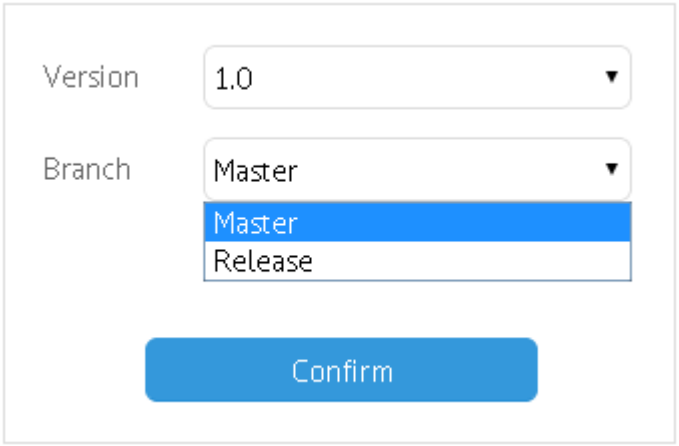
UI-related search looks like an input text field with a 'search icon' to indicate that it can be used for filtering data in a data-presenting component to get to the necessary item.

Filtering function should be written by a developer. See the attached sample for details.

Controls Common Functionality

API Reference

Select



RELATED SAMPLE: Select Box ('select')

Select is a control that allows selection from several items. Looks like a dropdown list.

```
Select Initialization

// full form
{view:"select", label:"Branch", value:1, options:[
    {id:1, value:"Master" }, // the initially selected value
    {id:2, value:"Release" }
  ], labelAlign:'right'
}

//short form
{ view:"select", options:["Master", "Release"]}
```

```
//server options
{ view:"select", options:"server/data.json"}
```

Main properties

- **label** (string) - text label of a control. It can be customized by:
  - **labelAlign** (string) - label alignment towards its container. Possible values are "left" and "right". In any way, it's placed left to the control;
  - **labelWidth** (number) - width of the label container;
- **options** (array, object, string) - defines a set of items to select from;
- **value** (string, number)
  - within **options** array it sets text values for select items;
  - within Select constructor it defines the initially selected item of the control (**option ID** in case of a full form, **option text** in case of a long form);

Select option, a list record, may contain:

- a short string, like "Apple";
- **multi-line text** with html tags. In this case, parent (e.g. toolbar) height should be increased.

```
"toolbar", height:100,
.....

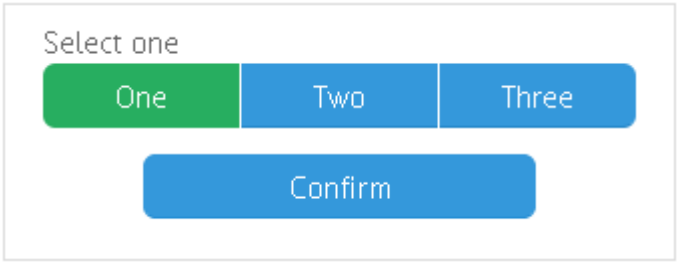
options:[
    {id:1,value:"Some long multiline content<ul><li>item1</li><li>item
2</li></ul>"},
    {id:2,value:"Papaya" }
]
```

RELATED SAMPLE: Select Boxes

Controls Common Functionality

API Reference

Segmented Button



RELATED SAMPLE: Holistic Panel with Multiple Buttons ('segmented')

Segmented is an integral button with as many segments as you like. Can be used for switching between multiview cells.

```
Segmented Initialization

{view:"segmented", multiview:true, value:1, options:[
    { id:"1", value:"Section A" }, // the initially selected segment
    { id:"2", value:"Section B" },
    { id:"3", value:"Section C" }]
}

//short form for options array
{ view:"segmented", options:["Section A", "Section B", "Section C"]}
```

Main properties

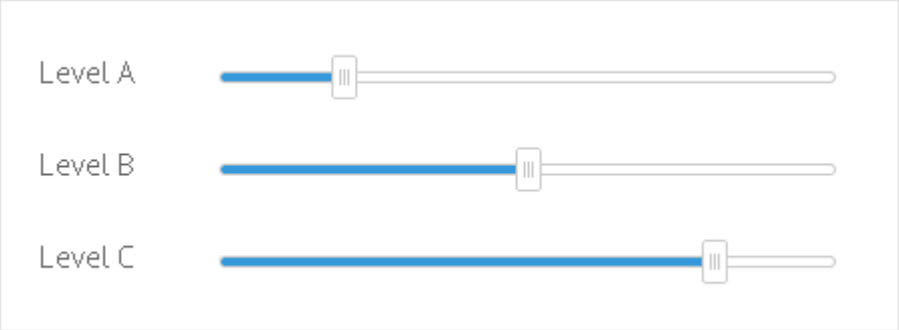
- **multiview** (boolean) - connects the control to the multiview cells, enables switching between these cells;
- **options** (array, object) - defines the buttons (segments) within the control. Details;
- **value** (string, number)
  - within **options** array it sets titles for button segments;
  - within Segmented constructor it defines the initially selected item for the control;
- **label** (string) - text label of a control. It can be customized by:
  - **labelAlign** (string) - label alignment towards its container. Possible values are "left" and "right". In any way, it's placed left to the control;
  - **labelWidth** (number) - width of the label container;
- **selected** (string) - defines the tab that will be visible initially.

Adding and Deleting Segments (Options) on the Go

Controls Common Functionality

API Reference

Slider



RELATED SAMPLE: Slider

Slider is designed for setting a numeric value from a predefined range of values. It's done by dragging a marker left and right over the value line.

Negative values are possible.

```
Slider Initialization

{ view:"slider", label:"Level", value:"20", min:10, max: 120; name:"s1"}
```

Main Properties

- **value** (number) - initial value of the control;
- **label** (string) - text label of a control. It can be customized by:
  - **labelAlign** (string) - label alignment towards its container. Possible values are "left" and "right". In any way, it's placed left to the control;
  - **labelWidth** (number) - width of the label container;
- **name** (string) - name of the control used to get its value in the form;
- **step** (number) - distance to the next selectable value (1 by default);
- **min** (number) - minimal value of a slider control. 0 by default.
- **max** (number) - maximum value of the slider control. 100 by default.
- **title** (template, function) - text label over the slider marker

RELATED SAMPLE: Slider: Custom Step.

Note that if you set **min** and **max** parameters, the initial control **value** should be within their range.

Slider Title is based on the current value of the control.

For simple title that displayes current value and **static text**, set a template via **webix.template** class:

```
{view:"slider", title:webix.template("Selected: #value#")}
```

For complex title that **changes its text** depending on the current value, define a function template:

```
{view:"slider", title:function(obj){
    var text = obj.value > 20 ? "Minimum level reached. " : ""; //title
for values over 20
    return text + "Value: "+ obj.value;} //title for other values
}
```

Slider Events

Make use of slider events to set changing title for a drag marker:

- **onChange** - a common event for all conrtol that fires each time control value changes. For slider it takes place as soon as you **stop dragging** the marker, which sets a new value.
- **onSlider Drag** - fires each time you drag the marker over the line while paying attention to the changing value.

```
{view:"slider", on:{
    onChange:function(){
        this.define("title", "Final value " + this.getValue());
        this.refresh();
    },
    onSliderDrag:function(){
        this.define("title", "Dragging... Currently "+this.getValue());
        this.refresh();
    }
}}
```

RELATED SAMPLE: Slider: Labels

The new title is set with the help of **define()** method while current value of derived with **getValue()**. Read more about Changing Component properies.

Controls Common Functionality

API Reference

Range Slider



The control is available only in **Webix Pro** edition.



RELATED SAMPLE: RangeSlider

Range Slider is an advanced slider control that allows selecting a specific range of values with two markers. The colored space between the markers defines the values included into the range.

```
Range Slider Initialization

{ view:"rangeslider", label:"Level B", value:[,50], name:"s2"}
```

Main Properties

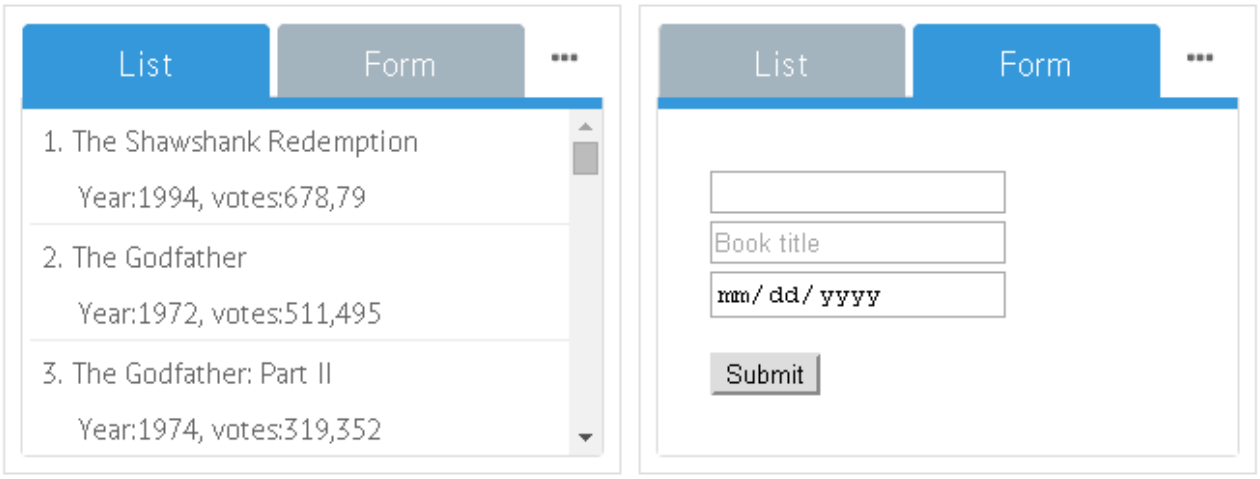
- **value** (string/array) - the starting and the ending values of the range;
- **label** (string) - text label of the control. It can be customized by:

        - **labelAlign** (string) - label alignment towards its container. Possible values are "left" and "right". In any way, it's placed left to the control;
        - **labelWidth** (number) - width of the label container;
    - **name** (string) - name of the control used to get its value in the form;
    - **title** (template, function) - text label over the slider marker

Controls Common Functionality

API Reference

Tabbar



RELATED SAMPLE: Tabbar

RELATED SAMPLE: Panel with Clickable Items ('tabbar')

U-related tabbar is an integral button that allows for changing between views in a multiview layout. Its tabs can be nested with images and icons.

```
Tabbar Initialization

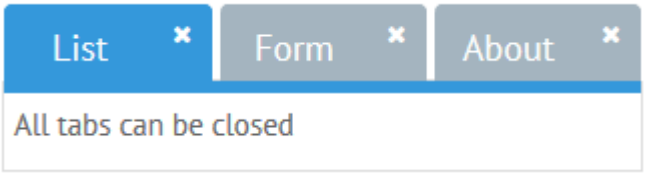
rows:[
    {view:"tabbar", id:'tabbar', value: 'listView', multiview:true,
options: [
        { value: 'List', id: 'listView'},
        { value: 'Form', id: 'formView'},
        { value: 'Empty', id: 'emptyView'}]
    },
    //and then goes multiview
    {cells:[...]}
]
```

Main properties

- **multiview** (boolean)- if *true*, it links tabs to multiview cells;
- **options** (array, object) - defines buttons for switching between views. Details;

- **value** (string) -
  - within **options** array it sets titles for tabs;
  - within Tabbar constructor it defines the tab that will be shown initially on page loading.
- **type** (string) - if set to "bottom", the tabbar gets other styling, more suitable for tabs placed under the multiview. It can be done by changing the order of layout rows;
- **close** (boolean) - if *true*, the "close" functionality for each tab is provided. Each tab can be closed by clicking a dedicated icon on it.

Defining tabs that can be closed



Tabs featuring a **close** property in tabbar **options** array gain the functionality:

```
//all tabs can be closed
{view:"tabbar", close:true, options:[...]}

//only "Tab1" can be closed
{view:"tabbar", options:[
    {id:1, value:"Tab1", close:true},
    {id:1, value:"Tab2"}
]}
```

RELATED SAMPLE: Tabview: Close tab button

Defining Icons for the tabs



Icons are defined through additional HTML provided within the tab **value**:

```
 { view:"tabbar", options: [
    { value: "<span class='webix_icon fa-film'></span>List", id:
'listView' }
 ]}
```

RELATED SAMPLE: Multiview Tabbar with Icons

More about different types of defining icons in Webix you can learn in the Icons with UI Components article.

Advanced Tabbar Usage

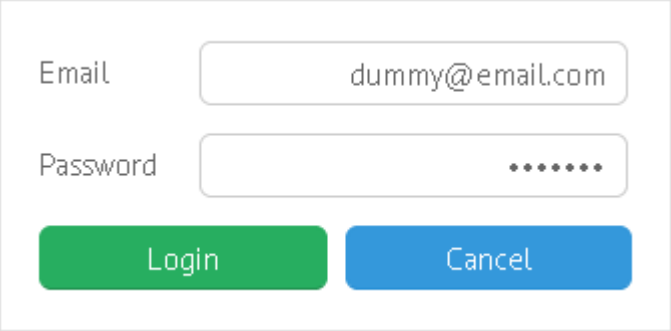
- Adding and Deleting Tabs (Options) on the Go

- Dynamic Tabbar

Controls Common Functionality

API Reference

Text



RELATED SAMPLE: Text Input ('text')

Ui-related text control is a one-line input field with a label.

```
Text Initialization

{view:"text", value:'dummy@email.com', label:"Email", labelAlign:'right'},
{view:"text", type:'password', value:'123pass', label:"Password",
labelAlign:'right'}
```

Main properties

- **value** (string, integer) - the initial value for a text field, can be editable or readonly on your choice;
- **label** (string) - text label of a control. It can be customized by:
  - **labelAlign** (string) - label alignment towards its container. Possible values are "left" and "right". In any way, it's placed left to the control;
  - **labelWidth** (number) - width of the label container;
  - **labelPosition** (string) - defines label position related to the input field. The only possible value is "top";
- **type** (string)- type of a text field, can be either "text" by default, or "password" or "email". Type is vital for HTML5 input data validation.
- **placeholder** (string) - the initial text in the text field, disappears as you start typing in it;
- **suggest** (array, string) - collection of suggest values, or the ID of an attached suggest list with these values.

At the same time, "text" properties can be set via an **attributes** object containing properties that coincide with HTML input tag properties. The object is used for HTML5 data validation.

- **attributes** (object)

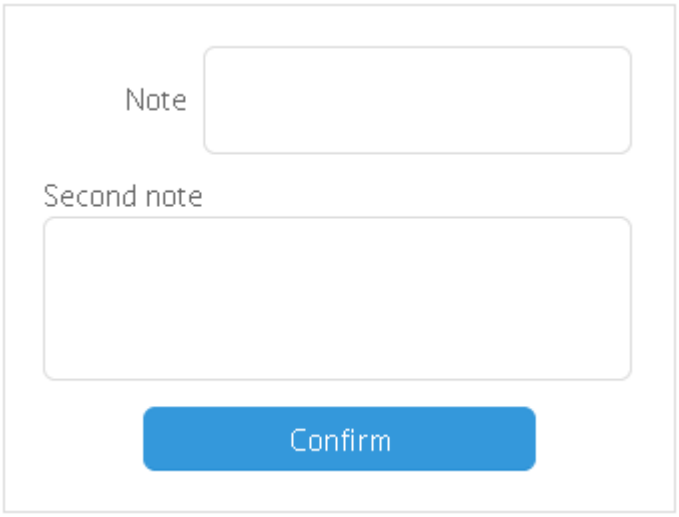
- **maxlength** (number) - the maximum quantity of symbols that can be typed into the text field;
- **required** (boolean) - indicates whether this field must be filled before the form can be submitted. False by default;
- **title** (string) - a prompt message that appears on pointing cursor over the field.

The same is true for textarea.

Controls Common Functionality

API Reference

Textarea



RELATED SAMPLE: Multiline Text Input('textarea')

Ui-related textarea control is a multi-line input field with a label.

```
Textarea Initialization

{ view:"textarea", label:"Note", labelAlign:"right", height:200, value:
"type here" }
```

Main properties

- **label** (string) - text label of a control. It can be customized by:
  - **labelAlign** (string) - label alignment towards its container. Possible values are "left" and "right". In any way, it's placed left to the control;
  - **labelWidth** (number) - width of the label container;
- **width, height** (integer) - defines the dimension of the text field;
- **labelAligh** (string) - sets the position of a text label;
- **value** (string) - sets the initial text in the field (replaced by input text later)
- **placeholder** (string) - the initial text in the textarea field, disappears as you start typing in it;

- **attributes** (object) - hash of properties. See text description for details.

Controls Common Functionality

API Reference

Toggle



RELATED SAMPLE: Toggle Button ('toggle')

A two-state button that can be not only clicked but pressed. Hence, it has two states - pressed and unpressed (initial), which makes it resemble checkbox that also has two states (checked and unchecked).

If needed, Toggle can change its values for each each state.

```
Toggle initialization

{ view:"toggle", type:"iconButton", name:"s4",
    offIcon:"pause",
    onIcon:"play",
    offLabel:"Disabled", onLabel:"Enabled" },
```

Main properties

- **name** (string) - read-only property, defining the toggle's name. It should be unique within a page;
- **label** (string) - text on the button body, one and the same for pressed and unpressed states;
- **onLabel** (string) - text label on the button in a pressed state;
- **offLabel** (string) - text label on the button in an unpressed state;
- **icon** (string) - icon for pressed and unpressed toggle states;
- **onIcon** (string) - icon for a pressed toggle;
- **offIcon** (string) - icon for an unpressed toggle.

Icons taken from the Font Awesome, so pick the necessary icon name at their site.

Toggle Types

Toggle type is defined by the **type** property. If it isn't defined, a standard button is displayed. For image toggles there exist the following types:

- **iconButton** for a standard button with an icon;
- **image** for an image with a left-hand label (if any);
- **imageButton** for a standard button with an image and a left-hand label;
- **imageTop** for a picture and a label below it;

- **imageButtonTop** for a standard button with a picture and a label under the picture;
- **next** for a button with a right-hand arrow;
- **prev**for a button with a left-handed arrow.

Image Toggle types

`type:"iconButton"`	
`type:"image"`	
`type:"imageButton"`	
`type:"imageTop"`	
`type:"imageButtonTop"`	

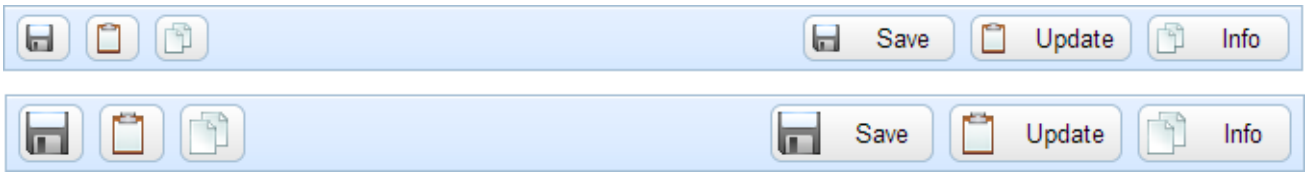
RELATED SAMPLE: Toggle Image Button ('toggle')

Controls Common Functionality

API Reference

Manipulating Groups of Buttons

In case of image buttons, you can use a function to define differently-sized buttons for a toolbar.



RELATED SAMPLE: Buttons: Type 'imageButton'

Image button is a derivation from a standard button. It is defined with an **image** property where you specify the path to the necessary image.

Button type depends on its size with typical sizes being 18, 24, 32 and 48. If you have the images stored in different folders with corresponding titles, you can make use of a custom function to set the needed size.

```
function buttons(size){
    return  [
    { view:"button", type:"image"+size,
        image:"../common/imgs/"+size+"/save.gif", width:size+14}, //for
image button
    { view:"button", type:"image"+size, label:"Save",
        image:"../common/imgs/"+size+"/save.gif", width:size+60}
        //for image button with a text label - it should be wider
            ];
        }
```

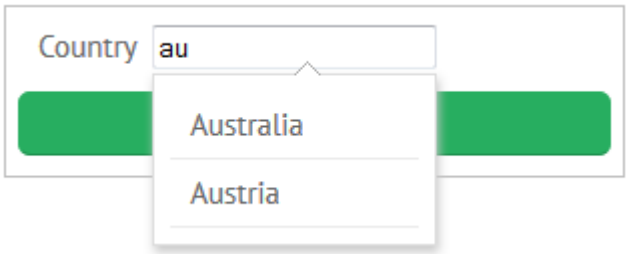
Then, when defining properties of a toolbar constructor, include the function into its **cols** property and pass the necessary parameter into it.

```
   webix.ui({
        container:"areaA",
        view:"toolbar", paddingY:2, height:28,
        cols: buttons(18)
    });
```

The toolbar above will initially work with icons measuring 18 pixels in width while the width of the button itself is calculated withtin the function.

Suggest List

Suggest is a **list of options** for input controls that aids form filling by suggesting the necessary value on the base of already typed text. It is based on Webix list component.



RELATED SAMPLE: Suggest List for Input: Populating with Server-side Data

Suggest list can be used with the following components (should be configured separately):

- standard HTML input;
- Webix JS text field;
- text editor within a component item.

Suggest list is already implemented in the following Webix components (but can be customized):

- combo and richselect form controls;
- combo and richselect component editors.

How does it work?

A popup window with a suggest list control appears each time you try to enter a value into the control or editor:

- when you **type** something into an HTML input or Webix text and combo control.
- when you **click** the richselect control;

Suggest list and Text input

When suggest list is combined with **text** control (editor) or HTML input its values are automatically filtered according to the letters you have already typed. You can select an item from the list to fill the input or ignore the suggest list and type you own text.

Suggest list and Combo

When suggest list is combined with **combo** control (editor) its values are automatically filtered according to the letters you have already typed. You can either select an option fromt he list or type the text to match any of the options. You cannot ignore the suggest list and enter you own text.

Suggest list and Richselect

When suggest list is combined with richselect control (editor) a popup appears the moment you click its non-

editable input. You must obligatory choose some value from the list.

Suggest list values can be stored

- on client side;
- on server side.

Default **timeout** between key pressing and filtering equals to 1ms, still it can be modified by the dedicated property.

```
{
    view:"suggest",
    data:[..] //or url:"some.php",
    keyPressTimeout:100ms
}
```

Initializing suggest list

Suggest list can be initialized as:

1 . **Separate view** attached to an input with the help of the **input** property that points to the input's **ID**:

```
webix.ui({
    view: "suggest",
    input: "country4", // will be linked to 'country4' input
    data: [
        {id:1, value: "Albania"},
        {id:2, value: "Bhutan"},
    ... //list of suggest values can be as long as you wish
    ]
});

<input type='text' id='country4' value='Sweden' />"}
```

2 . The **same-name property** of JS text input, combo and richselect:

```
{view:"text", name:"country", label:"Country", value:"Belarus", suggest:[
        {id:1, value: "Albania"},
        {id:2, value: "Bhutan"},
        ...]
}
```

3 . The **same-name property** of JS combo and richselect and take extensive configuration:

```
webix.ui({
    view:"richselect", suggest:{
        data:[
            {id:1, value:"One"},
            {id:2, value:"Two"} //options list
```

```
        ],
        ready: function(){
            $$("richselect_1").setValue(this.getFirstId()); //defines the
initially visible option
        }
    }
});
```

RELATED SAMPLE: Suggest for richselect and combo

Note that you must select any value from a suggest list, since it's required by combo and richselect nature.

Suggest list with client-side data

On the client side the list with suggestions can be stored either of supported formats. It may be put directly into **data** property (as shown above) or in a variable declared beforehand.

```
Suggest List with JSON data

var countries = [
    {id:1, value: "Albania"},
    {id:2, value: "Bhutan"}
];

{ view:"text", label:"Country", value:"Belarus", suggest:countries}
```

RELATED SAMPLE: Suggest List for Input: Populating with Client-side Data

Suggest list with server-side data

At the same time, data can come from server side. All you need is to specify the script file that will get the data from the database.

```
As value of dataFeed property of the Suggest component

webix.ui({
    view: "suggest",
    input: document.getElementById("country4"),
    dataFeed:"server/data.php"
});
```

```
As value of suggest property

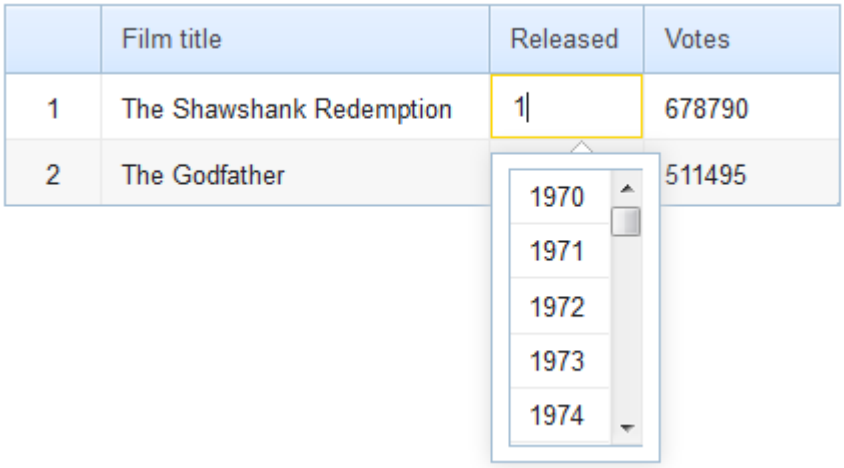
{ view:"text", name:"country", label:"Contry", value:"Albania",
suggest:"server/data.php"}
```

RELATED SAMPLE: Suggest List for Input: Populating with Server-side Data

Suggest list is connected with an input field by **suggest** property included into the **text** constructor.

Suggest list for component editors

Component editors duplicate editing controls yet they are inited within the component body.



Suggest can be inited for **combo** and **text** editors. To do this, make the following steps:

- create a suggest list as **separate view** or define its **configuration** in a JSON object and set options for it:
- create a component, specify **text** editor type for it and link to the suggest:

```
var year_suggest_a = webix.ui({
    view: "suggest",
    data:[...]
});

//or
var year_suggest_a = {
    view: "suggest",
    data:[...]
};

webix.ui({
    view:"datatable",
    columns:[
        {id:"year", editor:"text", suggest:year_suggest_a}
    ]
});
```

RELATED SAMPLE: Datatable: Autosuggest

At the same time, you can provide all the logic by API, which allows customizing suggest behaviour:

- create a suggest list as **separate view** and set options for it:
- link suggest list to the input field of the editor object with the **linkInput** method:
- provide handlers for edit events.

```
var year_suggest_b = webix.ui({
    view: "suggest",
    data:[...]
});

var gridb = webix.ui({
    view:"datatable",
    columns:[
        {id:"year", editor:"text"}
    ]
});


gridb.attachEvent("onAfterEditStart", function(object){
    if (object.column == "year") { //only for editors in this column
        var editor = this.getEditor(object);
        year_suggest_b.linkInput(editor.getInput());
    }
});

gridb.attachEvent("onAfterEditStop", function(object){
    year_suggest_b.hide();
});
```

RELATED SAMPLE: Datatable: Autosuggest

Suggest list can as well be attached to **combo** and **richselect** editors, yet then it looses its status of 'adviser' and selection from it becomes compulsory.

```
webix.ui({
    view:"datatable",
    columns:[
        {id:"year", editor:"combo", suggest:{
            data:[...], //or url:""
            on:{
                //events
            }
        }}
    ]
});
```

RELATED SAMPLE: Datatable: Advanced Combo Editor

Customizing suggest list

Suggest list allows defining lots of custom settings in case of a long initialization pattern.

It works for a suggest defined as a standalone view:

```
{view:"suggest",
    //popup settings
    ...,
    body:{
        //list settings
    }
}}
```

As well as for a suggest defined as a control/editor property:

```
{view:"combo", options:{
    //popup settings
    ...,
    body:{
        //list settings
    }
}}
```

- Popup settings include typical popup properties and suggest-specific ones, like fitMaster, or **url** and **data**;
- List settings inlcude typical list properties.

Note that an suggest list configuration can be set via either **suggest** or **options** property for richselect, combo and multiselect controls.

Where can the customization be useful?

- Advanced Configuration of Popup Selectors
- Customizing built-in data editors

Sizing and positioning

Popup height

Suggest list is tuned to show maximum 10 items at a time. If there are more items, a scrollbar appears. If there are fewer item in the list, its height shrinks.

The number of items in the suggest list can be controlled by an **yCount** property of **list object**:

```
{view:"combo", suggest: {
    body:{
        yCount:5,
        data:[...]
    }
}}
```

Popup width

By default, a popup is adjusted to the width of a master control. To change it, set fitMaster property to *false* and define any desirable width or use the default 300px:

```
{view:"combo", suggest: {
    fitMaster:false,
    width:400,
    data:[...]
}}
```

Popup position

Suggest API offers the following variants of popup positioning in relation to the text field it's inited for:

- **"bottom"** (default). Doesn't require direct initialization;
- **"top"**;
- **"left"**;
- **"right"**.

Since we speak here about **relative** position, the property to set it has the same name:

```
{ view:"text", suggest:{ data:countries, relative:"left" }}
```

RELATED SAMPLE: Attaching Suggest List to Input

Advanced suggest lists

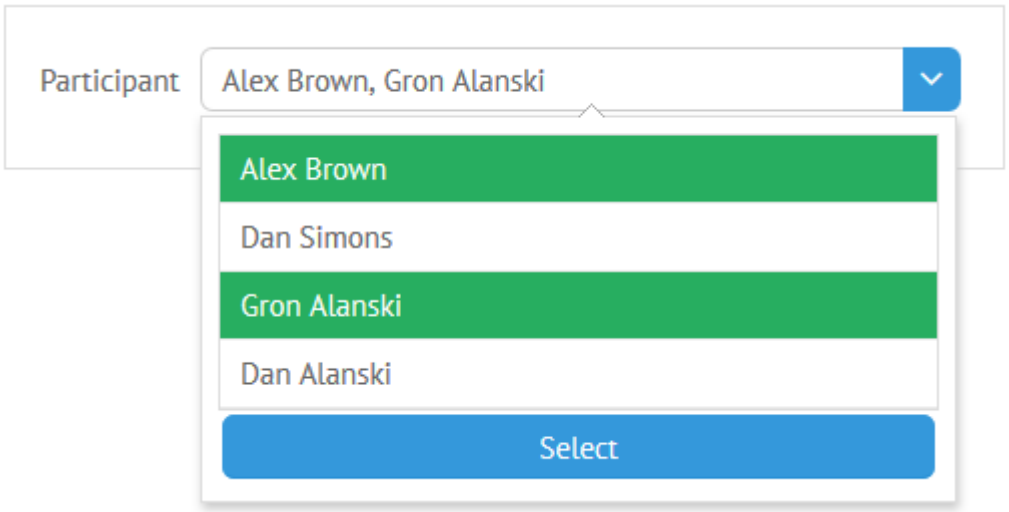


The functionality is available in **Webix Pro** edition only.

Webix offers advanced suggest controls for width extra functionality and greater visual effect:

- **multisuggest** - a standard suggest control based on Webix list with the ability to select **mupliple items** at a time.
- **gridsuggest** - a suggest control based on Webix datatable;
- **datasuggest** - a suggest control based on Webix dataview.

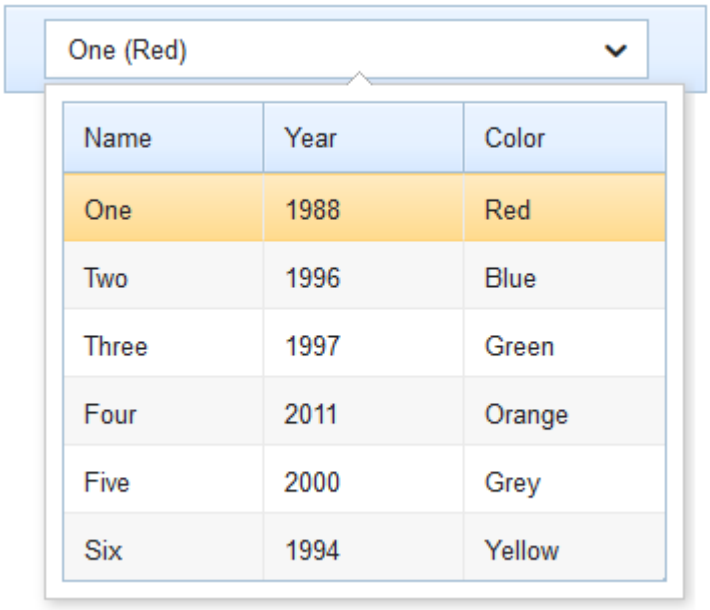
Multisuggest



Multisuggest control is not used standalone, normally. You can find its implementations in the following **Webix Pro** features:

- multiselect form control;
- multiselect datatable filter;
- multiselect component editor.

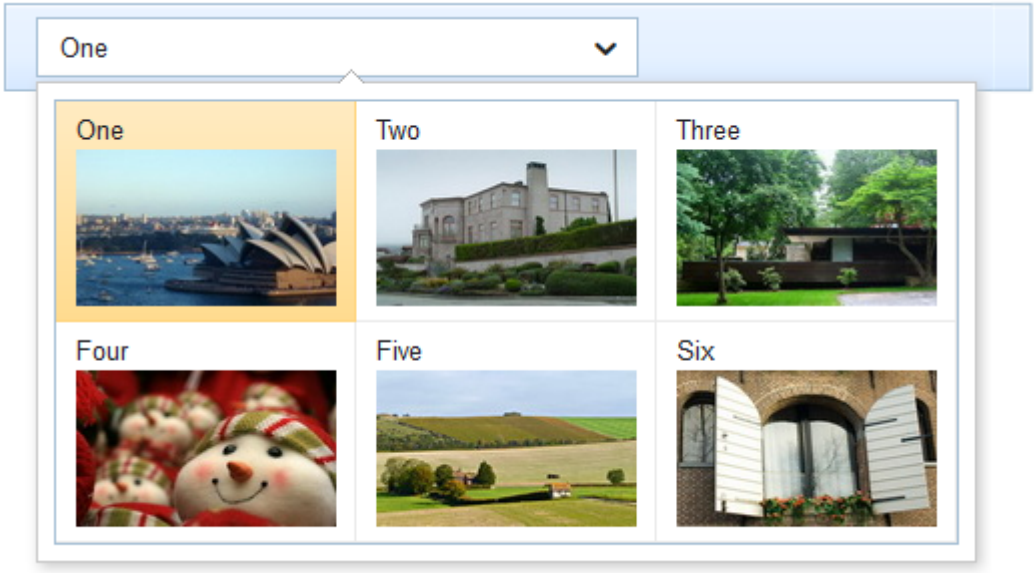
Gridsuggest



RELATED SAMPLE: Grid Suggest

Gridsuggest can be used with form controls as well as with component editors. More about its configuration can be found in the related article.

Datasuggest



RELATED SAMPLE: Data Suggest

Datasuggest can be used with form controls as well as with component editors. More about its configuration can be found in the related article.

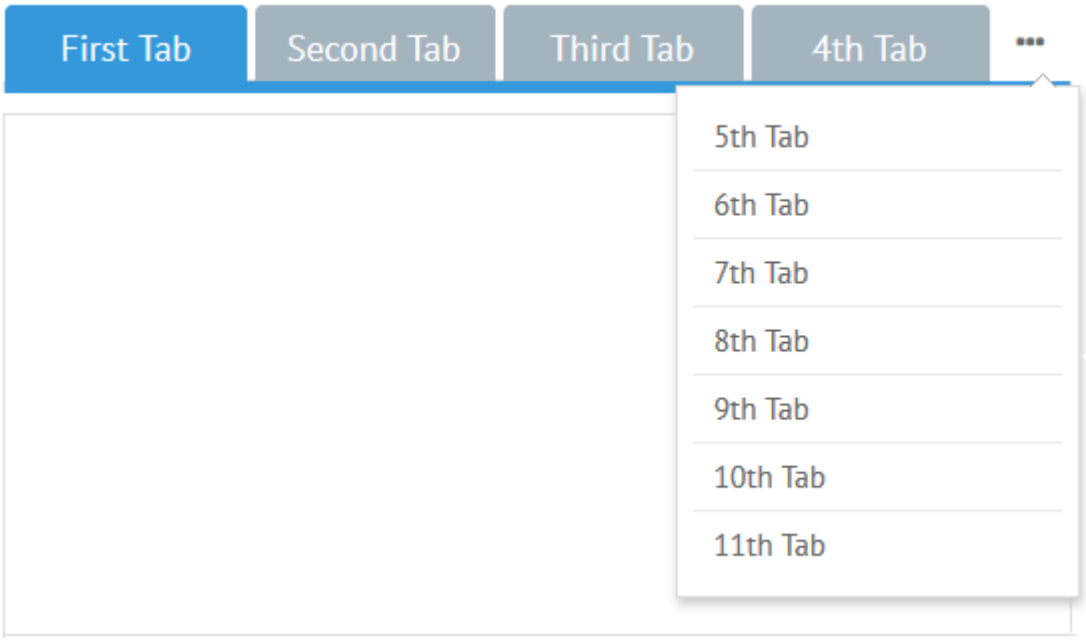
API Reference

- Suggest List API
- Multisuggest List API
- Gridsuggest API
- Datasuggest API

Dynamic Tabbar

Tabbar can take the following modifications:

- Tabbar may have **non-static tab width** and **quantity** that are both changed dynamically on window resizing;
- Tabs can be pushed to a **related popup** while retaining their default behavior.



RELATED SAMPLE: Dynamic Tabbar

Everything is done fully automatically - when window size changes, tabs are minimized and then pushed to a popup. So following code is enough:

```
Default tabbar

{ view:"tabbar", options:["First tab", "Second Tab"] }
```

Customization

- The minimal width of each tab is **100px** by default. It is common for all the tabs of this tabbar and can be changed with the help of **tabMinWidth** property;
- The minimal width of an area to the right that triggers a popup is **40px**. By default an **ellipsis** icon is used. Both parameters can be changed:
  - **tabMoreWidth** sets the width of this area;
  - **moreTemplate** sets other icon/text/HTML of this area. Either function or string can be used;
- The width of the popup is **200px** by default. It can be changed with the **popupWidth** property;
- The default number of list items in the popup is **7**. It can be chaged with the **yCount** parameter. If there are more items in the popup, scroll will appear.

- By default list items feature *#value#* template. Other template can be set by **popupTemplate** property.

```
Customized tabbar

{
    view:"tabbar",
    tabMinWidth:150,
    tabMoreWidth:80,
    moreTemplate:"Show more",
    //or moreTemplate:function(){ return "Show more" },
    popupWidth:250,
    popupTemplate:"#text#",
    yCount:10,
    options:[
        {id:1, value:"First Tab", text:"Some text for first tab"},
        {id:2, value:"Second Tab", text:"Some text for second tab"}
        ...
    ]
}
```

Note that **options** config for tabbar must comnpulsory include **value** property, otherwise it will render *id* on tabs and through an error. Still, there can be other properties that can be rendered in popup list with **popupTemplate**.

More info about Webix data templates can be found in the dedicated documentation article.

Advanced Configuration of Popup Selectors

Combo, richselect and multiselect controls can be highly customized because they are comprised of the following components:

- Webix list from which templating and loading rules, item sizes and look-and-feel are taken;
- Webix popup from which popup size and behaviour are taken.

The combination is called **suggest list** that can be accessed from the control configuration, which allows for setting a number of custom options.

Suggest list is described in detail separately.

All the functionality described below is shown in this sample:

RELATED SAMPLE: Advanced options for select controls ('combo' and 'richselect')

Changing 'options' structure and template

Option data may contain any properies instead of/in addition to default *id-value* pairs (still, **id** attribute is mandatory).

Options attribute is used in another way - it houses list configuration, dataset and template to render this dataset properly:

```
view:"combo",  value:2, options:{
    template:"#name#",
    data:[
       { id:1, name:"Banana"   },
       { id:2, name:"Papai"   },
       { id:3, name:"Apple" }
]
```

The same can be used for default options, moreover, extra text or HTML can be added:

```
view:"combo",  value:2, options:{
    template:"Extra #value#",
    data:[
       { id:1, value:"Banana" },
       { id:2, value:"Papai" },
       { id:3, value:"Apple"}
]
```

Options data may as well contain **multi-line text** with html tags. In this case, the height of a parent component (here:toolbar) should be increased:

```
view:"richselect",
options:[
    { id:1, value:"Here can be some long multi line content <ul><li>item1
<li> item 2</ul>"},
    { id:2, value:"Papai"  }
]
```

RELATED SAMPLE: Select Boxes

Changing Filtering Pattern (combo only)

When a custom template is used for a combo box, filtering should be redefined as well, since by default combo popup is filtered by **value** property.

Options attribute houses configuration of a suggest list in use while its **body** attribute houses a popup list:

```
view:"combo", value:1, options:{
    filter:function(item, value){
if(item.name.toString().toLowerCase().indexOf(value.toLowerCase())===)
            return true;
        return false;
    },
    body:{
        template:"#name#",
        data:[
           { id:1, name:"Banana"},
           { id:2, name:"Papai"}
        ]
    }
}
```

Serverside Options

The easiest way to load options from server is to set a link to the necessary script right in the **options** attribute:

```
view:"richselect", value:"2", options: "server/data.json"
```

For extra customization, see below how to define options dynamically.

Getting Popup and List objects

The controls feature special Api to work with their parts:

```
var popup = richselect.getPopup(); //popup object
var list = richselect.getPopup().getList(); // list object
```

Combo box Api reference

Richselect Api reference

Defining Options Dynamically

Defining options data through the control:

Options can be redefined like any other property:

```
combo.define("options", [ {id:1, value:"Apple"}, {id:2, value:"Banana"}
]);
combo.refresh();
```

Parsing options into a popup list

Options are parsed to popup list under Webix data loading rules:

```
var list = $$("combo1").getPopup().getList();
var new_options = [
      { id:1, name:"Germany"},
      { id:2, name:"Great Britain"}
];
list.clearAll();
list.parse(new_options);
```

Options can be loaded by Ajax as well:

```
webix.ajax().get("server/data.php", function(text, data){
    var options = data.json();

    var list = $$("combo1").getPopup().getList();
    list.clearAll();
    list.parse(new_options);
});
```

Loading options into a popup list

Options are loaded to popup list under Webix data loading rules:

```
var list = $$("richselect2").getPopup().getList();
list.clearAll();
list.load("server/data.json");
```

Syncing options with a DataCollection

Options list can be synchronized with either other component data or Datacollection data:

```
$$("my_combo").getPopup().getList().sync($$("list1")); //pushes this data
into combo popup
```

RELATED SAMPLE: Get Popup View

Data syncing is described in the related article.

Attaching events to a popup list

You can attach functions to list under the common Events Handling pattern:

```
view:"combo", value:1, options:{
    body:{
        data:[
           { id:1, value:"Banana"},
           { id:2, value:"Papai"},
           { id:3, value:"Apple"}
        ],
        on:{
           'onItemClick':function(id){
               webix.message("Clicked: "+this.getItem(id).value);
            }
        }
   }
```

Still, to catch changes in the combo/richselect, it's better to use the related onChange event.

Sizing/positioning of a popup list

The dimensions of a popup list as well as its position in relation to a master control can be customized via **suggest/options** configuration object.

```
view:"combo", options:{
    fitMaster:false,
    body:{
        yCount:7
    }
}
```

Read more about it in the corresponding documentation article.

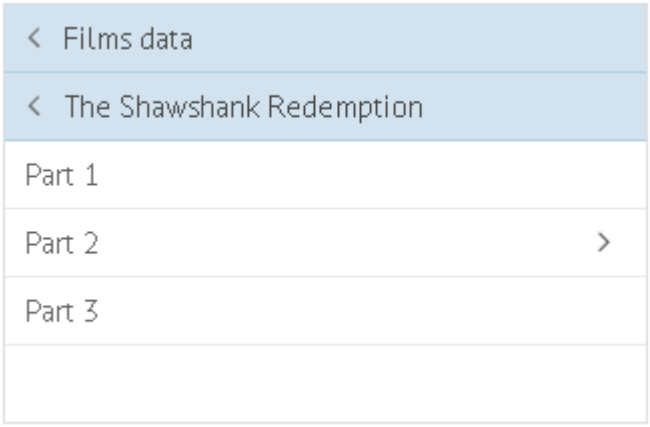
Grouplist

API Reference

- Methods, parameters and events
- Samples

Overview

Ui-related grouplist inherits from list and allows presenting data in groups. Data grouping is **hierarchical**, which means that any group item can start its own group. Hence, this component supports only XML and JSON data formats since they can arrange data in a hierarchical way.



Initialization

```
webix.ui({
    view:"grouplist",
    data:...
});
```

RELATED SAMPLE: Grouplist

Working with Grouplist

Rendering data in grouplist - template peculiarities.

1 . **Single-value hierarchical dataset:**

```
{id:"root", value:"Films data", open:true, data:[
        { id:"1", open:true, value:"The Shawshank Redemption", data:[
            { id:"1.1", value:"Part 1" },
```

```
            { id:"1.2", value:"Part 2", data:[
                { id:"1.2.1", value:"Page 1" },
                { id:"1.2.2", value:"Page 2" },
            ]},
            { id:"1.3", value:"Part 3" }
        ]},
        { id:"2", open:true, value:"The Godfather", data:[
            { id:"2.1", value:"Part 1" },
            { id:"2.2", value:"Part 2" }
        ]}
    ]}
```

Here we have two branches coming out from the root one, each of which having its own branches.

- text values to display are stored in the **value** property;
- for similar datasets we don't need **template** to define which data to display, as there's only one value per item;
- data for a lower hierarchical level in stored in the **data** object;
- initially expanded branches are defined by **open** property with *true* value.

2 . **Multiple-value non-hierarchical dataset:**

Far more often you'll need to display different data values on each level, which means that you'll need to set **template** as well as **grouping pattern** yourself.

```
{"id":1,"title":"The Shawshank Redemption","year":"1994","rank":"1"},
{"id":2,"title":"The Godfather","year":"1972", "rank":"2"},
{"id":3,"title":"The Godfather: Part II","year":"1974","rank":"3"}
```

Grouplist templating differs from that of other data-containing components. There're three properties to observe hierarchy:

- **templateItem** - defined data that will be presented in the normal data record (that is at the end of hierarchical ladder);
- **templateGroup** - defines data that will be presented in group header in the collapsed state;
- **templateBack**- defines data that will be presented in group header in the expanded state;
- **templateCopy** - not connected with data rendering yet defines the pattern according to which data is copied to clipboard.

Each grouping item features the **$count** property that stores the number of items in the group it forms.

```
    $$('grouplist1').getItem(id).$count;
```

Initial data grouping and (optionally) sorting can be set by data scheme with corresponding flags.

```
webix.ui({
    view:"grouplist",
```

```
    data:...,
    templateBack: "Year #value# <br> Results:#$count#",
    templateGroup:" Year #value#",
    templateItem:"#rank#. #title#",
    scheme:{
        $group:function(obj){
                return obj.year; //data is grouped by "year"
        },
        $sort:{ by:"value", dir:"desc" } //sorting by displayed values
    },                                   //"year" or "rank" depending on
heirarchy level
    ...
});
```

RELATED SAMPLE: List: Grouping

To learn more about rendering data through templates, data grouping and data sorting, consult related articles.

Getting Parent and Child Items

Grouplist API features the getOpenState() method to **retrieve IDs** of the items from the **currently opened branches**, both parent items IDs and that of its children.

```
var state = $$('grouplist1').getOpenState();
if(state){
    result = JSON.stringify(state);
    webix.message(result);
    }
};
```

The methods returns IDs of all the opened parents items (according to hierarchy ladder) as well as IDs of items belonging to the curretly opened branch.

Thus, the return object contains parent ID array and children array:

```
Hierarchical dataset state

{
    "parents":["root","1","1.2"],
    "branch":["1.2.1","1.2.2"]
}
```

RELATED SAMPLE: Getting Grouplist State

Working with Grouplist

- Data Object Operations
- Data Loading.
- Defining Data Template.

- Adding/Deleting Items.
- Editing Data.
- Data Filtering and Sorting
- Selection
- Paging
- Adding Active Elements to Grouplist Items

Note that there's no built-in possibility to edit data with grouplist. You should create a prototype editlist object beforehand.

Related Articles

- Sizing Components
- Resizing
- Redefinition of the Components
- List
- Unitlist
- X-list
- Grouplist CSS Image Map

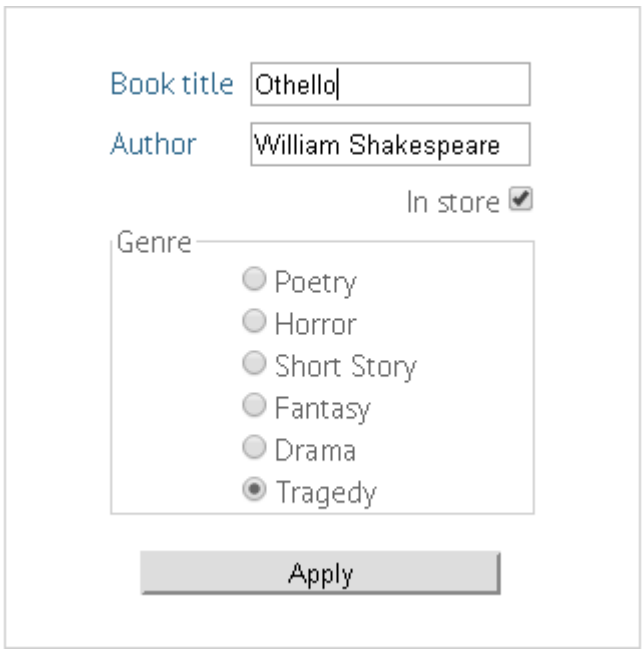
HTMLform

API Reference

- Methods, properties and events
- Samples

Overview

Though the webix library has its own form component, there exists a possibility to create a separate form made with the help of pure HTML. This is done within the special **htmlform** component.



Initialization

There're three ways of embedding a form into the component:

1 . Through content property:

Content property may either point the **ID** of **DIV container** where the form lies or to **document.body** if it is included directly into your web page body.

```
HTMLform, content

<div id="areaA">
        <input type="text" name="title" value="" placeholder="Book title"
/><br/>
```

```
        <input type="text" name="author" value="" placeholder="Author"
/><br/>
</div>
<script type="text/javascript">
webix.ui({
    view:"htmlform",
    content: "areaA" //coincides with ID of a *div* container
    //or
    content: document.body // in case the form isn't included into any DIV
    ...
});
</script>
```

RELATED SAMPLE: Initialization in Document Body

2 . With the help of template property:

- Template value may inlude the **ID** of a **DIV container** where form fields lie

```
HTMLform, template

<div id="tpl">
            <input type="text" name="title" value="" placeholder="Book
title" /><br/>
            <input type="text" name="author" value="" placeholder="Author"
/><br/>
</div>
<script type="text/javascript">
webix.ui({
    view:"htmlform",
    template: "html->tpl" //coincides with ID of a *div* container
        ...
});
</script>
```

RELATED SAMPLE: Initialization in Layout

- or it points to **external html-file** with a form:

```
HTMLform, http

webix.ui({
    id: "htmlform1",
    view:"htmlform",
    template: "http->data/form.html",
    ...
});
```

RELATED SAMPLE: Using HTTP-template

3 . Initialization from HTML or XML markup



The initialization from **XML markup** is available in **Webix Pro** edition only.

In this case, standard rules of markup initialization are used and the **markup** slightly **alters**:

- **HTML**: **form** tag receives new attributes;
  - **data-view** to indicate the component (here it is an *htmlform*);
  - standard htmlform properties with 'data' prefix (*data-id, data-height*, etc.);
- **XML**: There're special tags with 'x:' prefix:
  - **< x:ui >** to indicate a webix UI constructor
  - **< x:htmlform** > to indicate a component you want to init.

Form elements are created accirding to standard HTML.

HTMLform, HTML markup

```
<form data-view="htmlform" data-id="htmlform1" data-height="350">
    <input type="text" name="title" value="" placeholder="Book title"
/><br/>
    ...
</form>
```

HTMLform, XML markup

```
<html xmlns:x>
    <x:ui>
        <x:htmlform id="htmlform1" height="350">
            <input type="text" name="title" value="" placeholder="Book
title" /><br/>
                ...
        </x:htmlform>
    </x:ui>
</html>
```

Script part of the document includes the initing code:

```
webix.ready(function(){
    webix.markup.init(document);
});
```

Regardless of the initialization pattern, all the commom HTMLform functions work fine:

RELATED SAMPLE: HTML initialization : HTML form

4. Initialization from an external file loaded by Ajax.

```
webix.ui({
```

```
    view:"htmlform",
    src:"myform.php",
    ...
});
```

Working with HTMLform

- Data Validation
- Form and Htmlform Treatment
- File Uploading with Form
- Data Binding

Sending Form Data

Form data can be sent to server in either of the following ways:

1 . Using Webix Ajax helper:

```
webix.ajax().post("some.php", form.getValues());
//with callback
webix.ajax().post("some.php", form.getValues(), function(text, data, xhr){
});
```

2 . Using webix.send method that emulates HTML form submitting:

```
webix.send("come.php", form.getValues());
```

3 . Indirectly, via the **bound master** component or DataCollection:

The method is good when the form is used for editing the data of the main component (datatable, tree, list, etc.). In this case not the form data matters but the data of the main component. Form saves the data to the master component while the master handles the serverside part.

```
//data from selected list item is pushed to the form
form.bind(list);

//pushes the data back to list item
form.save()
```

Data binding concept is described in a separate documentation article.

Related Articles

- UI-related Form

- Initializing from HTML and XML

IFrame

API Reference

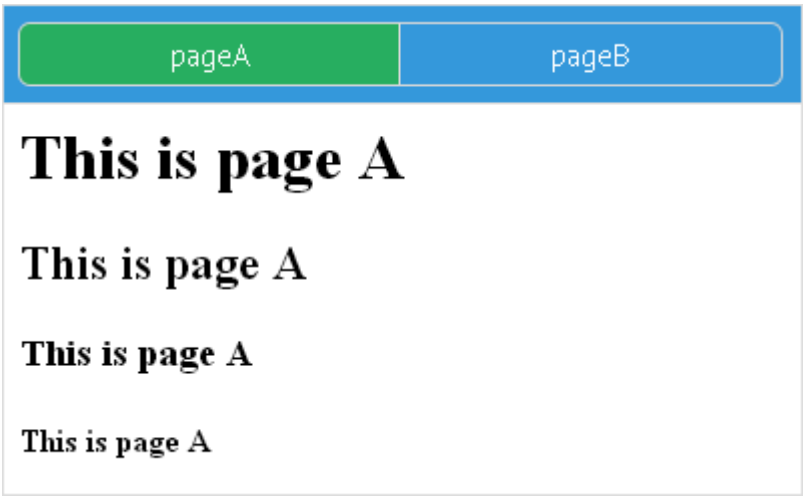
- Methods, properties and events
- Samples

Overview

Ui-related IFrame is a component that helps include an i-frame, or 'inline frame' into the page. Basically, it's a floating frame on the webpage that contains an external web document. The component, as a rule, is smaller than the page it's added into, so vertical and horizontal scrollbars appear.

You can insert more than one IFrame into the view provided that you initialize a switching control for them.

The component inherits from view.



Initialization

```
webix.ui({
    view:"iframe", id:"frame-body", src:"data/pageA.html"
});
```

RELATED SAMPLE: IFrame

- **src** (path) - defines external site url.

Working with IFrame

Data loading

The url to show in the iFrame can be defined

- as value of the src **property** (shown above);
- as parameter of the **load** method.

```
$$("frame-body").load("http://google.com")
```

Switching between iframes in a single view

Simply put, this is the basics of multiview functionality.

Make use of any of the **integral buttons** (segmented, tabs or tabbar) to enable switching. The values for button options will be necessary URL-s.

```
{ view:"segmented", id:"control", options:[
    { id:"data/pageA.html", value:"pageA"},
    { id:"data/pageB.html", value:"pageB"}
]}
```

Then, attach a switching function to the button. The event referring to a tab mouse click is called **"onAfterTabClick"**.

```
$$("control").attachEvent("onAfterTabClick",function(sid){
    $$('frame-body').define("src", sid);
});
```

The event fires on clicking any tab, takes its ID (sid) as parameter and sets the source for external webpage equal to this ID.

Related Articles

- Sizing Components
- Redefinition of Components
- Event Handling
- Multiview

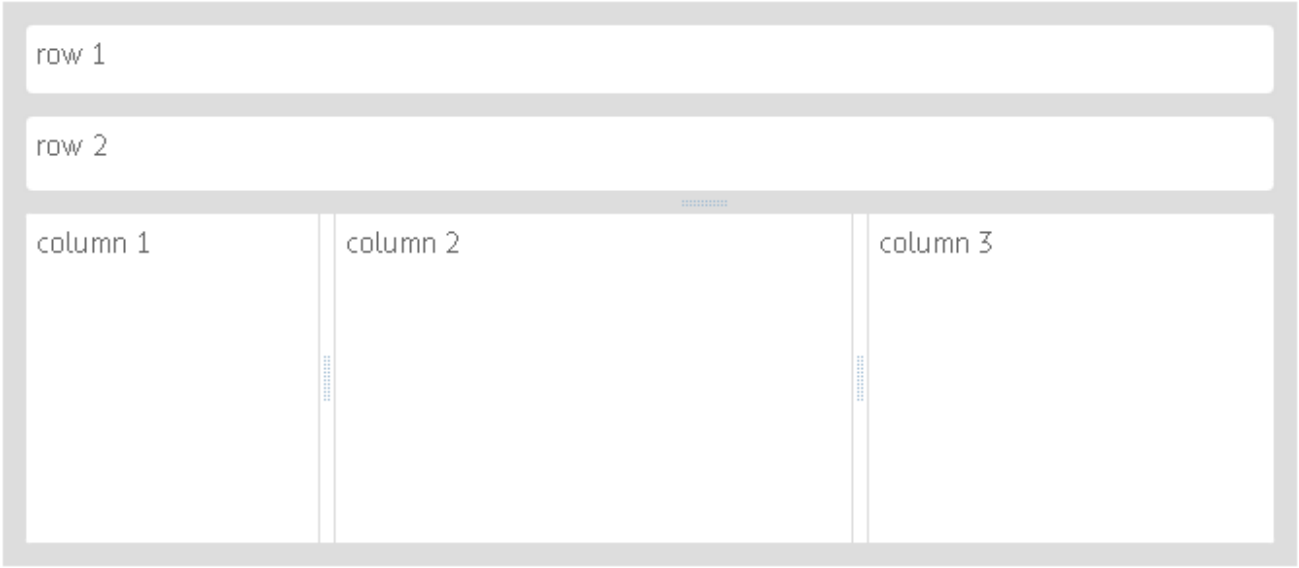
Layout and Resizer

API Reference

- Methods, parameters and events
- Samples

Overview

Layout is a concept element that doesn't contain anything in itself yet determines the way 'meaningful' components are arranged. It divides the page into **rows** and **columns** where all other components are placed.



Initialization

Layout consists of rows and columns separated from each other by borders (except for a borderless layout). **Rows** contain an array of view objects arranged horizontally while **cols** is an array of vertically arranged objects. Border setting and border types are described in the corresponding chapter of the manual.

```
webix.ui({
    container:"layout_div",
    id:"mylayout",
    type:"space",
    rows:[
        {...},
```

```
        {cols:[
            {...},
            {...}
        ]}
    ]
})
```

RELATED SAMPLE: Layout

When setting layout, you can easily omit the **view:"layout"** line and start setting rows and cols at once:

```
webix.ui({
    type:"wide",
    rows:[
        {...}, {...}
    ]
});
```

- **Container** property defines the ID of a DIV element where you place the layout. Though it is an optional parameter, it makes sizing and positioning simpler.
- Layout **ID** should be specified if you are going to work with it later.

Horizontal and Vertical Layout

Horizontal layout:

row 1
row 2
row 3

Three-row layout

```
webix.ui({
    container:"layout_div", //corresponds to the ID of the div block
    rows:[
        { template:"row 1" }, //here you place any component you like
        { template:"row 2" },
        { template:"row 3" }
    ]
});
```

Vertical layout:

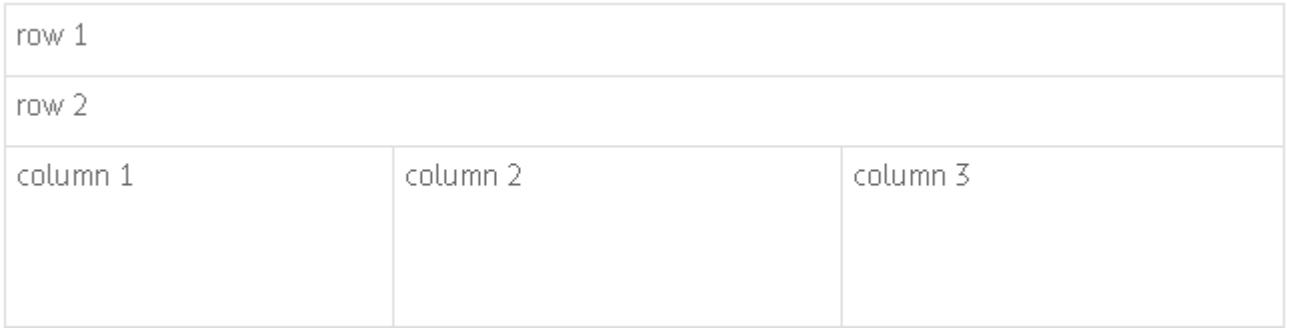



```
Three-column layout

webix.ui({
    container:"layout_div", //corresponds to the ID of the div block
    cols:[
        { template:"col 1" }, //here you place any component you like
        { template:"col 2" },
        { template:"col 3" }
    ]
});
```

Complex layout

Any row can be divided into columns and any column may contain rows. This is how simple layouts are nested into each other:




```
webix.ui({
    container:"layout_div", //corresponds to the id of the div block
    rows:[
        {template:"row 1"}, //here you place any component you like
        {template:"row 2"},
        { cols:[
            { template:"col 1" },
            { template:"col 2" },
            { template:"col 3" }
        ]}
    ]
});
```

RELATED SAMPLE: Limits on Resizing

Resizer Lines

With this library you can make draggable border, or so-called, **resizer** lines between layout rows and columns. The resizer line can be moved in both directions and change the dimensions of the components on both sides of it.

Resizer lines don't actually change the component's size. They only help users adjust the component's visual scheme to their needs.

It's recommended to set **minWidth** and **maxWidth** parameters for columns subject to resizing (**minHeight** and **maxHeight** for the rows). It prevents them from collapsing as when the minimal width or height value is achieved, the resizer line cannot be moved anymore.

```
webix.ui({
    container:"layout_div",
    id:"layout",
    rows:[
        {...}, //1st row, non-resizable
        {...}, //2nd row
        {view:"resizer"},
        {...}, //3rd row
        {view:"resizer"},
        {cols:[  //4th row
            { // 1st column
                width:150,
                minWidth:50,
                maxWidth:250
            },
            {view:"resizer"},
            {....}, //2nd column
            {....} //3rd column
        ]}
    ]
})
```

RELATED SAMPLE: Layout and Resizer

Responsive layout behavior depending on window/screen size in discussed separately.

Headers

Check how to make headers for different components here.

```
webix.ui({
    rows:[
        {   view:"template", //optional
            template:"Header for Column 1",
            type:"header"
        },
        {...}
    ]
})
```

Related Articles

- Sizing Components
- Resizing
- Setting Borders to the Components
- Adding Headers to Components
- Multiview
- Accordion
- Scrollview
- Carousel
- Layout CSS Image Map

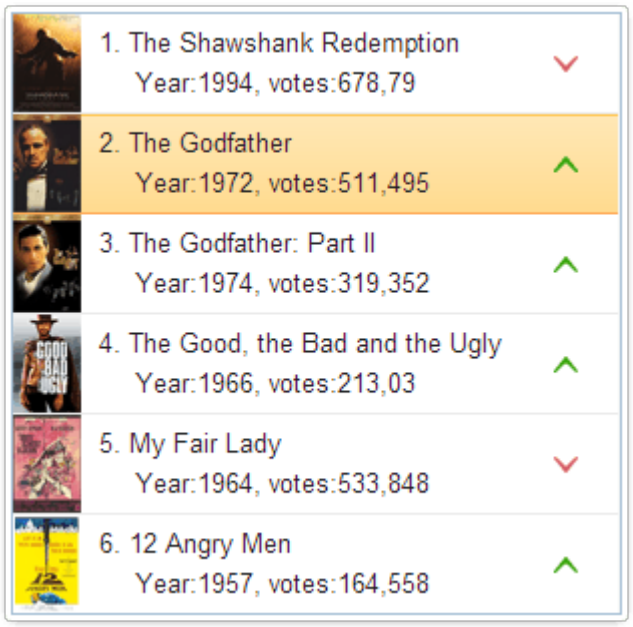
List

API Reference

- Methods, properties and events
- Samples

Overview

List is a ui-related components that inherits from view and presents listed data items. The webix library offers three variations of list apart from the standard one - grouplist, x-list and unitlist.



Initialization

```
webix.ui({
    view:"list",
    template:"...",
    data:..//variable || path || dataset
});
```

RELATED SAMPLE: List

You can load data in any of the supported data formats.

Working with List

- Data Object Operations
- Data Loading.
- Defining Data Template.
- Adding/Deleting Items.
- Editing Data.
- Data Filtering and Sorting
- Selection
- Paging
- Adding Active Elements to List Items

Note that there's no built-in possibility to edit data with List. You should create a prototype *editlist* object beforehand.

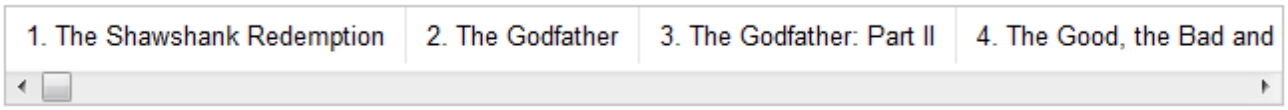
Related Articles

- Setting Dimensions for the Components
- Dynamic Resizing and Adjustment
- Redefinition of the Components
- List CSS Image Map

X-list

Overview

UI-related x-list inherits from list and differs from it in the way of data presenting: while all lists arrange items vertically, x-list offers the horizontal pattern.



RELATED SAMPLE: Horizontal List

Initialization

```
webix.ui({
    view:"list",
    scroll:"x",
    layout:"x", // responsible for horizontal arrangement
    select:true,
    data: ...
});
```

Working with X-list

- Data Loading.
- Defining Data Template.
- Adding/Deleting Items.
- Editing Data.
- Data Filtering and Sorting
- Selection
- Paging
- Adding Active Elements to List Items

Note that there's no built-in possibility to edit data with list. You should create a prototype *editlist* object beforehand.

API Reference

Methods, properties and events

Related Articles

- Sizing Components
- Resizing
- Redefinition of the Components
- List
- Unitlist
- Grouplist
- XList CSS Image Map

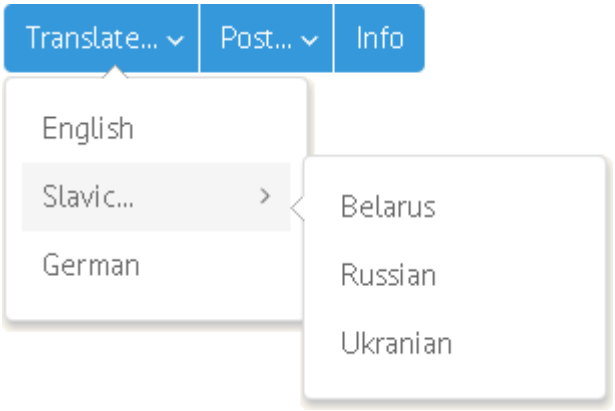
Menu

API Reference

- Methods, properties and events
- Samples

Overview

UI-related pulldown **Menu** inherits from list and lets users select the necessary item from the list of grouped items (submenus). Each element of a submenu can start its own submenu.
A dynamic **Submenu** becomes visible only when you put a mouse pointer over the corresponding item of the main menu (on 'onMouseOver' event) and hides as soon as you remove mouse pointer ("onMouseOut").



Initialization

Menu object is stored in the **data** property.

```
webix.ui({
    view:"menu",
    id:"my_menu",
    subMenuPos:"right",
    layout:"y",
    data:[ //menu data
        { value:"Translate...", submenu:["English", "Slavic...",
"German"]},
        { $template:"Separator" },
        { value:"Post...", submenu:[ "Facebook", "Google+", "Twitter" ]}
    ],
```

```
    type:{
            subsign:true,
            height:50
    }
});
```

RELATED SAMPLE: Basic Menu

Submenu configuration may include:

- **subMenuPos** (string) - sets position of pulldown submenus that will appear on mouse over;
- **submenuConfig** (object) - sets common configuration for all the submenus in this menu regardless of their nesting level;
- **layout** (string) - sets the arrangement of menu items: **x** (default) for a horizontal menu, **y** for a vertical one;
- **$template** (string) - flag that indicates a separator between groups of menu items. Possible types are as follows:
  - **"Separator"** - a grey separating line;
  - **"Spacer"** - an empty item with white background that doesn't change on mouse over.
- **type** - object that contains setting for menu items, namely:
  - **subsign** (boolean) - sets an arrow for an item containing a submenu;
  - **height** and **width** (number) of each item;
- **openAction** (string) - alters the way of submenu opening - **"click"**. If set, you should first click menu item to enable its opening and hiding on "onMouseOver" and "onMouseOut" events;
- **template** - (string, function) - rules of data presentation. See Data templates article for details.
- **data** (object) - inline data desires format. See Loading data article for details.

RELATED SAMPLE: Showing Menu on Mouse Click (dropping down on hover is disabled)

Working with Menu Items

Configuring Menu Items

A menu/submenu item can be a **text item** including the one that starts its own **submenu** and any **Webix component** specified by ID.

Menu items are stored in the **data** array, each item being an object. Submenu items are stored in the **submenu** array. Each menu item may have the following attributes:

- **id** - item id.If omitted, it's given an auto-generated one;
- **value** - defines text value for an item;
- **href** - defines a link for an item;
- **config** - defines configuration of a child submenu popup (if any). The configuration overrides the one

defined via **submenuConfig**;

- **badge** - sets an orange information circle that notifies about the number of new activities for this menu item (e.g. messages).

The easiest way to define menu items is to pass them as a simple array:

```
{ view:"menu", data:["Google", "Facebook", "Twitter"] }
```

Or an associative one:

```
{ view:"menu", data:[
    { id:1, value:"Google"},
    { id:2, value:"Facebook"},
    { id:3, value:"Twitter"}
]}
```

Links in Menu Items

Each menu and submenu item can be supplied with a **link** specified in the data object by **href** and **target** (optional) attributes:

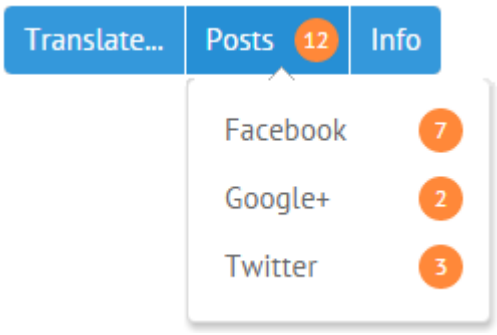
```
view:"menu",
data:[
    { id:"1", value:"Imitation of Spenser", href: "#verse1",
target:"_blank"},
    { id:"2", value:"The Human Seasons", href: "#verse2"}
]
```

If not specified, target will be set as empty string.

RELATED SAMPLE: Menu hrefs

Badges in Menu Items

Badges add info circles to menu items with the number of entities you'd like to associate with this or that item (e.g. messages):




```
view:"menu",
data:[
```

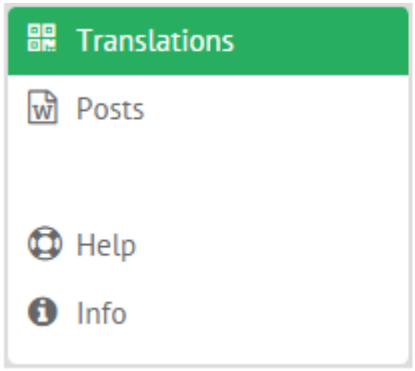
```
    { id:"1",value:"Translations", badge:"1"},
    { id:"2",value:"Posts", badge:"12" },
    { id:"3",value:"Info", badge:"24" }
]
```

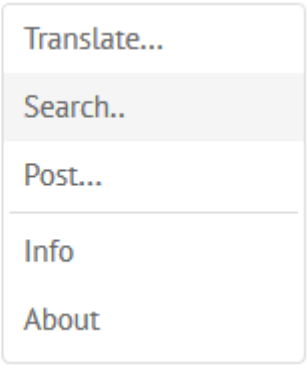
RELATED SAMPLE: Menu with Badges

Separators in Menu

To separate one group of items from another one in vertical menu, use the property **$template** with one the possible values:

- **"Spacer"** - adds horizontal space between menu items;
- **"Separator"** - add a horizontal grey line between menu items.






```
Spacer in vertical menu

view:"menu",
layout:"y",
data:[
    { id:"1", value:"Translations", icon:"qrcode", badge:20 },
    { id:"2", value:"Posts", icon:"file-word-o", badge:3 },
    { $template:"Spacer" },
    { id:"3", value:"Help", icon:"support"},
]
```

RELATED SAMPLE: Vertical Menu

```
Separator in vertical menu

view:"menu",
layout:"y",
data:[
    { value:"Translate..." },
    { value:"Post...", submenu:[ "Facebook", "Google+", "Twitter" ]},
    { $template:"Separator" },
    { value:"Info" }
]
```

RELATED SAMPLE: Basic Menu

Customizing Submenus

Submenu items can be set as an **array of values**:

```
{ id:"2",value:"Translate...", submenu:[ "English", "Slavic", "German" ]},
```

Or, as an **array of menu objects**:

```
{ id: "1.2", value:"Slavic...", submenu:[
    {id: "1.2.1", value:"Belarus"},
    {id: "1.2.2", value:"Russian"},
    {id: "1.2.3", value:"Ukranian"}
]}
```

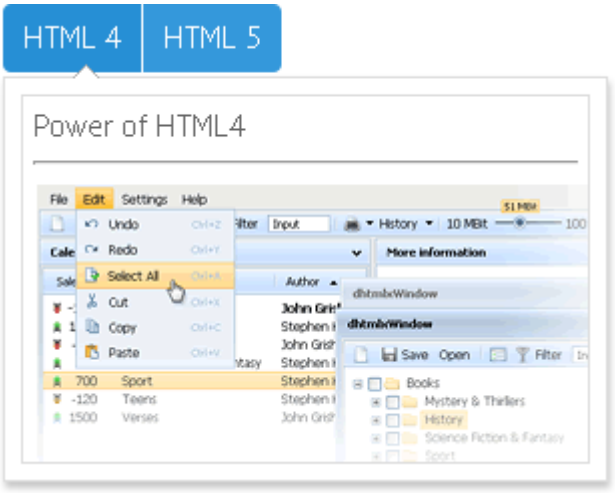
Any nested submenu (a popup) can be configured separately with the help of a **config** attribute of its parent:

```
view:"menu",
submenuConfig:{
    width:400
},
data:[
    { id:"2",value:"Custom...",
        config:{
            width:500,
            on:{
                onItemClick:function(id){
                    webix.message("Submenu click: "+id);
                }
            }
        },
        submenu:[ "Facebook", "Google+", "Twitter" ]
     }
]
```

This configuration overrides common configuration for submenus defined by **submenuConfig**.

RELATED SAMPLE: Configuring Submenus

Webix Component as Menu Item



Submenu parameter may point as well to any Webix component that has been initialized previously:

```
webix.ui({
    id:"details1",
    view:"context", width:300, height:200,
    body:{ content:"html1" }
});

webix.ui({
    view:"menu",
    data:[
        { value:"HTML 4", submenu:"details1" },
        { value:"HTML 5", submenu:"details2" }
    ]
});
```

RELATED SAMPLE: HTML Elements as Menu Items

The same way you can initialize Webix **submenu** component separately and then point to it within *submenu* property:

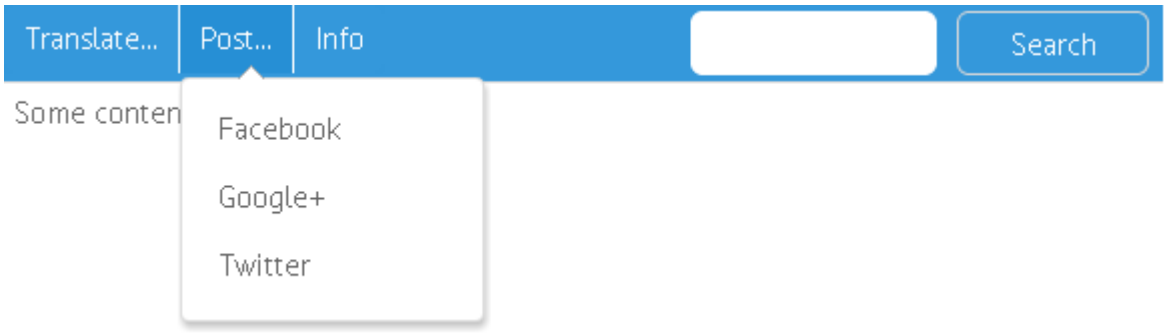
```
webix.ui({
    view:"submenu", id:"test", data:[
        { id:"1.1", value:"English"},
        { id:"1.3", value:"German"}
    ]
});

webix.ui({
    view:"menu",
    data:[
        { id:"1", value:"Translate...", submenu:"test"},
        { id:"2", value:"Post..."
    ]
});
```

Each submenu item can take its own submenu component.

MenuBar

Menu takes after a toolbar in appearance, so the best way to create a menubar is to initialize menu and toolbar in neighbouring columns:




```
webix.ui({
    cols:[ menu, toolbar ]
});
```

RELATED SAMPLE: Menu in Toolbar

Hiding and Disabling Menu Items

Hiding

To hide and show menu item, you should apply **hideItem()** and **showItem()** to the menu component while specifying item ID as function parameter.

The event that triggers function execution can be **checkbox state change** in the Webix tree populated with the same data as menu, provided that checkbox item ID = related menu item ID.

```
$$("tree1").attachEvent("onItemCheck",function(id,state){
    if (state) //if checked
        $$("top_menu").hideItem(id);
    else
        $$("top_menu").showItem(id);
});
```

As a result, only "unchecked" menu items are shown.

RELATED SAMPLE: Hiding Menu Items

Disabling

To disable menu item, you should apply special **disableItem()** and **enableItem()** functions to menu passing ID of the necessary item into them. The methods can be triggered by changing state of a checkbox which ID equals to the ID of the corresponding menu item.

```
$$("tree1").attachEvent("onItemCheck",function(id,state){
```

```
    if(state) //if checked
        $$("top_menu").disableItem(id);
    else
        $$("top_menu").enableItem(id);
});
```

In essence, disabling and enabling is performed via **CSS classes**, that's why menu items get a **template** that helps display normal and disabled item differently:

```
view:"menu",
template:function(obj){
    if(obj.disabled)
        return "<span class='disabled'>"+obj.value+"</span>";
    return obj.value;
}
```

RELATED SAMPLE: Disabling Menu Items

Event Handling with Menu and Submenu Items

All menu and submenu items can be accessed by their IDs.

1 . Any item can be accessed directly with **getMenuItem()** method that takes item ID as parameter and returns item object:

```
//getting menu item value by its ID
$$('menu1').getMenuItem(id).value;
```

RELATED SAMPLE: Basic Menu

2 . At the same time, menu items can be derived through submenu object returned by **getSubMenu()** method for an item with the specified ID. If there's no submenu for an item - an object of the calling component will be returned.

```
var menu = $$("menu1").getSubMenu(id);
var item = menu.getItem(id).value;
```

RELATED SAMPLE: HTML Elements as Menu Items

Both methods can be used in either of **menu inner events**, for instance, click events:

- **onMenuItemClick** - fires on clicking all menu items regardless of hierarchy level. Ignores disabled items;
- **onItemClick** - standard event that fires on clicking any item of the same hierarchy level. Fires for disabled item as well.

```
General "onMenuItemClick" handler
```

```
view:"menu",
data:[...],
on:{
    onMenuItemClick:function(id){
        webix.message("Click: "+this.getMenuItem(id).value);
    }
}
```

RELATED SAMPLE: Basic Menu

```
Submenu specific "onItemClick" handler

view:"menu",
data:[
    { id:"1",value:"Custom...",
        config:{
            width:250,
            on: { onItemClick:function(id){
                webix.message("Submenu click: "+id);
            }}
        },
        submenu:{...}
    }
]
```

RELATED SAMPLE: Configuring Submenus

Related Articles

- Event Handling
- ContextMenu
- Menu CSS Image Map

Multiview

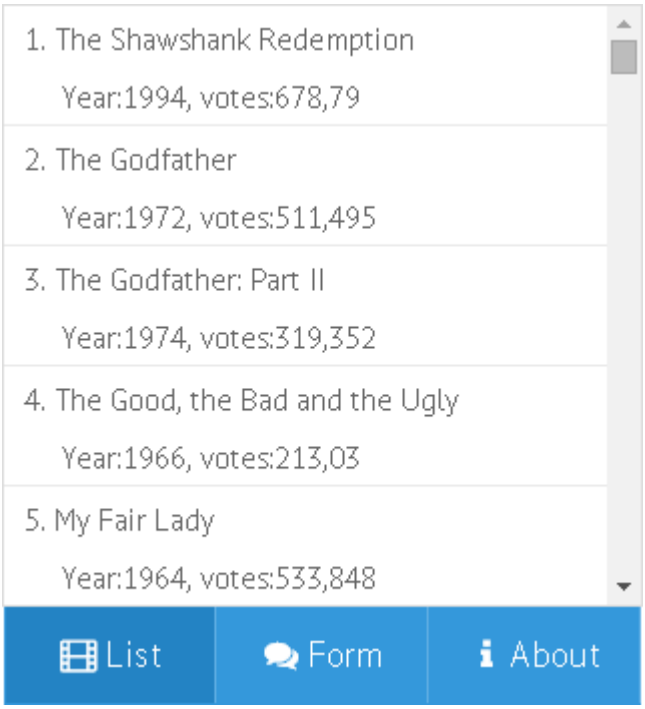
API Reference

- Methods, properties and events
- Samples

Overview

Multiview helps you efficiently use space on the page by placing different views into one and the same area. Only one view is visible at a time. The others are accessible with the help of dedicated buttons.

Multiview inherits from layout and if combined with a switching control like a tabbar, forms a tabview.



Initialization

```
Multiview with three views

webix.ui({
  view: "multiview",
  cells: [
    {id: "listView", view: "list", .... },
    {id: "formView", view: "htmlform", .... },
```

```
    {id: "emptyView" }
  ]
});
```

RELATED SAMPLE: Multiview Tabbar with Icons

The direct initialization of the component (**view:"multiview"**) is optional and you can just init **cells** and the view will be automatically recognized.

```
webix.ui({
  cells:[
    {id: "listView", view: "list", .... },
    {id: "formView", view: "htmlform", .... },
    {id: "emptyView"}
  ]
});
```

Working with Multiview

Adjusting tab dimensions

The **dimensions** of multiview cells are adjusted to their content. By default, the whole multiview (all the cells) take the size of smallest cell. To adjust it to the biggest cell, use **fitBiggest** property:

```
webix.ui({
  view: "multiview",
  cells: [
    ...
  ],
  fitBiggest:true
});
```

Managing animation

My default multiview cells are switched with horizontal animation of a "slide" type. To disabled animation, or chnge animation type, configure the **animate** property:

```
webix.ui({
  view: "multiview",
  cells: [
    ...
  ],
  animate:false
});
```

RELATED SAMPLE: Disabling Animation

Learn more about animation types in the Animation article.

Switching between views

1 . **Tabbar and Segmented Integral Buttons.**

Here switching is implemented with the help of special library controls that have built-in switching functionality as soon as they are bound with multiview cells by ID of these cells.

2 .**Tabview Functionality.**

Multiview built-in means of changing the view cells. Look very much like the tabbar from the previous point.

3 . **Functions for Switching Actions**.

Here you use common methods (**show and back**) and create custom buttons each of which can take you to the necessary view. **Show()** method in conjunction with its counterpart **hide()** are as well used for changing visibility of UI components.

Saving inactive views

When dealing with iframe content, e.g. WYSIWYG editors (CKEditor), Multiview removes inactive views when they become hidden.

To solve this problem, set the *keepViews* config property to true.

```
webix.ui({
  view: "multiview",
  cells: [
    ...
  ],
  keepViews:true
});
```

History API

Learn how to "memorize" the currenty opened tab in the related article.

Related Articles

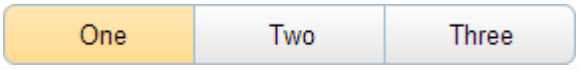
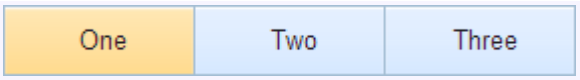
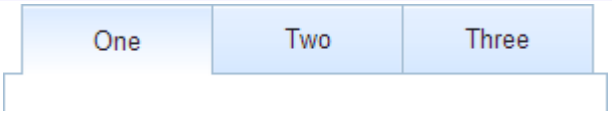
- Sizing Components
- Resizing
- Setting Borders to the Components
- Controls
- Tabview Functionality

- Layout
- Scrollview
- Accordion
- Carousel

Controls to Switch Between View Cells

There are two buttons to switch between multiview cells at your disposal - tabbar and segmented.

Table 1 Switch controls

`view:"segmented"`	
`view:"tabbar"`	
`view:"tabbar", multiview:true`	

Both of them are integral buttons that can contain as many options as you need. It is the option values that are connected to the IDs of multiview cells, which enables switching.

The difference between the buttons lies in their design. See Controls Guide to learn more about them.

In case of three views you should make three switching possibilities, i.e put three elements into the **options** array of either of the stated controls. Make sure that option IDs correspond to the **IDs** of the view cells. It ensures that every time you click on the necessary tab/button segment, the view changes to the corresponding one.

Note that you don't need to attach "switching" functions to the tabs. Everything is done automatically as soon as you set the *multiview* parameter to a *true* value.

```
webix.ui({
    rows:[
        { cells:[ {id:"listView", ... },
             {id:"formView", ... },
             {id:"emptyView", ... }]
        },
        { view:"tabbar", value: 'listView', multiview:true,
            options: [
                {value: 'List', id: 'listView'},
                {value: 'Form', id: 'formView'},
                {value: 'Empty', id: 'emptyView'}
             ]
        }
    ]
});
```

RELATED SAMPLE: Tabbar

The **initially visible tab** is set by the tabbar/segmented value property that contains the ID of the necessary tab.

Related Articles

- Tabview Functionality
- Functions to Control Switching

Common Functions to Switch Between Views

When switching views programmatically, you can refer either to the cells and view in them or to the switching controls.

Setting Value for the Tabbar/Segmented/Tabview

Either of the controls as well as tabview have common initialization pattern where **tabs** are stored in the **options** array:

```
{view:"tabbar", id:"tabbar1", multiview:true, options:
        [{id:"1", value:"Tab1"}, {id:"2", value:"Tab2"}]
}
```

Tabs are connected to the multiview cells, so if you set the value for button, the dedicated view will be opened:

```
$$("tabbar1").setValue("2"); //the 2nd tab with "Tab2" value will be set
```

Showing the Necessary View

These functions are used to show the specified view:

- **show();** - makes the specified view visible. Is called from the needed view;
- **back();** - switches to the view you've come from. Is called from the multiview object.

To enable switching between views you can:

- make use of a JS button from this lib or a standard HTML button and attach the function to it;
- trigger the switching function on any event in the app. For instance, "onItemClick" event allows clicking any component (not necessarily a button) and make the specified view visible.

```
Two-Cell Multiview

webix.ui({
    view:"multiview",
    id:"views",
    cells:[
        { id:"listView", view:"list" },
        { id:"formView", view:"form" }
           ]
 })
```

```
 //html button
 <input type='button' onclick='$$("cell_a").show()' value="show cell A">

//list item clicking enables the switching
$$("listView").attachEvent("onItemClick",function(id){
        $$('formView').show();

//back navigation - fires alongside with a form-saving function
function save(){
        $$("formView").save();
        $$("views").back();
       }
```

RELATED SAMPLE: 'Show/Back' Navigation

Nested Multiview

If a cell is a multiview itself, i.e. contains several cells of its own, the switching function requires **true** argument to get to these lower-level views.

```
rows:[
     { cells:[ //1st cell
               {
                id:"listView",
                view:"list"},
               {
                 id:"chartView", //2nd cell
                 cells:[
                      {
                       view:"chart", 1st sub-cell
                       type:"bar"},
                       {
                         view:"chart", //2nd sub-cell
                         type:"donut"}
                       ]
               }
            ]
     }]
```

To switch from listView to any of ChartView cells, you need to call the **show(true);** function:

```
<input
    type='button'
    onclick='$$("chart1").show(true)'
    value="show chart1 and parents"
>
```

At the same time, if you switch between chart1 and chart2 that lie on the same level, you needn't refer to the

parent view and can do with the **show();** function.

RELATED SAMPLE: Nested Multiviews

Related Articles

- Tabview Functionality
- Controls to Switch Between View Cells

Available only in **PRO Edition**

Organogram

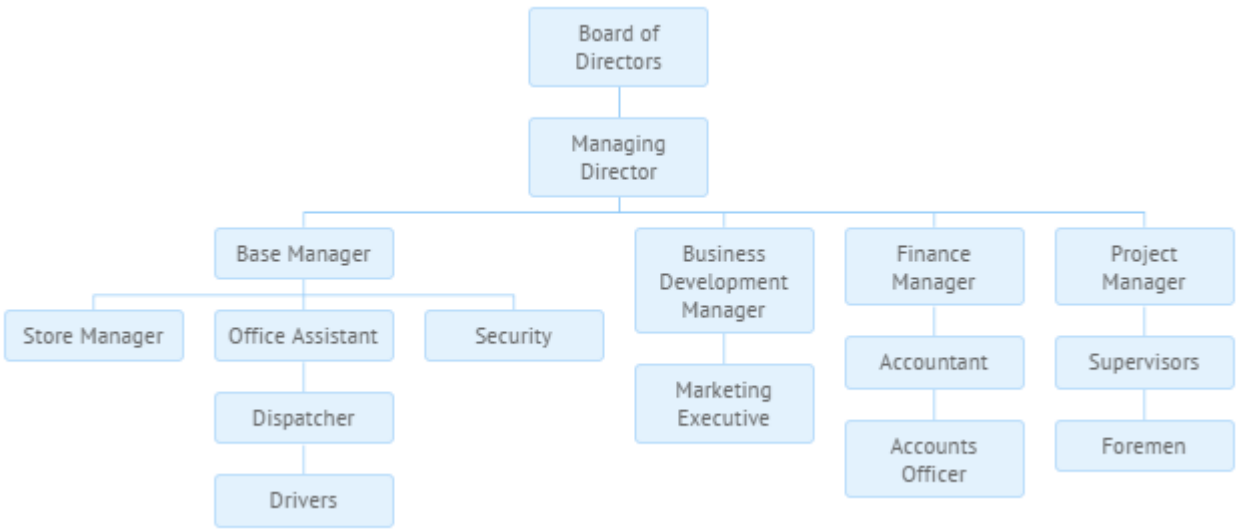
API Reference

- Methods, properties and events
- Samples

Overview

Organogram is an organizational chart component that can be used for creating hierarchical diagrams representing relations between people in an organization or showing different elements of some complex structure.

It inherits API from TreeStore and DataStore. The component's appearance can be easily customized. The component items can include various HTML-elements, e.g. photos, and can be styled according to the user's needs.



Creating Organogram

- Initializing Organogram on a page
- Loading Data
- Selecting Items

Teaches you how to create Organogram on the page, adjust it and load with data.

Look-and-feel

- Customizing Organogram
- Styling

Covers techniques for fine-tuning the Organogram appearance: applying templates, customizing chart items, styling the component.

Others

- Event Handling
- General How-Tos

Discusses event handling with Organogram and the most obvious points of the component's customization.

Available only in **PRO Edition**

Initializing Organogram on a page

A new HTML file and related code files

To begin with, create a new HTML file and include the related Webix code files in it.

Required code files are:

- webix.css
- webix.js

A basic html page with the included code files

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Organogram</title>
    <script src="../codebase/webix.js" type="text/javascript"
charset="utf-8"></script>
    <link rel="stylesheet" href="../codebase/webix.css" type="text/css"
charset="utf-8">
</head>
<body>
    <script>
        //here you will place your JavaScript code
</script>
</body>
</html>
```

Placing Organogram on a page

Next, you should create a div container to place your organogram into.

```
<div id="box" style="width:900px;height:450px;"></div>
```

Object constructor

Now you can initialize the component. To create a new Organogram object, you need to use the next constructor:

```
Organogram constructor
```

```
orgChart = new webix.ui({view:"organogram"});
```

If you want to place the component into the container, you should specify the container property inside of the view constructor:

```
Basic Organogram configuration

orgChart = new webix.ui({
    container:"box",
    view:"organogram"
});
```

RELATED SAMPLE: Organization Chart

Available only in **PRO Edition**

Loading Data

Webix Organogram can get data in the following predefined data formats:

- JSON (default);
- XML.

Details and examples of data formats in tree-like structures

JSON

```
[
    { id:"1", value:"Manager", data:[
        { id:"1.1", value:"Store Manager" },
        { id:"1.2", value:"Office Assistant" , data:[
            { id:"1.2.1", value:"Dispatcher", data:[
                { id:"1.2.1.1", value:"Drivers" }
            ] }
        ] }
     ]}
]
```

XML

```
<data>
    <item id="1" value='Manager'>
        <item id="1.1" value='Store Manager'></item>
        <item id="1.2" value='Office Assistant'>
            <item id="1.2.1" value='Dispatcher'>
                <item id="1.2.1.1" value='Drivers'></item>
            </item>
        </item>
    </item>
</data>
```

There are two main ways you can specify the data source for Organogram:

- Define the data set in the object initialization;
- Use parse or load methods.

Both methods lead to the same result.

If you load XML data you need to set the datatype property in the constructor config or the

second parameter of the parse / load method to the name of the expected data type, i.e. 'xml'.

Loading from inline dataset

If you want to specify the data set directly on the page, you can use:

property data

method parse

```
Specifying data inline

chart_data = [
    {id:"root", value:"Board of Directors",  data:[
        { id:"1", value:"Managing Director", data:[
            {id:"1.1", value:"Base Manager", data:[
                { id:"1.1.1", value:"Store Manager" },
                { id:"1.1.2", value:"Office Assistant", data:[
                    { id:"1.1.2.1", value:"Dispatcher", data:[
                        { id:"1.1.2.1.1", value:"Drivers" }
                    ] }

                ] },
                { id:"1.1.3", value:"Security" }
            ]},
            { id:"1.2", value:"Business Development Manager", data:[
                { id:"1.2.1", value:"Marketing Executive" }
            ]},

            ...
        ]}
    ]}
];

var orgChart = new webix.ui({
  view:"organogram",
  ...
  data: chart_data
});

//or
orgChart.parse(chart_data)
```

RELATED SAMPLE: Organization Chart

Loading from a data file

To load data from a file on the server, you can use:

- property url
- method load

Remember, in case of XML data you should specify the data type explicitly as 'xml':

```
orgChart = new webix.ui({
  view: "organogram",
  ...
  url: "data/data.xml",
  datatype: "xml"
});
//or
orgChart.load('data/data.xml', 'xml');
```

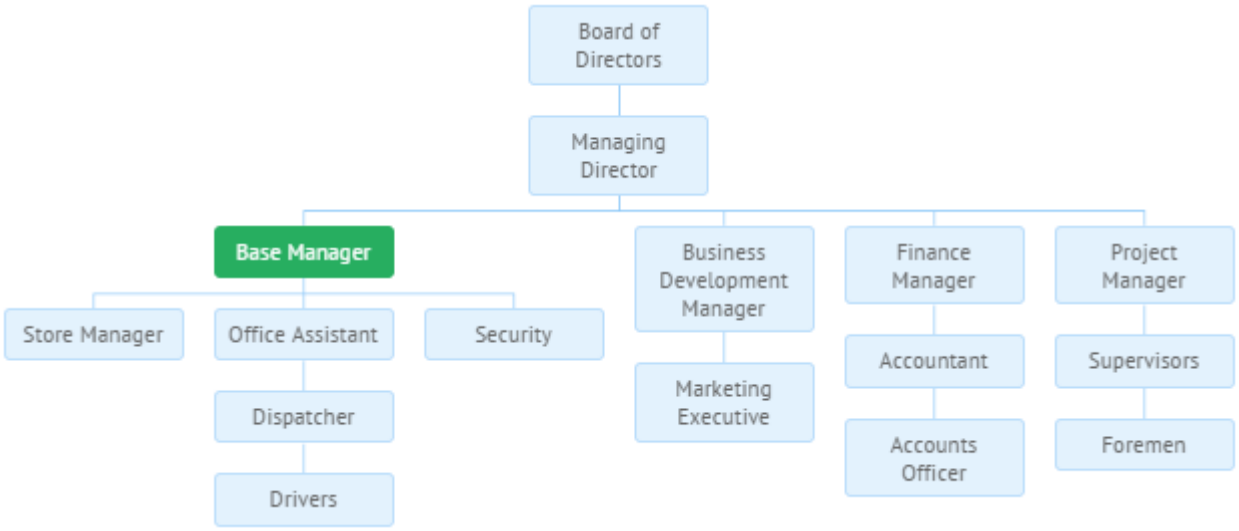
Loading from a database

As the Organogram component is based on the TreeStore, you can also load data from a database on the server. For the details, read the article Loading data.

Available only in **PRO Edition**

Selecting Items

You can select items within the Organogram component.



By default, selection is disabled in Organogram. To activate it you should set the select parameter to *true*.

```
Enabling selection in Organogram

orgChart = new webix.ui({
  view: "organogram",
  ...
  select: true
})
```

RELATED SAMPLE: Organization Chart

Related methods

getSelectedId	method	returns the selected cells ids
select	method	selects the specified item
selectAll	method	selects all items
isSelected	method	checks whether the specified item is selected or not
unselect	method	cancels selection of the specified item
unselectAll	method	removes selection from all items

Related events

onAfterSelect	event	fires after a cell is selected
onBeforeSelect	event	fires before a cell is selected
onSelectChange	event	fires when selection is changed in DataTable

Available only in **PRO Edition**

Customizing Organogram

The dimensions of items in Organogram, their arrangement and content, as well as the position of the component in relation to the container can be customized with the help of the **item type**.

Common configuration

The following properties can be set in the type to customize the whole component:

- **autowidth** - boolean, adjusts the container to fit the chart width

```
webix.ui({
    view: "organogram",
    autowidth: true,
    ...
});
```

- **autoheight** - boolean, adjusts the container to fit the chart height

```
webix.ui({
    view: "organogram",
    autoheight: true,
    ...
});
```

- **lineColor** - color of the line that connects items in organogram

```
webix.ui({
    view: "organogram",
    lineColor: "#90caf9",
    ...
});
```

Items configuration

The appearance of Organogram items can also be changed by setting the following properties in the **type** parameter:

- **width** - the width of an item in pixels, must be a fixed number

```
webix.ui({
    view: "organogram",
```

```
    type:{
         width: 120
    },
    ...
});
```

- **height** - the height of an item, "auto" by default, fits to the content's height

```
webix.ui({
    view: "organogram",
    type:{
       height: 100
    },
    ...
});
```

- **padding** - the space between the organogram and the container borders

```
webix.ui({
    view: "organogram",
    type:{
        padding: 20
    },
    ...
});
```

- **marginX** - horizontal space between two items

```
webix.ui({
    view: "organogram",
    type:{
        marginX: 20
    },
    ...
});
```

- **marginY** - vertical space between two items

```
webix.ui({
    view: "organogram",
    type:{
        marginY: 20
    },
    ...
});
```

- **template** - defines inner html for each item, "#value#" by default

```
```

```
{
    template: webix.template("#value#")
}
```

If you want to set any other html content for some item you should redefine the **template** property. For example, let's set images for chart items:

```
var orgChart = new webix.ui({
    container:"testA",
    view:"organogram",
    template: function(obj){
        var image = obj.img;

        if(image){
            html = "<img src='"+image+"' class='photo'>";
        }
        return (image||"")+obj.value;
    }
});
```

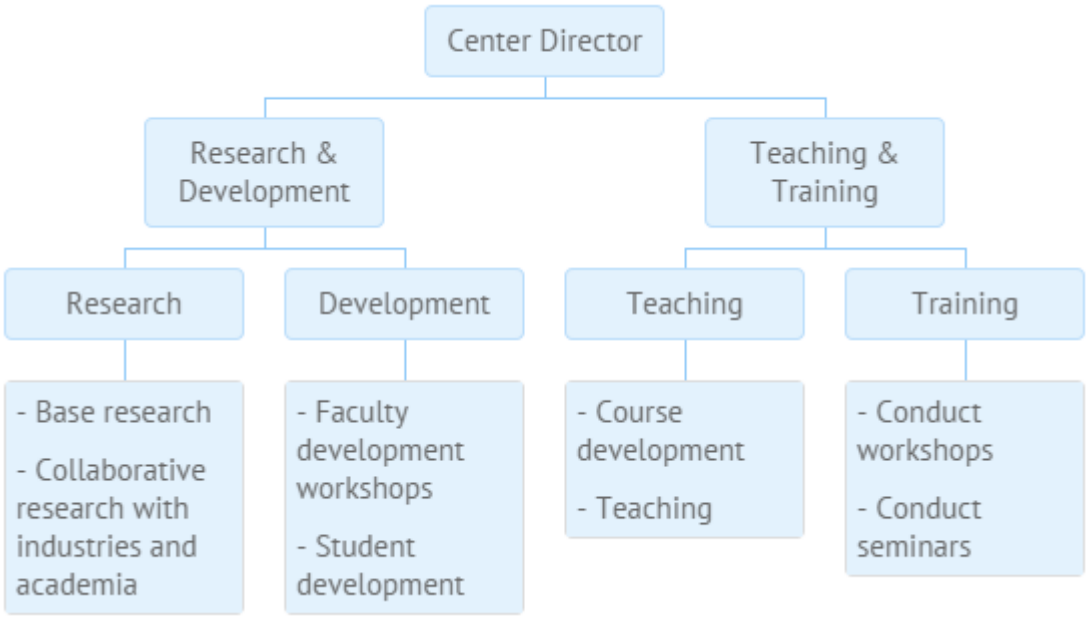
You can also apply a custom template to specify elements of the list. Let's mark the list items by the mark ">". To parse this sign correctly, we will set it as a **">"** html character code.

```
var orgChart = new webix.ui({
    container:"testA",
    view:"organogram",
    template: function(marks){
        var html = "";
        if(marks && marks.list_item){
            html = " &gt; ";
        }
            return html;
    }
});
```

RELATED SAMPLE: Templates

Creating list blocks

There's a possibility to make a list block inside of an item.



To create a list block, you need to specify the **$type** property with the value "list" in the parent of the items that you want to turn into a list.

Let's have a look at the organogram in the above picture. The items of the 4th level are united into lists.

For example, to unite the child items of the "Research" item ("Base research" and "Collaborative research with industries"), we should set the **$type** property with the value "list" in its definition.

The same actions should be performed with the "Development", "Teaching" and "Traning" items and their child items.

The code will look like this:

```
var orgChart = new webix.ui({
    container:"testA",
    view:"organogram",
    data: [
        {id:"1", value:"Center Director", data:[
            { id:"1.1", value:"Research &amp;amp; Development", data:[
                { id:"1.1.1", value:"Research", $type: "list", data:[
                    { id:"1.1.1.1", value:"Base research" },
                    { id:"1.1.1.2", value:"Collaborative research with
industries" }
                ]},
                { id:"1.1.2", value:"Development", $type: "list", data:[
                    { id:"1.1.2.1", value:"Faculty development workshops"
},
                    { id:"1.1.2.2", value:"Student development" }
                ]}
            ]},
            ...
        ]}
    ]
```

```
});
```

RELATED SAMPLE: List blocks

Available only in **PRO Edition**

Styling

You can customize any element of Webix Organogram. All you need to do is to redefine the needed attributes of the related css class in the <style> block of your page. The untouched attributes will take default values.

```
<style>

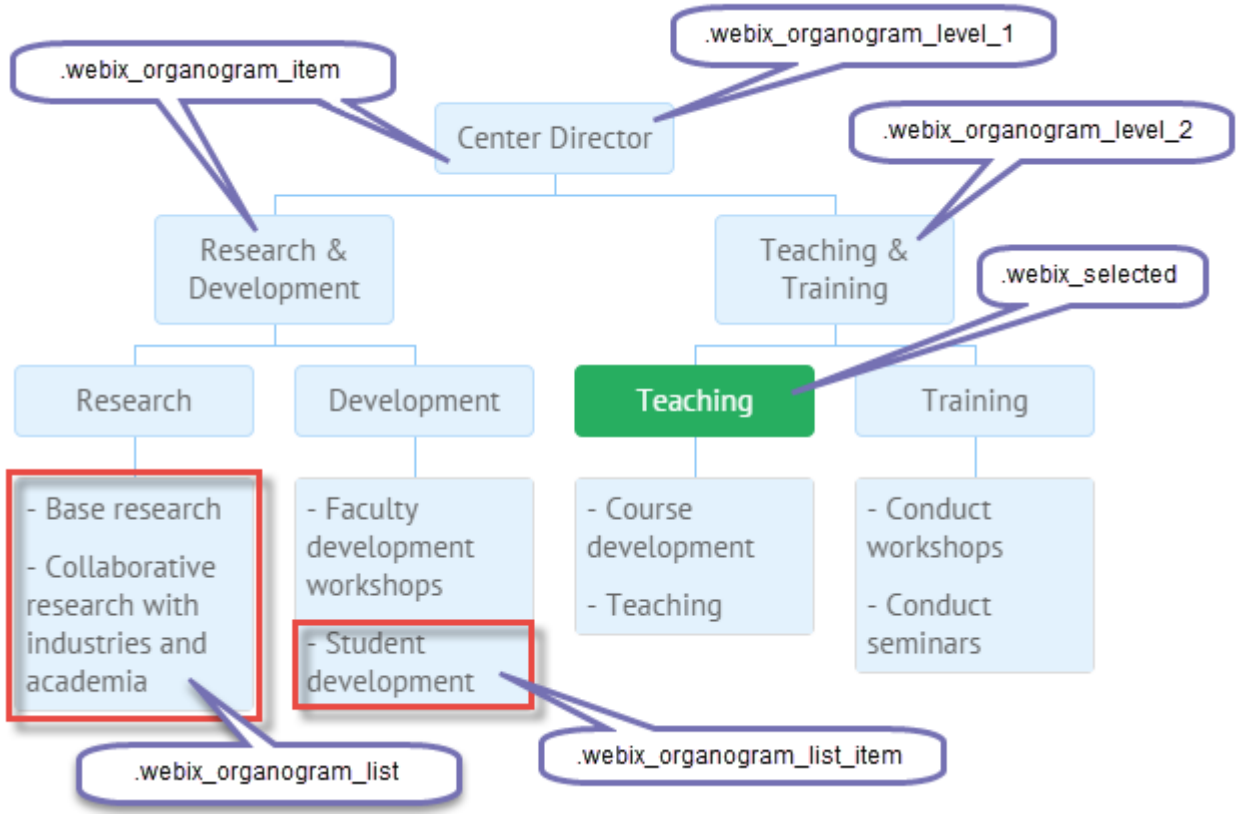
    .webix_organogram{
        background-color: #2196f3;
    }

</style>
```

In the above example we redefined only the background color. All the other attributes defined in the '**.webix_organogram**' class took the default values.

CSS classes and the related elements

- **.webix_organogram** - css rule for container of the organizational chart
- **.webix_organogram_item** - css rule for all chart items
- **.webix_organogram_level_1** - css rule for the items of the 1st level
- **.webix_organogram_level_2** - css rule of the items of the 2nd level
- ...
- **.webix_organogram_level_[n]** - css rule for the items of the n level
- **.webix_organogram_list** - css rule for list block
- **.webix_organogram_list_item** - css rule for items inside list block
- **.webix_selected** - css rule for the items in selected state



Custom styling

You can also redefine the default styles or apply your own css rules for chart items.

Use the **$css** property in item data to specify item style in one of the two ways:

- create new css rule in <style> block and apply it to the necessary items
- specify css style directly in the item definition

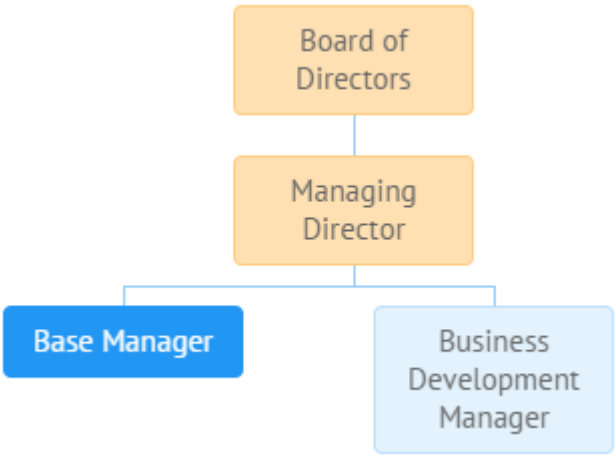
To illustrate the described techniques, we will define the **.item_top** css rule and apply it to the item of the 1st level. For the item of the 2nd level we'll set specific css properties directly in the data set.

Finally, we'll redefine the **.webix_selected** rule to change background color of selected items.

```
<style>
    // "top" ccs rule definition
    .item_top{
        background-color: #ffe0b2;
        border-color: #ffcc80;
    }
    // the style for selected item
    .webix_selected{
        background-color: #2196f3 !important;
        border-color: #2196f3 !important;
    }
</style>
```

```
<script>
orgChart = new webix.ui({
    container:"box",
    view:"organogram",
    data:[
        {id:"root", value:"Board of Directors", $css: "item_top",  data:[
            {id:"1", value:"Managing Director",
              $css:{background: "#ffe0b2", "border-color": "#ffcc80"},
data:[
                {id:"1.1", value:"Base Manager"},
                { id:"1.2", value:"Business Development Manager"}
                ...
            ]}
        ]}
    ]
});
</script>
```

You can see the result in the picture below:



RELATED SAMPLE: Styling

Available only in **PRO Edition**

Event Handling

Organogram supports various events that can be used to provide a custom behaviour for your organogram.

There are 2 ways you can add a handler to the event:

- through the method attachEvent();
- through the parameter on.

Event names are case-insensitive

Method attachEvent()

You can attach several handlers to the same event and detach them using two respective methods:

```
A general way to attach/detach the event handler

// to attach event
var myEvent = $$("orgChartId").attachEvent("onItemClick", function () {
// event handler code
})

//to detach event
$$("orgChartId").detachEvent(myEvent);
```

Parameter 'on'

With the help of parameter on you can also attach any event(s) to an Organogram object. But in contrast to using the attachEvent method you can't detach the attached events later.

```
Attaching the event handler through parameter 'on'

webix.ui({
    view:"organogram",
    ...
    on: {
        onitemClick: function (id) {
            alert("item has just been clicked");
       }
    }
);
```

Cancelable Events

All events with the subword '**onbefore**' can be cancelled.

To cancel some event you should return **false** within the appropriate event handler.

```
Cancelling the event handler

var myEvent = $$("orgChartId").attachEvent("onBeforeSelect", function (id)
{
    ... // some event handler code
    return false;
})
```

Accessible objects and data

Inside the event handler you can refer to the holder component through the keyword **this**. Besides, most event handlers get incoming argument(s), like the **id** of a data item (see organogram events to know exactly what arguments are passed inside event handler).

Btw, using the **id** of a data item you can access this item itself and all its properties. For example:

```
Referring within the event handler

$$("orgChartId").attachEvent("onAfterSelect",function(id){
  var level = this.getItem(id).$level;
})
```

RELATED SAMPLE: Events

Available only in **PRO Edition**

General How-Tos

How to create list blocks?

To create a list block, you need to specify the **$type** property with the value "list" in the parent of the items that you want to turn into a list.

In the example below, we will unite the "Base research" and "Collaborative research with industries" items into a list. For this purpose, we will set the **$type** property with the value "list" in their parent item called "Research".

The same actions will be performed with the "Development" value and its child items.

The code will look like this:

```
var orgChart = new webix.ui({
    container:"testA",
    view:"organogram",
    data: [
        {id:"1", value:"Center Director", data:[
            { id:"1.1", value:"Research &amp;amp; Development", data:[
                { id:"1.1.1", value:"Research", $type: "list", data:[
                    { id:"1.1.1.1", value:"Base research" },
                    { id:"1.1.1.2", value:"Collaborative research with
industries" }
                ]},
                { id:"1.1.2", value:"Development", $type: "list", data:[
                    { id:"1.1.2.1", value:"Faculty development workshops"
},
                    { id:"1.1.2.2", value:"Student development" }
                ]}
            ]},
            ...
        ]}
    ]
});
```

RELATED SAMPLE: List blocks

How to add an image into an item?

To add an image for an item, you should redefine the **template** property:

```
var orgChart = new webix.ui({
    container:"testA",
    view:"organogram",
    template: function(obj){
        var image = obj.img;

        if(image){
            html = "<img src='"+image+"' class='photo'>";
        }

        return (image||"")+obj.value;
    }
});
```

RELATED SAMPLE: Templates

How to specify a custom style for items?

You can also redefine the default styles or apply your own css rules for chart items.

Use the **$css** property in item data to specify item style in one of the two ways:

- create new css rule in <style> block and apply it to the necessary items
- specify css style directly in the item definition

To illustrate the described techniques, we will define the **.item_top** css rule and apply it to the item of the 1st level. For the item of the 2nd level we'll set specific css properties directly in the data set.

Finally, we'll redefine the **.webix_selected** rule to change background color of selected items.

```
<style>
    // "top" ccs rule definition
    .item_top{
        background-color: #ffe0b2;
        border-color: #ffcc80;
    }
    // the style for selected item
    .webix_selected{
        background-color: #2196f3 !important;
        border-color: #2196f3 !important;
    }
</style>

<script>
orgChart = new webix.ui({
    container:"box",
    view:"organogram",
    data:[
        {id:"root", value:"Board of Directors", $css: "item_top",  data:[
```

```
            {id:"1", value:"Managing Director",
              $css:{background: "#ffe0b2", "border-color": "#ffcc80"},
data:[
                {id:"1.1", value:"Base Manager"},
                { id:"1.2", value:"Business Development Manager"}
                ...
            ]}
        ]}
    ]
});
</script>
```

RELATED SAMPLE: Styling

Popup

API Reference

- Methods, properties and events
- Samples

Overview

Popup is a variant of a ui-related window. Unlike it, the popup disappears as you click somewhere outside it. Popup is be used to alert some info as well can form a popup menu.



Initialization

Popup consists of **body** (both taking either view or text template) without the **head** section.

The component should be made visible with the **show()** method you call during the component initialization (if, of course, you want the popup to appear on page loading).

```
webix.ui({
    view:"popup",
    id:"my_popup",
    head:"My Window",
    body:{
        template:"Some text"
    }
}).show();
```

RELATED SAMPLE: Popup Window

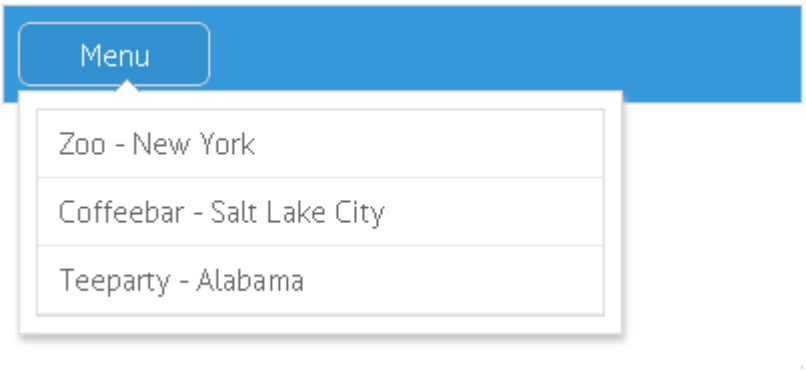
Popup Sizing and Positioning

For adequate work, you need to specify the dimensions of your popup as well as its offsets (**top** and **left**). Otherwise, the component occupies the whole screen or sticks to its top left corner respectively.

```
webix.ui({
    view:"popup",
    width: 400, height:300,
    top:200, left: 300
}).show();
```

There exist more **positioning options**. All of them are described in a separate documentation article.

Creating a Popup Menu



RELATED SAMPLE: Popup Menu

To create a popup menu that will appear on button click, you should create a button with **popup** property that takes popup **ID** as value.

The popup is hidden initially. Toolbar and popup are inited within different **webix.ui** containers.

```
Popup Menu

webix.ui({
       view:"toolbar",
       elements:[
            {view:"button", label: 'Menu', popup:"my_pop"},
            ...
        ]
});

webix.ui({
       view:"popup",
       id:"my_pop",
       body:{
            view:"list",
        ...}
```

```
}).hide();
```

RELATED SAMPLE: Popup Menu

Related Articles

- Window
- Window and Popup Positioning
- Control Guide
- Toolbar
- Changing Visibility of Components
- Popup CSS Image Map

Available only in **PRO Edition**

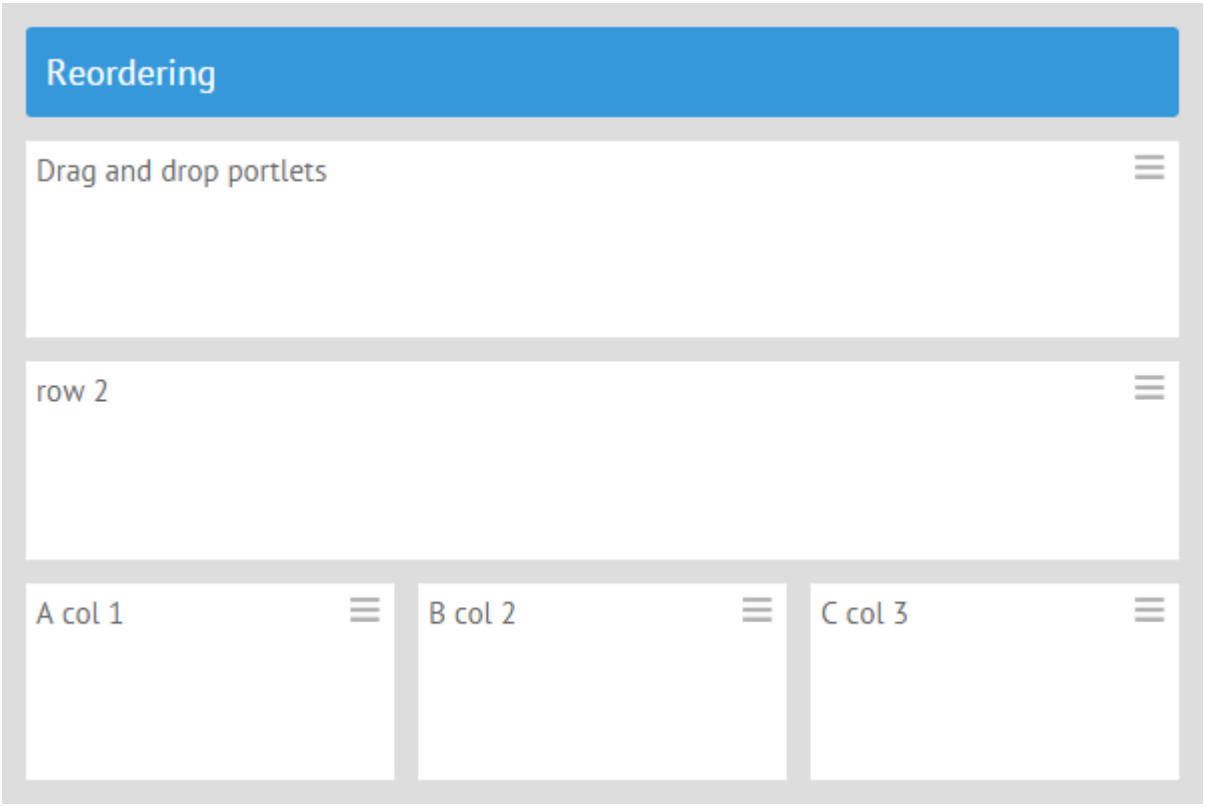
Portlet

API Reference

- Methods, properties and events
- Samples

Overview

Portlet is a layout component that allows placing other widgets inside of it. You can easily drag this or that portlet anywhere you want on the page. Thus, it's possible to change the app's structure in a simple way, without writing any extra code.



Initialization

As any other component, Portlet can be initialized in a container or as a full-screen view.

To initialize a portlet in a div container, you should create a container first:

```
<div id="divA" style="width:600px;height:400px;"></div>
```

Then, in order to place the component into the container, specify the container property inside of the view constructor, as it's shown in the example below.

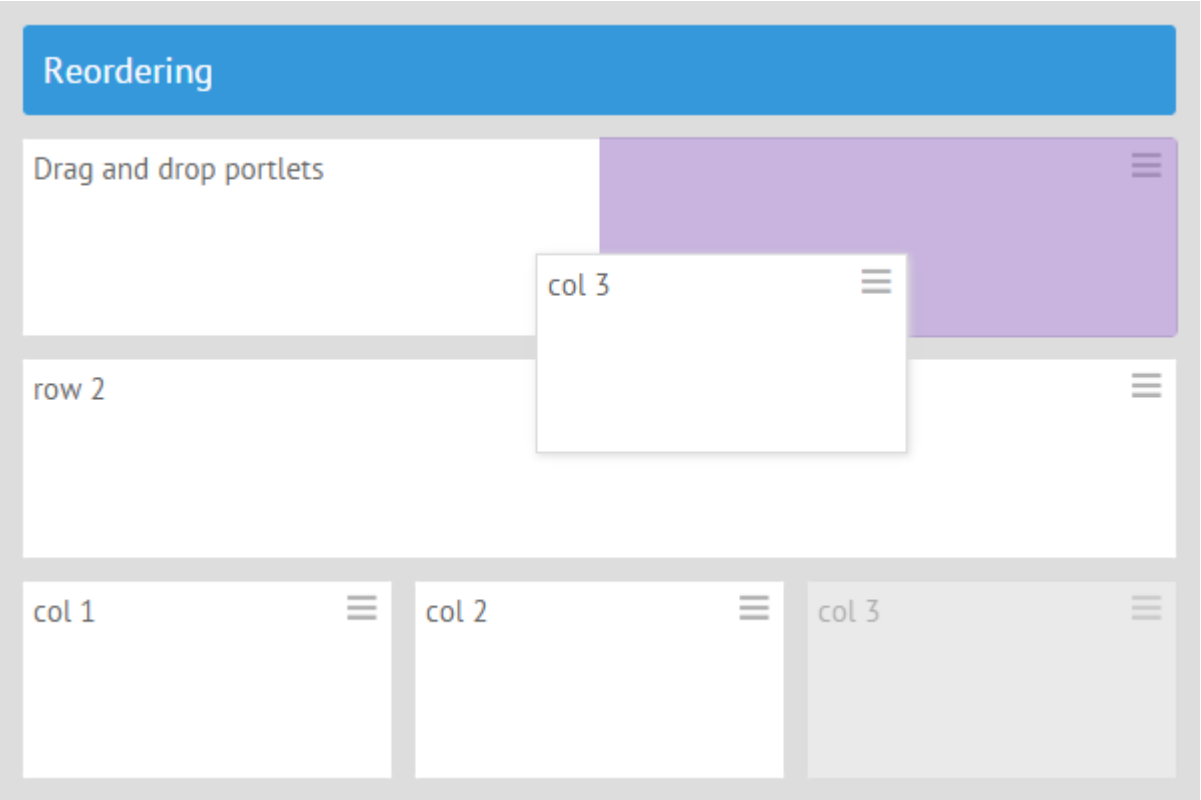
Portlet consists of a body (taking either a view or a text template) without the head section. A header can also be added. You'll find the details below.

```
webix.ui({
    container:"divA",
    type:"space",
    view:"portlet", body:{
        template:"Drag and drop portlets"
    }
});
```

RELATED SAMPLE: Portlet: Move cells

Moving portlets

You can move portlets by the handler. While dragging a portlet, possible positions for dropping are defined by highlighting. A portlet can be placed as a column or as a row.

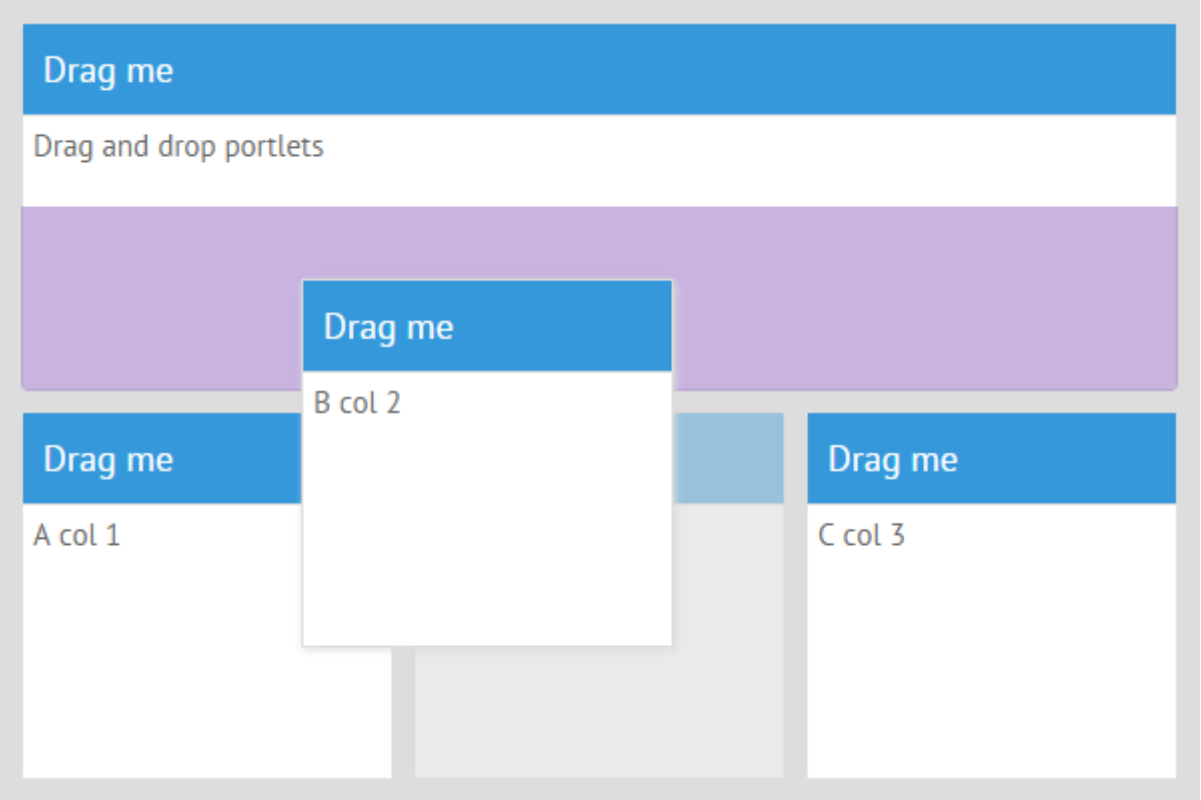


RELATED SAMPLE: Portlet: Move cells

Dragging by the header

In order to drag a portlet you can use not only the handler, but also the portlet header. Just set some string value for the *header* property in the porlet definition:

```
webix.ui({
    type:"space",
    view:"portlet", header:"Drag me", body:{
        template:"Drag and drop portlets"
    }}
});
```



RELATED SAMPLE: Portlet: Drag Header

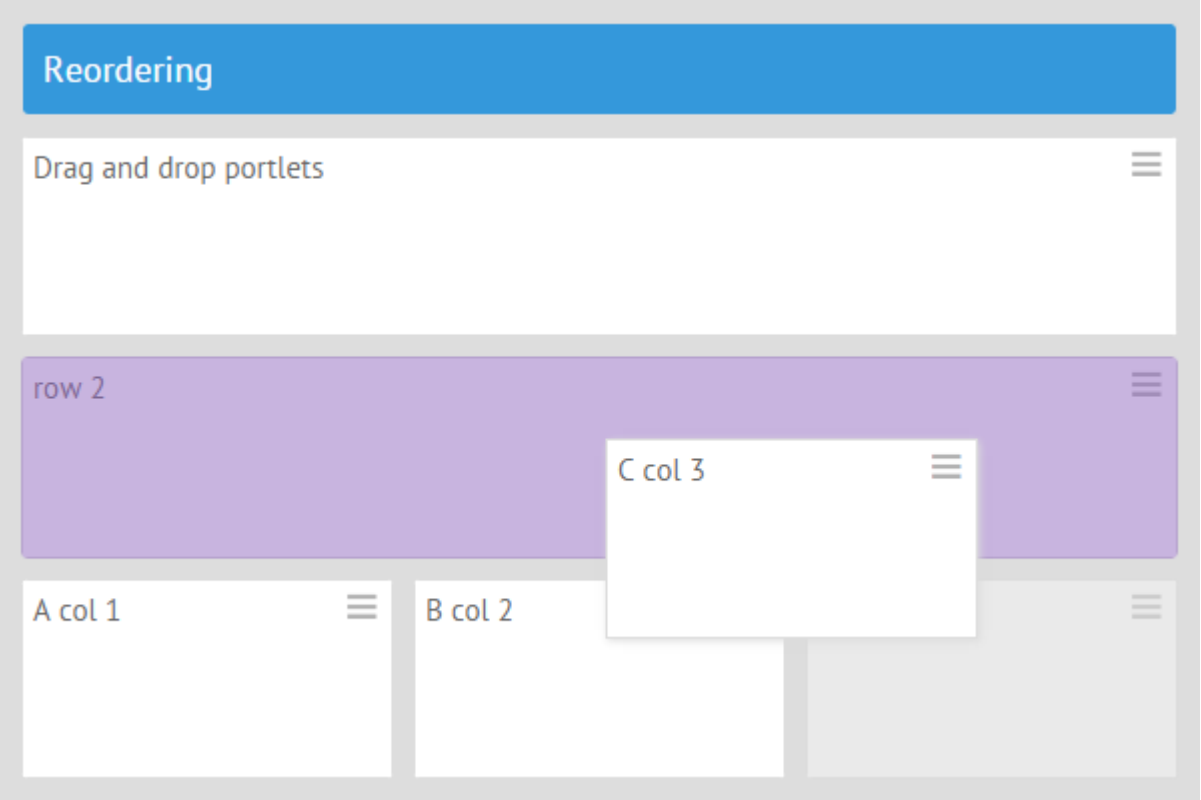
Modes of moving portlets

Replacing portlets

Instead of simply moving a portlet to a new position, it's also possible to replace the target portlet with the moved one. This mode is enabled by specifying the *mode* property with the value "replace":

```
webix.ui({
    type:"space",
    view:"portlet", mode:"replace", body:{
        template:"Drag and drop portlets"
    }}
```

```
});
```

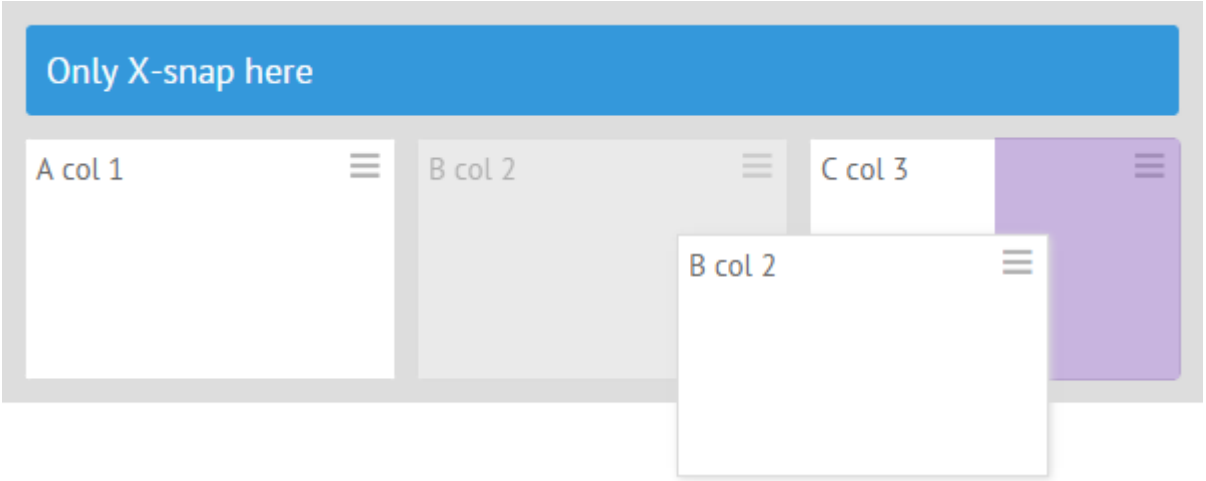


RELATED SAMPLE: Portlet: Replace cells

One direction moving

There are two modes that allow portlets move either only in the horizontal direction or in the vertical one. They are called "cols" and "rows", respectively. In the "cols" mode portlets are moved only along the x-axis. In the "rows" mode portlets can only be placed along the y-axis.

```
webix.ui({
    type:"space",
    rows:[
        { template:"Only X-snap here", type:"header"},
        { type:"wide", cols:[
            { view:"portlet", mode:"cols", body:{
                template:"A col 1"
            }},
            { view:"portlet", mode:"cols", body:{
                template:"B col 2",
            }},
            { view:"portlet", mode:"cols", body:{
                template:"C col 3"
            }},
        ]
        }
    ]
});
```

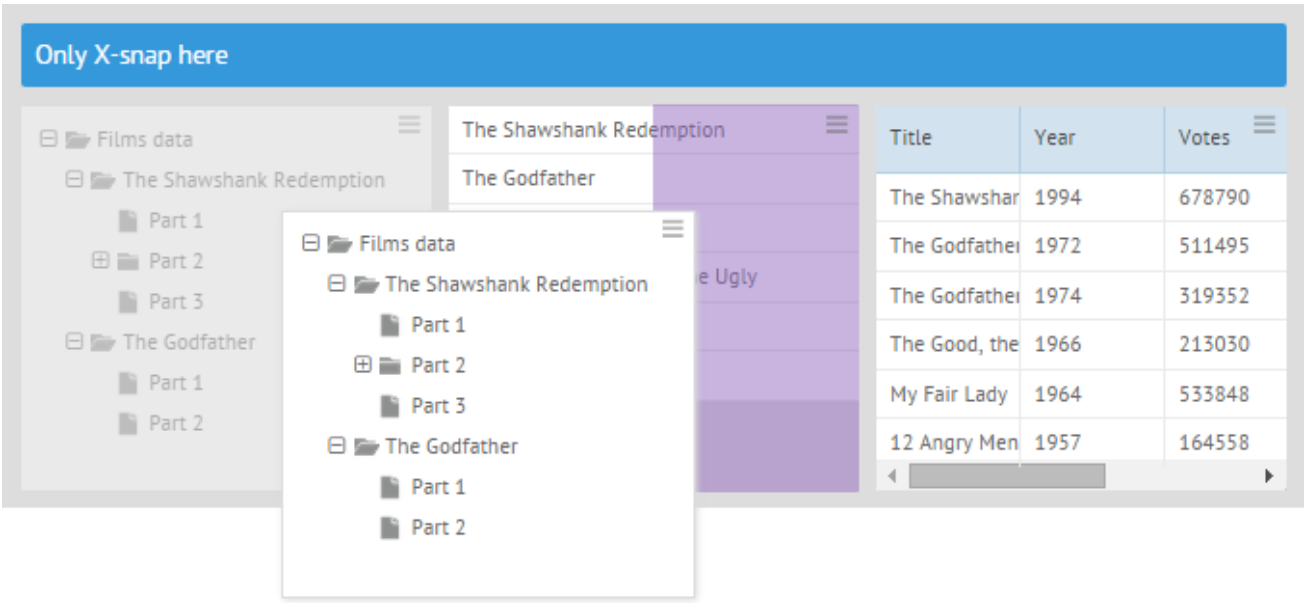


RELATED SAMPLE: Portlet: One direction

Adding other components

It's very easy to add other component into a portlet. All you need to do is to specify the view name, data source and any other necessary properties in the portlet's body:

```
webix.ui({
    type:"space",
    rows:[
        { template:"Only X-snap here", type:"header"},
        { type:"wide", cols:[
            { view:"portlet", mode:"cols", body:{
                view:"tree", data:smalltreedata
            }},
            { view:"portlet", mode:"cols", body:{
                view:"list", data:small_film_values_set, select:true,
autoheight:true
            }},
            { view:"portlet", mode:"cols", body:{
                view:"datatable", data:small_film_set, autoConfig:true,
autoheight:true
            }},
        ]
        }
    ]
});
```



RELATED SAMPLE: Portlet: Drag Components

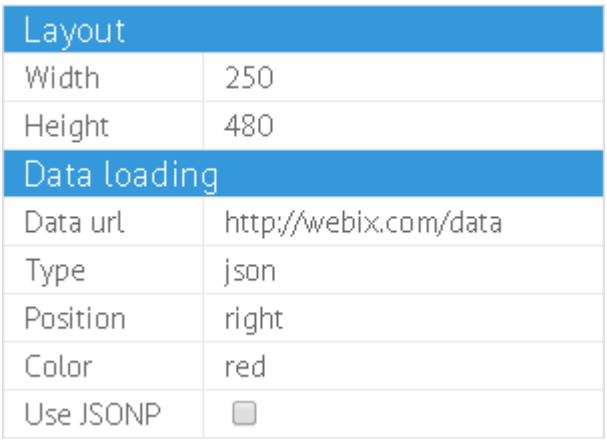
Property Sheet

API Reference

- Methods, properties and events
- Samples

Overview

Property sheet is a list of properties. Ui-related property sheet inherits from view.



Initialization

```
{ view:"property",  id:"sets", width:300,
    elements:[
       { label:"Layout", type:"label"},
       { label:"Data url", type:"text", id:"url"},
       { label:"Data type", type:"select", options:["json","xml","csv"],
id:"type"},
       { label:"Use JSONP", type:"checkbox", id:"jsonp"}
    ]
};
```

RELATED SAMPLE: Property Sheet: Basic Initialization

The elements with **label** gets header CSS. If there's no type the element is non-styled and non-editable.

Working with Property Sheet

Property Sheet Values

The elements of a property sheet can be treated together, you can get and set their values just like you do it with form and HTMLform.

- setValues();
- getValues();
- HTMLform and Form Treatment

```
$$("sets").setValues({
    width:250,
    height:480,
    url:"http://Webix.com/data",
    type:"json"
});

var values = $$("sets").getValues();
```

Editing Data in Property Sheet Elements

Each element features a built-in editor specified by **type** property. Possible values coincide with editor types and include:

- **color** - an editor for colors, colorboard appears on editing;
  - RELATED SAMPLE: Property Sheet: Colorpicker
- **text** - a standard editor for text values;
- **password** - a text editor for passwords;
- **select** - for selection among several text values with the help of an HTML 'select' control;
- **combo** - for selection among several values with the help of Webix combo control. Input is editable;
- **richselect** - for selection among several values with the help of Webix richselect control. Input is non-editable;
- **checkbox** - for two-state values.

Select editors features **options** in their configuration that can be defined in the following ways:

```
elements:[
    { type:"select", options: ["json","xml","csv"]},
    { type:"select", options: [
        {id:1, value:"json"},
        {id:2, value:"xml"},
        {id:3, value:"csv"}
    ]}
]
```

Customizing Editor Collection

Besides, other editors types can be registered for the elements with the help of the **registerType()** method. During registering the new element type, you should specify its **appearance** and **behavior**.

```
'Toggle' editor for two-state values

$$("sets").registerType("toggle",{
    template:function(value, config){
        return "<input type='button' class='webix_toggle_button_custom'
value='"+
          value+"' style='margin:0px; width:95%; height:20px; font-
size:12px; '>";
    },
    click:{
        webix_toggle_button_custom:function(e, id){
            var data = this.getItem(id);
            if (data.options[] == data.value)
                data.value = data.options[1];
            else
                data.value = data.options[];

            this.editStop();
            this.refresh(id);
            this.callEvent("onCheck",[id, data.value]);
        }
    },
    editor:"inline-text"
});
```

And then use it:

```
webix.ui({
    view:"property", id:"sets", elements:[
        { label:"Data type", type:"toggle", options:["json","xml"],
id:"type"}
    ]
});
```

RELATED SAMPLE: Property Sheet: Custom Editors

Using Custom Popup Editors

Popup editors are based on Webix popup window and can be registered with either default or custom logic:

```
Default Popup

//including the editor into the property sheet
webix.ui({
    view:"property", id:"sets", elements:[
        { label:"Color front",  type:"popup", popup:"myColorF",
id:"colorF" },
```

```
        ...
    ]
});

//defining its appearance
webix.ui({
    id:"myColorF",
    view:"popup",
    body:{ view:"textarea", width:300, height:100 }
});

//registering the new type
prop.registerType("popup",{
    editor:"popup"
});
```

Custom Popup

```
// standard popup editors is extended with new functions
webix.editors["popup-ra"] = webix.extend({
    focus:function(){ },
    getValue:function(node){
        return this.getInputNode(node).data.edit;
    },
    setValue:function(node, value){
        this.getPopup(node).show(node);

        this.getInputNode(node).data.edit = value;
        this.getInputNode(node).refresh();
    }
}, webix.editors["popup"]);

//new type is registered
prop.registerType("popup-ra",{
    editor:"popup-ra"
});
```

RELATED SAMPLE: Property Sheet: Custom Editors

Related Articles

- Data Object Operations
- Sizing Components
- Resizing
- Redefinition of the Components
- Property Sheet CSS Image Map

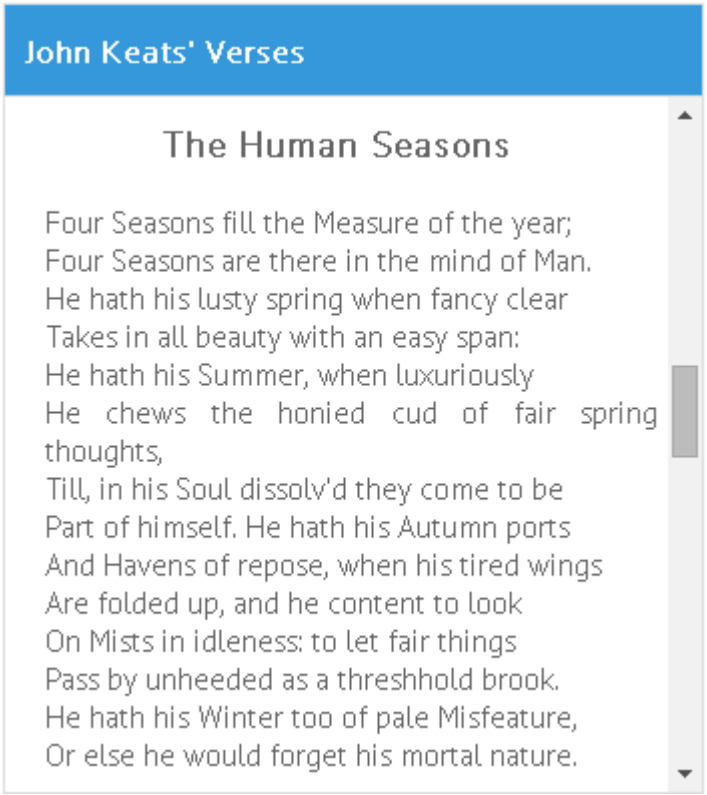
Scrollview

API Reference

- Methods, parameters and events
- Samples

Overview

Scrollview inherits from view and makes any of the nested components scrollable, either in horizontal or in vertical mode. It is handy in case of long homogenious data items, like text templates.



Initialization

It's reasonable to use the scrollview just as a sccrolling container while text into is stored in HTML.

```
<div id="mybox1">Some very long article-1</div>
<div id="mybox2">Some very long article-2</div>
<div id="mybox3">Some very long article-3</div>
...
```

Then you can put these 'long articles' into the scrollview and form a very long list where you can scroll to any

article you wish: HTML content is nested into UI component with the help of a dedicated data template:

```
webix.ui{(
        view:"scrollview",
        id:"verses",
        scroll:"y", //vertical scrolling
        body:{
            rows:[
                {id:"verse_1", template:"html->my_box1"}, // corresponds
to the dedicated div ID
                {id:"verse_2", template:"html->my_box2"}
                {...}
            ]}
    })
```

RELATED SAMPLE: Scrollview

Scrollview features a **body** object property where items (text templates, components) are stored in either **rows** or **cols** array for vertical or horizontal item arrangement respectively.

Working with Scrollview

Within scrollview as well as within any component with a scrollbar you can:

- scroll to a definite position of the component;
- define your position - how far you have scrolled away from the top and left border of the scrollview;
- enable and disable scrolling and specify scrolling directions.

Read more about **scroll control** in the corresponding article of the documentation.

Related Articles

- Setting Borders to the Components
- Adding Headers to Components
- Sizing Components
- Resizing
- Layout
- Multiview
- Carousel
- Accordion

Tabview

API Reference

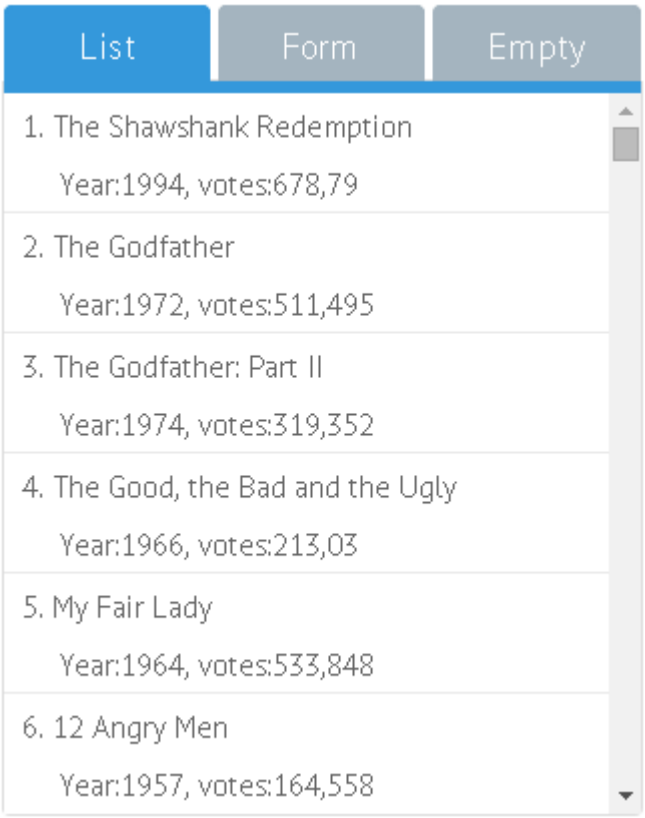
- Methods, properties and events
- Samples

Overview

Tabview is a **hybrid** component that is made of a multiview and tabbar.

Tabview allows for quick initialization of a multiview with a built-in ability to switch between the views.

It also features dedicated methods to add and remove views together with the corresponding tabs.



Initialization

There's no need to create a multiview and switching buttons for it. Everything lies within one and the same **"tabview"** component with **cells** property that contains collections of views.

```
Tabview
```

```
webix.ui({
  view: "tabview",
  cells: [
    {
      header: "List",
      body: {
        id: "listView",
        view: "list", ...
      }
    },
    {
      header: "Form",
      body: {
        id: "formView",
        view: "form", ...
      }
    }
  ]
});
```

RELATED SAMPLE: Tabview

Each tabview cell features:

- **header** - string with a heading for the cell, displayed as **tab title**;
- **body** - configuration object for the **component** that is placed in this cell.

Note that switching between views is enabled automatically via a tabbar control. For more customization, use a multiview component and choose the needed switching method.

Working with Tabview

The hybrid nature of a tabview allows configuring each of its parts separately:

- through the same-name object properties in the initial configuration:

These configuration objects may contain native properties of multiview and tabbar that need to be redefined. If the configurations are omitted, default values will be used.

```
webix.ui({
    view:"tabview",
    id:"my_tabview",
    cells: [],
    tabbar:{
        ...
```

```
    },
    multiview:{
        ...
    }
});
```

- by accessing multiview and tabbar objects to work with them dynamically:

```
var tabbarObj = $$("my_tabview").getTabbar();
var multiviewObj = $$("my_tabview").getMultiview();
```

All the API methods, events and properties of a ui.tabbar and ui.multiview respectively are applicable to these objects.

Enabling animation

By default the switching of tabview tabs is not animated. Since animation is a multiview feature, it can be enabled within **multiview** configuration.

```
view:"taview",
cells:[],
multiview:{
    animate:true
}
```

The **animate** property may take an object with advanced animation settings.

Setting icons for tabs



Icons for the tabbar tabs are set via additional HTML in the tab **header**:

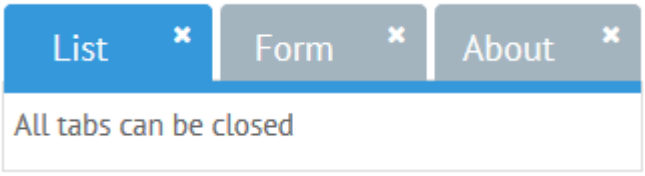
```
view: "tabview",
cells: [
  { header: "<span class='webix_icon fa-film'></span>List",
    body:{}
  }
]
```

RELATED SAMPLE: Multiview Tabbar with Icons

More about different types of defining icons in Webix you can learn in the Icons with UI Components article.

Defining tabs that can be closed

Closing functionality is a tabbar feature.



To make all tabs closable, **close** property should be set to *true* in the **tabbar** configuration:

```
view: "tabview",
tabbar: {
  close:true
}
```

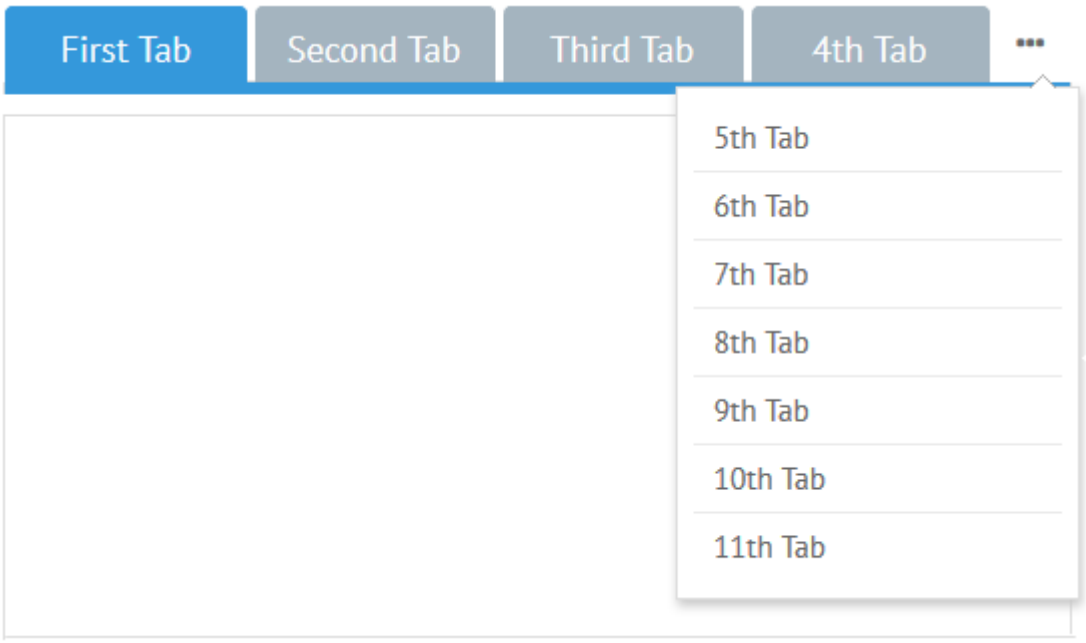
To make a separate tab closable, use the **close** proerty in its configuration:

```
view: "tabview",
cells: [
  { header:"List",
    close:true,
    body:{...}
  }
]
```

RELATED SAMPLE: Tabview: Close tab button

Responsive tabbar

Tabbar default feature. If the tabs are wider than the available space, some of them are pushed to a popup list and can be reached though the related menu.



Everything is done fully automatically, though customization options for tab and popup width, icons, etc. are available through the **tabbar** configuration:

```
view: "tabview",
tabbar: {
  popupWidth: 300,
  tabMinWidth: 120
}
```

More in detail in the dedicated documentation article.

Adjusting tab dimensions

The width of each tab can be set separately:

```
view: "tabview",
cells:[
  { header:"List", width:150, body:{...} },
  { header:"Form", width:250, body:{...} },
]
```

If tabs are of different width and height, they usually take the size of the smallest tab once the user switches to it.

To avoid this behavior, you should use the **fitBiggest** property in the **multiview** configuration object:

```
view: "tabview",
multiview: {
  fitBiggest:true
}
```

RELATED SAMPLE: Tabview: Fit Active vs Fit Biggest

Adding and removing cells dynamically

The functionality is described in detail in the Adding and Deleting Multiview Cells Dynamically article.

Related Articles

- Multiview
- Common Functions to Switch Between Views
- Controls to Switch Between View Cells
- Adding and Deleting Multiview Cells Dynamically

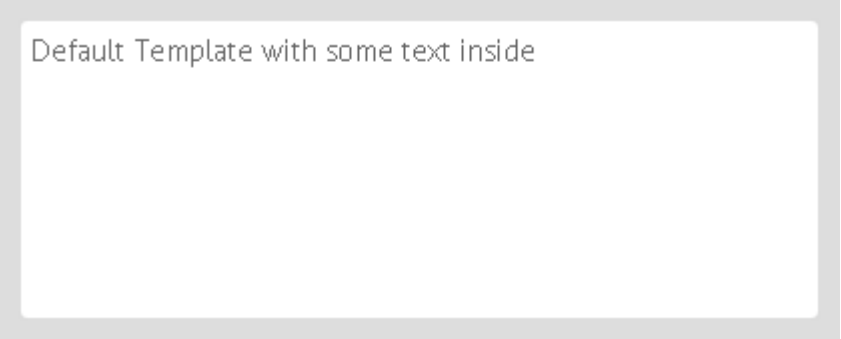
Template

API Reference

- Methods, properties and events
- Samples

Overview

Ui-related **template** looks like a non-editable area for rendering plain text or a single data record. Its content is set with the help of the same-name *template* parameter.



Initialization

Simple template

The **template** property of the component sets its content:

```
webix.ui({
  view: "template",
  template: "Default template with some text inside" // text content
});
```

RELATED SAMPLE: Default Template

External Template

Template's content can as well be taken from an HTML node pointed to by:

- the **content** property that links to a visible container:

```
<div id="myDiv">Default Template with some text inside</div>

webix.ui({
```

```
  view: "template",
  content: "myDiv"
});
```

RELATED SAMPLE: Default Template

- the **template** property that points to the container ID prefixed by "html->". The container should be invisible initially:

```
<div id="my_box1" style='display:none;'>Some text here</div>

{ template: "html->my_box1", autoheight: true },
```

RELATED SAMPLE: Autosizing to Content

As a result, contents from HTML is placed inside the view.

Loaded Template

```
{ view: "template", src: "loadtext.php" }
```

Notes

Specifying the `view` type as "template" is optional. The following code works as well.

```
webix.ui({
  container: "layout_div",
  rows: [
    { template: "Default Template with some text inside" },
    { ... }
  ]
})
```


The same-name property of the component, **template**, is also used by Webix data components. It allows setting pattern to display texts and render data items from multiple datasets alongside with adding styling to them. The Data Templates article covers all the rules of templating.

Template Content

The template content is set with the **template** property, which can be:

- **plain text**:

```
{ template: "Plain text" }
```

- **text** with **HTML** markup:

```
 {template: "Text with <b>HTML</b> markup" }
```

- **an individual data item** specified by the **data** property:

```
{ template: "#title#", data: { title: "Image One", src:
"imgs/image001.jpg" } }
// shows "Image One"
```

If a template is used as an individual data item (as shown above) the data fields should be interpolated in either of the two ways:

- using a string template containing **data keys** between **hash** signs to define which fields from the data record should be displayed;
- using a template **function** that takes the data record (`obj`) as parameter:

```
{
  data: {title: "Image One", src: "imgs/image001.jpg" },
  template: function (obj) {
    // obj is a data record object
    return '<img src="'+obj.src+'"/>'
  }
}
```

RELATED SAMPLE: Carousel

The rules of data templating can be studied in detail in the dedicated documentation article.

Changing Template Text Dynamically

Dymanic content setting can be implemented with the help of:

- a setHTML method to change text or HTML content of a template;
- a define method used in lots of the components to reset property values;
- a setValues method that allows to define template data;
- a template-specific setContent method that can set the contents of an existing HTML node as template value.

Template Types

Default

The default template is a white non-editable area with some text or HTML content. It's surrounded by a grey border.

```
{ view: "template", template: "Text" }
```

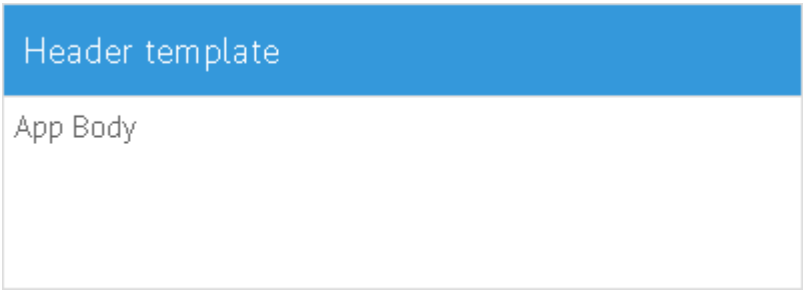
Clean

The `clean` template looks very much like the default one, but has no borders.

```
{ view: "template", template: "Text", type: "clean"}
```

Header

Header template is used to set headers for layout rows/columns. Looks like a blue bar above the main contents.

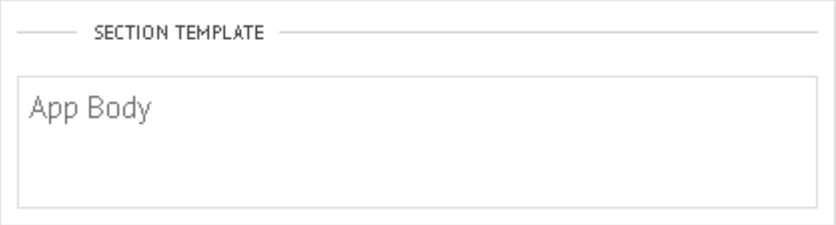



```
webix.ui({
  rows: [
    { view: "template", template: "Header Template", type: "header" },
    { view: "...", ... }
  ]
});
```


Note that headers of accordionitems are set in a different way.

Section

Section template places header within the line that encircles a row/column thus making it possible to neatly divide the webpage into blocks. As there's a line in this template type, it would be nice to resort to borderless layout design.




```
{view: "template", template: "Header template", type: "section" }
```

RELATED SAMPLE: Template Types

Toolbar

API Reference

- Methods, properties and events
- Samples

Overview

Ui-related toolbar inherits from layout. It is the place to nest all the possible controls including, buttons, labels, icons, checkboxes, combos, etc. For the full list of controls please refer to the library's Controls Guide.



Initialization

Toolbar controls are put into the **elements** array and are arranged into a horizontal line.

To distinguish between horizontal and vertical control arrangement, use **cols** and **rows** arrays respectively.

```
A three-button toolbar

webix.ui({
    view:"toolbar",
    id:"myToolbar",
    cols:[
        { view:"button", id:"LoadBut", value:"Load", width:100,
align:"left" },
        { view:"button", value:"Save", width:100, align:"center" },
        { view:"button", value:"Info", width:100, align:"right" }]
});
```

RELATED SAMPLE: Buttons: Alignment

Specify the ID of a toolbar as well as ID-s of all the necessary controls if you want to work with them later.

There exists a possibility to specify common configuration for all controls included into this or that toolbar.

Working with Toolbar

Image Buttons



RELATED SAMPLE: Buttons: Type 'image'

Differently-sized buttons can be included into the toolbar with the help of a custom function:

```
    webix.ui({
        container:"areaD",
        view:"toolbar",
        cols: buttons(48) /*here you specify the parameter you want to
                           pass to the function defined later*/
    });

   function buttons(size){
                return  [
                    { view:"button", type:"image"+size,
                        image:"../common/imgs/"+size+"/save.gif",
width:size+7},
                    { view:"button", type:"image"+size,
                        image:"../common/imgs/"+size+"/copy.gif",
width:size+7}
                ];
               }
```

More info about how to manipulate button groups.

Icon Buttons

Icons can be nested into all the library conponents. Within toolbar, icon buttons look definitely stylish.



RELATED SAMPLE: Buttons: Type 'icon'

Learn more about icon types in the dedicated chapter of the manual.

Related Articles

- Controls
- Tabview Functionality
- Toolbar CSS Image Map

Tree

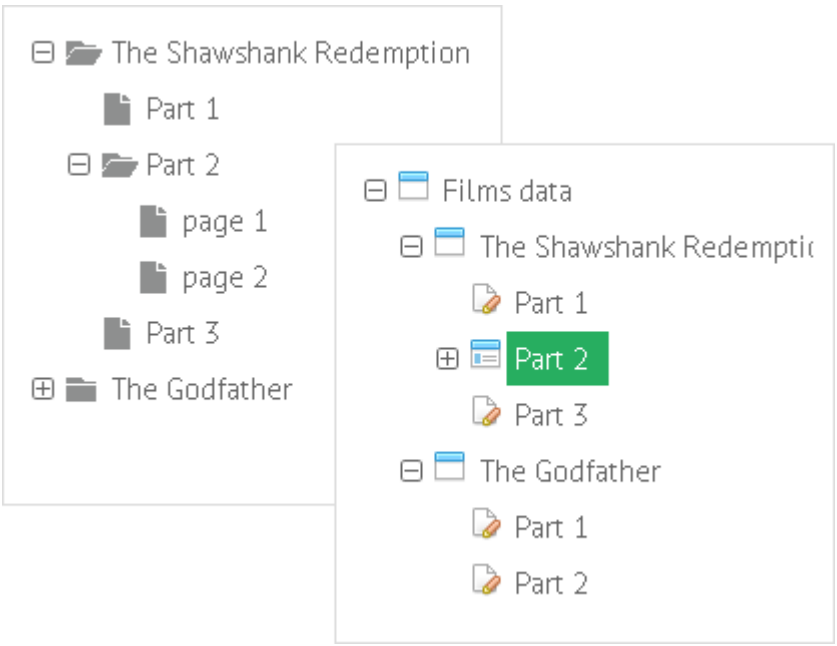
API Reference

- Methods, properties and events
- Samples

Overview

Tree is a pure JavaScript component for building various tree hierarchies.
It's a powerful and customizable tool with lots of features: support for XML, JSON data sources, static and dynamic loading, multitree drag-and-drop, inline editing, optional checkboxes, state saving, filtering, sorting, copying etc.

*Confused where to start? Read article '*Quick start with Tree*' to kick things off.*



Guides

Managing data

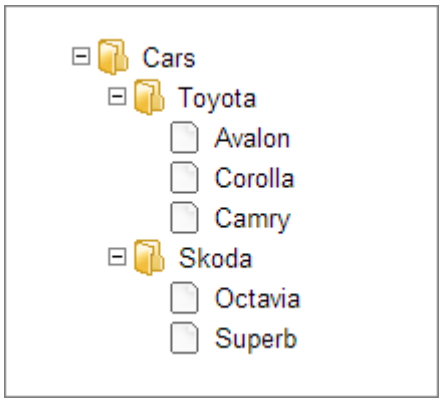
- Loading data
- Items Selection
- Editing in the Tree
- Filtering in the Tree
- Sorting in the Tree

Teaches you how to treat data in the tree to present the desired set of items.

Nodes manipulations ◦ Adding nodes ◦ Deleting nodes ◦ Expanding/collapsing branches ◦ Updating the tree ◦ Saving the tree state	Says about means you've got to manipulate with the tree branches and leaves: CRUD operations, tree state storage, etc.
Look-and-feel ◦ Node templates ◦ Adding Active Elements to Nodes ◦ Styling ◦ Checkboxes in Tree	Covers techniques for fine-tuning the tree appearance: use of data templates, adding checkboxes, icons, lines to items, redefining the default CSS classes, etc.
Others ◦ Copy-pasting data ◦ Drag-and-drop ◦ Events handling	Describes the remaining procedures related to the tree but not mentioned in the chapters above: work with the clipboard, support for internal/external dnd and handling events.

Quick start with Tree

This is a tutorial that will walk you through creating a basic application containing Tree. You will learn how to initialize Tree customize it and populate with data.



The final code of the tutorial can be used as the start point while creating applications using Tree.

RELATED SAMPLE: Quick start with DataTree

A new HTML file and related code files

To start, create a new HTML file and include the related Webix code files in it.

Required code files are:

- *webix.css*
- *webix.js*

A basic html page with the included code files

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Quick start with Tree</title>
    <script src="../codebase/webix.js" type="text/javascript"
charset="utf-8"></script>
    <link rel="stylesheet" href="../codebase/webix.css" type="text/css"
charset="utf-8">
</head>
<body>
    <script>
        //here you will place your JavaScript code
</script>
</body>
</html>
```

Placing Tree on a page

On the next step, you should create a *div* container where you then place your tree.

```
<div id="box" style="width:400px;height:200px;"></div>
```

Object constructor

After you have prepared the site for the tree you can move to its initialization.

To create a new Tree object you need to use the next constructor:

```
Tree constructor

tree = new webix.ui({ view:"tree" })
```

The only mandatory parameter you should specify inside the constructor:

- container - the HTML container for Tree. We created it on the previous step.

```
Basic Tree configuration

tree = new webix.ui({
        container:"box",
        view:"tree",
});
```

General settings

On the previous step, we have defined primary configuration for Tree. For more available settings, read other articles in this documentation, such as:

- Node templates - available templates for customizing data content;
- Items Selection - describes how to enable selection in the tree;
- Sorting in the Tree and Filtering in the Tree - says about means for sorting and filtering.

Loading data

There are different ways to load data into Tree (read about them in chapter Loading data).

We will use the easiest of the ways and specify data directly in the constuctor with the help of the data property:

```
Loading data into the tree

tree = new webix.ui({
    container:"testA",
```

```
    view:"tree",
    data: [
        {id:"root", value:"Cars", open:true, data:[
            { id:"1", open:true, value:"Toyota", data:[
                { id:"1.1", value:"Avalon" },
                { id:"1.2", value:"Corolla" },
                { id:"1.3", value:"Camry" }
            ]},
            { id:"2", open:true, value:"Skoda", data:[
                { id:"2.1", value:"Octavia" },
                { id:"2.2", value:"Superb" }
            ]}
        ]}
    ]
});
```

Congratulations! Now you can run your app and it'll produce Tree shown at the very beginning.

RELATED SAMPLE: Quick start with DataTree

What's next

What can we suggest you to do next?

- Check the code sample of the mentioned above example (use the link above);
- Read about possible data formats and how you can get data from database;
- Find out about nodes manipulations (adding, deleting nodes etc.);
- Check other supplied samples.

Getting/setting the node

Tree provides a set of methods for manipulating the node value and retrieving the node id.

Getting the node object/value

To get the node object, you should use method getItem:

```
Getting the node object

tree = new webix.ui({
    view:"tree",
    data: [
        { id:"branch1", value:"The Shawshank Redemption", data:[
            { id:"part1", value:"Part 1" },
            { id:"part2", value:"Part 2" }
        ]}
    ]
});

//'node' variable will contain an object of the related tree node
var node = tree.getItem('branch1');

//you can access data members directly
var value = node.value; // ->"The Shawshank Redemption"
```

Item object has the following properties:

- ***id*** - (*string*) the item id
- ***value*** - (*string*) the item value
- ***$level*** - (*number*) the level of nesting (one-based numbering)
- ***parent*** - (*string*) the id of the item's parent
- ***$count*** - (*number*) the number of childs
- ***open*** - (*boolean*) defines whether the node is collapsed or expanded (just for parent nodes)

For example, variable **node** from the code snippet above looks as in:

```
node = {
    $count: 2,
    $level: 1,
    $parent: 0,
    id: "branch1",
    open: true,
    value: "The Shawshank Redemption"
}
```

Setting the node value

To change the current value of a node (the node title), you should use the following technique:

Changing the node title

```
tree = new webix.ui({ view:"tree",...});

var nodeObj = tree.getItem(node_id);
nodeObj.value = new_value;

tree.refresh();
//or
tree.updateItem(node_id, nodeObj);
```

Methods refresh and updateItem lead to one and the same result and don't have any specificity.

Getting the node index

To get the index of a node in the branch (zero-based numbering) you should use the getBranchIndex method:

Getting the node index

```
tree = new webix.ui({
    view:"tree",
    data: [
        { id:"branch1", value:"The Shawshank Redemption", data:[
            { id:"1.1", value:"Part 1" },
            { id:"1.2", value:"Part 2" }
        ]}
    ]
});

var index  = tree.getBranchIndex('1.1'); // -> 0
var index1 = tree.getBranchIndex('1.2'); // -> 1
```

There is also the getIndexById method to get the node index but we don't suggest using it. It's a common method that is inherited by all data-containing components from the DataStore class and not intended for using with tree-like data structures.

Getting the node id

To get the id of a node you can use one of the following methods:

Table 1 Methods for getting the node id

Method	Description
getIdByIndex	returns the item id by its index

Method	Description
getParentId	returns the id of the parent node
getFirstChildId	returns the id of the first child of the specified branch. If there are not any, returns null
getNextSiblingId	returns the id of the next sibling of the specified node
getPrevSiblingId	returns the id of the previous sibling of the specified node

Getting the parent node id

```
tree = new webix.ui({
    view:"tree",
    data: [
        { id:"branch1", value:"The Shawshank Redemption", data:[
            { id:"part1", value:"Part 1" },
            { id:"part2", value:"Part 2" }
        ]}
    ]
});

var parent = tree.getParentId('part1'); // -> 'branch1'
```

Loading data

Webix Tree can get data in the following predefined data formats:

- JSON (default);
- XML.

Details and examples of data formats in tree-like structures

There are two main ways you can specify the data source for Tree:

- Define the data set in the object initialization;
- Use one of the parse or load methods.

Both methods lead to the same result.

If you load XML data you need to set the datatype property in the constructor config or the second parameter of the parse / load method to the name of the expected data type, i.e. 'xml'.

Loading from inline dataset

If you want to specify the data set directly on the page, you can use:

- property data
- method parse

```
Specifying data inline

treedata = [
  { id: "1", value: "Book 1", data: [
    { id: "1.1", value: "Part 1" },
    { id: "1.2", value: "Part 2" }
  ]},
  { id: "2", value: "Book 2", data: [
    { id: "2.1", value: "Part 1" }
  ]}
];

tree = new webix.ui({
  view: "tree",
  ...
  data: treedata
});

//or
```

```
tree.parse(treedata)
```

RELATED SAMPLE: Tree: JSON Dataset

Loading from a data file

To load data from a file on the server, you can use:

property url

method load

Remember, in case of XML data you should specify the data type explicitly as 'xml':

```
tree = new webix.ui({
  view: "tree",
  ...
  url: "data/data.xml",
  datatype: "xml"
});
//or
tree.load('data/data.xml', 'xml');
```

RELATED SAMPLE: Tree: JSON Dataset

Loading linear (list-like) data

It's possible to load plain (list-like) data into the tree. The tree structure is achieved by means of grouping. Grouping is specified through the **$group** key of the scheme parameter.

```
tree = new webix.ui({
  view: "tree",
  scheme: {
    $group: "#make#",
  },
  data: [
    { id: 1, value: "Avalon", make: "Toyota" },
    { id: 2, value: "Corolla", make: "Toyota" },
    { id: 3, value: "Camry", make: "Toyota" },
    { id: 4, value: "Octavia", make: "Skoda" },
    { id: 5, value: "Superb", make: "Skoda" }
  ]
});
```

RELATED SAMPLE: Tree: Plain JSON Data with Client-side Grouping

Loading from a database

To load data from a database on the server you should deal with both client and server sides.

On the client side

You should either define a url property or call the load method that would point to a server REST API endpoint.

```
Static loading from a server database. Client-side code.

tree = new webix.ui({
  ...
  url: "data/tree_data.php"
});
//or
tree.load("data/tree_data.php");
```

On the server side

- you can write the server logic by yourself;
- or you can use dhtmlxConnector

If using dhtmlxConnector, the related server-side script will look like this:

```
Static loading from a MySQL database on the server. Server-side code.

require_once("../connector-php/codebase/data_connector.php");  // includes
the connector file
$conn = mysql_connect('localhost', "root","");  // creates a connection
mysql_select_db("sampleDB");  // selects a database

$data = new TreeDataConnector($conn, "MySQL");  //initializes the
connector object
// loads data from the specified database table
$data->render_table("packages_tree","id","value, state","","parent_id");
```

RELATED SAMPLE: Dynamic Loading

Details and examples of data formats



Expected data formats

JSON

```
[
    { id:"1", open:true, value:"Toyota", data:[
        { id:"1.1", value:"Avalon" },
        { id:"1.2", value:"Camry" },
        { id:"1.3", value:"Corolla" }
    ]},
    { id:"2", value:"Skoda", data:[
        { id:"2.1", value:"Octavia" },
        { id:"2.2", value:"Superb" }
    ]}
]
```

XML

```
<data>
    <item value='Toyota' open='1'>
        <item value='Avalon'></item>
        <item value='Camry'></item>
        <item value='Corolla'></item>
    </item>
    <item value='Skoda'>
        <item value='Octavia'></item>
        <item value='Superb'></item>
    </item>
</data>
```

Available data keys/attributes

You can define the following attributes/keys in the data set:

Attribute	Description
value	the item value (the title of the related node)
open	defines whether the branch will be opened initially
css	the name of a css class that will be applied to the data item
checked	checks the check box related to the item (if exists)
intermediate	sets the 3rd state of the check box related to the item (if exists)
webix_kids	the flag of dynamic loading

JSON data. Using simplified form

You can use the simplified form of JSON and specify data as in:

```
[
    { value:"Toyota", open:true, data:[
        "Avalon", "Camry", "Corolla"
    ]},
    { value:"Skoda", data:[
        "Octavia", "Superb"
    ]}
]
```

RELATED SAMPLE: Tree: JSON Dataset

Compare simplified format with the standard form.

Custom JSON format

To use custom attributes in JSON data set you can define your custom JSON format (read details in article Data drivers).

When you specify a custom format, don't remember to set the expected data type to the name of a new format

Here is a simple example:

```
var myjson = webix.DataDriver.myjson = webix.copy(webix.DataDriver.json);

myjson.child=function(obj){
    if (obj.$level == 1)
        return obj.cars;
};
```

```
tree = new webix.ui({ view:"tree", datatype:"myjson", ... });
```

the 'child' function defines which object properties should store sub-items

As a result, your data will look as in:

```
//full form
[
    { id:"1", open:true, value:"Toyota", cars:[
        { id:"1.1", value:"Avalon" },
        { id:"1.2", value:"Camry" },
        { id:"1.3", value:"Corolla" }
    ]},
    { id:"2", value:"Skoda", cars:[
        { id:"2.1", value:"Octavia" },
        { id:"2.2", value:"Superb" }
    ]}
]
//simplified form
[
    { value:"Toyota", open:true, cars:[
        "Avalon", "Camry", "Corolla"
    ]},
    { value:"Skoda", cars:[
        "Octavia", "Superb"
    ]}
]
```

RELATED SAMPLE: Tree: JSON Dataset

Custom XML format

To use custom tags you can define your custom XML format (read details in article Data drivers).

When you specify a custom format, don't remember to set the expected data type to the name of a new format

Here is a simple example:

```
var myxml = webix.DataDriver.myxml = webix.copy(webix.DataDriver.xml);
myxml.records = "/cars/car";

myxml.child = "model";

tree = new webix.ui({   view:"tree", datatype:"myxml", ... });
```

the 'records' property defines the root tags

the 'child' property (also can be a function that takes the data object as a parameter) defines which tag should store sub-items

As a result, your data will look as in:

```
<cars>
    <car value='Toyota' open='1'>
        <model value='Avalon'></model>
        <model value='Camry'></model>
        <model value='Corolla'></model>
    </car>
    <car value='Skoda'>
        <model value='Octavia'></model>
        <model value='Superb'></model>
    </car>
</cars>
```

RELATED SAMPLE: Tree: XML Dataset

Tree Nodes manipulations

Adding nodes

Generally, to add a node (parent or child) you should use the add method. At what position the element will be added depends on parameters you pass to the method.

Method add accepts 3 parameters:

- **obj** - the item object
- **pos** - the position that the new item should have after being inserted into the tree (zero-based numbering).

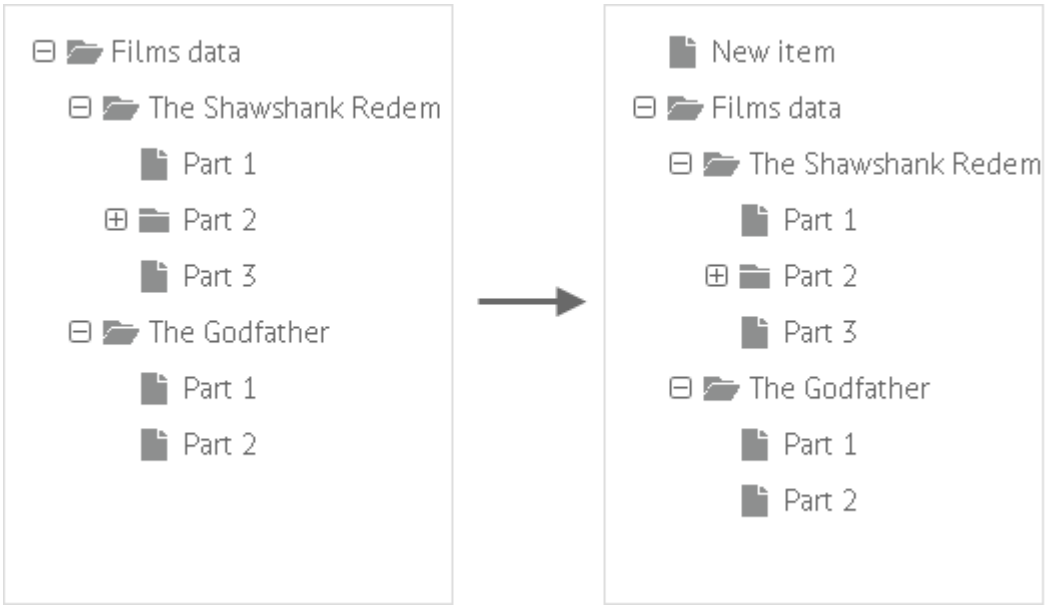
  Note,

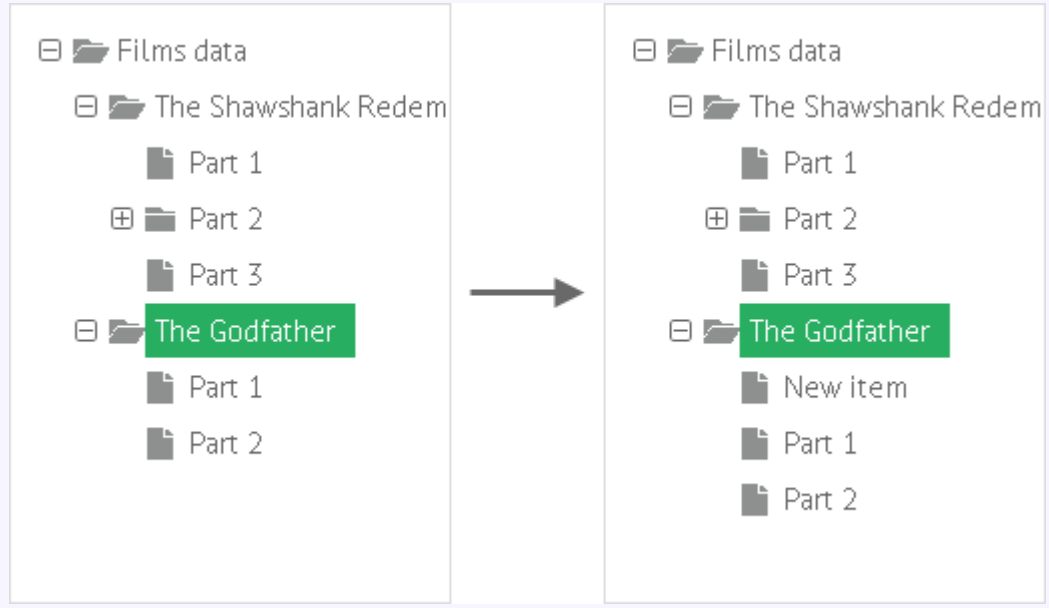
  - - the item will be added as the first child (start position)
  - *-1* - the item will be added as the last child (end position)
- **parent** - (optional) the parent node id. If it's not specified, the item will be added to the root of the tree

Note, you can use the getBranchIndex() method to get the position of a node by its id (read more).

Table 1 Adding nodes

Nesting level	Example

Root node	`tree = new webix.ui({ view:"tree",... });`
	
	`tree.add({ value:"New item"}, );`
	
	

Nesting level	Example
Child node (1st child)	

```
tree=new webix.ui({view:'tree',..});

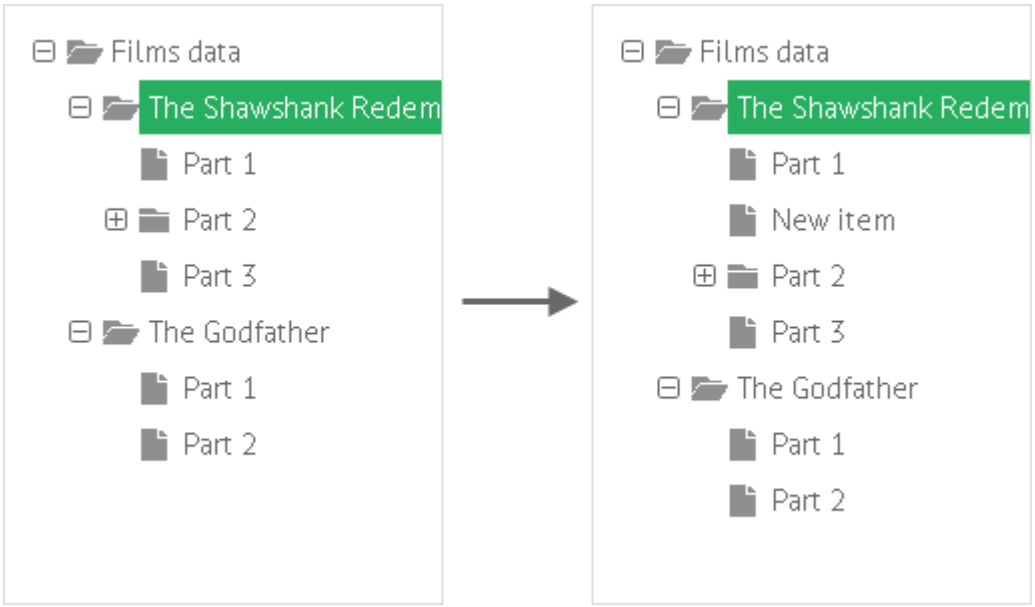
var parentId= tree.getSelectedId();
tree.add( {value:"New item"}, , parentId);
```


Child node (2nd child)

```
tree=new webix.ui({view:'tree',..});

var parentId = tree.getSelectedId();
var pos = tree.getBranchIndex("node2"); //'node2' is the 'Skoda' item's id

tree.add( {value:"New item"}, pos, parentId);
```



Deleting nodes

To delete a node (parent or child) you should use the remove method and specify the id of the required node there:

```
Deleting the currently selected node

tree = new webix.ui({ view:"tree",... });

var nodeId = tree.getSelectedId();
tree.remove(nodeId);
```

Selecting nodes

To select a specific node you should call the select method:

```
Selecting a node

tree = new webix.ui({ view:"tree",... });

tree.select("node2");// 'node2' is the item id
```

For more details, read article Items Selection.

Expanding/collapsing branches

Tree nodes are collapsed and expanded

- by clicking on a **"+/-"** icon or a node (default behavior);
- by clicking on a **node text** provided that **activeTitle** ptoperty is defined for the tree.

```
view:"tree",
activeTitle:true, //false by default
data:[...]
```

RELATED SAMPLE: Tree: JSON Dataset

Tree provides several methods to manage 'open'/'close' state of a node. They are:

Table 1 Managing 'open'/'close' state of a node

Method	Description
open	opens the branch with the specified id
close	closes the branch with the specified id
openAll	opens all branches in the tree
closeAll	closes all branches in the tree

Method	Description
isBranchOpen	checks whether the specified branch is opened
getOpenItems	returns ids of the opened branches

```
Checking whether the currently selected branch is opened

tree = new webix.ui({ view:"tree",... });

var nodeId = tree.getSelectedId();
tree.isBranchOpen(nodeId);
```

Filtering nodes

To filter the tree nodes you should call the filter method:

```
Filtering the tree

tree = webix.ui({ view:'tree', ...});

tree.filter("#value#", "abc");//leaves in the tree just items that contain
text 'abc'
```

RELATED SAMPLE: Filtering

Read more on the topic in article Filtering in the Tree.

Sorting nodes

To filter the tree nodes you should call the sort method:

```
Ascending sort in the tree

tree = webix.ui({ view:'tree', ...});

tree.sort("#value#", "asc");//sorts all nodes (parent, child)
```

RELATED SAMPLE: Sorting

Read more on the topic in article Sorting in the Tree.

Updating the tree

To update the tree you may call one of 2 appropriate methods:

- update
- refresh

Methods lead to one and the same result and don't have any specificity.

```
tree.refresh();
//or
tree.updateItem(node_id, nodeObj);
```

Items Selection

Tree provides an ability to select leaves and branches within it.

Single-item selection

By default, selection is disabled in Tree. To activate it you should set the select parameter to *true*.

```
Enabling selection in Tree

tree = new webix.ui({
  view: "tree",
  ...
  select: true
})
```

RELATED SAMPLE: Selection

Multiple selection

To enable multiple selection in Tree, you should set parameter select to *multiselect*.

```
Enabling multi-selection in Tree

tree = new webix.ui({
  view: "tree",
  ...
  select: "multiselect"
})
```

RELATED SAMPLE: Multiline Selection

Related methods

getSelectedId	method	returns the selected cells
moveSelection	method	moves selection in the specified direction
select	method	selects the specified item
selectAll	method	selects all items
isSelected	method	checks whether the specified item is selected or not
unselect	method	cancels selection of the specified item
unselectAll	method	removes selection from all items

Related events

onAfterSelect	event	fires after a cell is selected
onBeforeSelect	event	fires before a cell is selected
onSelectChange	event	fires when selection is changed in DataTable

Node templates

To customize data presented in the nodes' titles you can use templates. Data template for the tree is set by property template.

With templates you can present almost any content in the tree: images, links, numbers, strings, dates etc.

By default, tree uses the following template for its items:

```
Default tree template

webix.ui({
    view:"tree",
    template:"{common.icon()} {common.folder()}<span>#value#<span>"
});
```

It means that each item features:

- an **expand/collapse node** icon;
- a **folder/file** icon that will automatically change depending on whether a node has children or not;
- data **value** from the datasource.

More about Webix pattern of data interpolation you can find in the Data templates article.

Predefined Templates

Webix tree and treetable offers a set of predefined helpers that aid in item rendering and contain tree-specific elements like item icons, expand/collapse icons and checkboxes.

- **{common.icon()}** - '+'/'-' icons for collapsed/expanded nodes
- **{common.folder()}** - an icon of the folder or file. The folder can be closed or open depending on a node state;
- **{common.checkbox()}** - a checkbox;
- **{common.treetable()}** - (treetable only) - a combination of common.icon() and common.folder() in one line.

All of them are the methods of a **common** object that is accessable by all of the tree items and defined within tree/treetable Type mixin.

String definition

If you set a template by a string, you can include the tree-specific helpers via string:

```
Specifying the template parameter as a string

webix.ui({
    view:"tree",
```

```
    template:"{common.icon()} {common.checkbox()} {common.folder()}
<span>#value#</span>",
});
```

RELATED SAMPLE: 2-state Checkboxes

The tree above will render items featuring an expans-collapse icon, checkbox and file/folder icon.

Function definition

If you set a template via a function, you can include the tree-specific helpers via functions, and they are functions in essense:

```
Specifying the template parameter as a function

webix.ui({
    view:"tree",
    template:function(obj, common){
        return common.icon(obj, common) + common.checkbox(obj, common)
            +common.folder(obj, common)+"<span>"+obj.value+"</span>",
    }
});
```

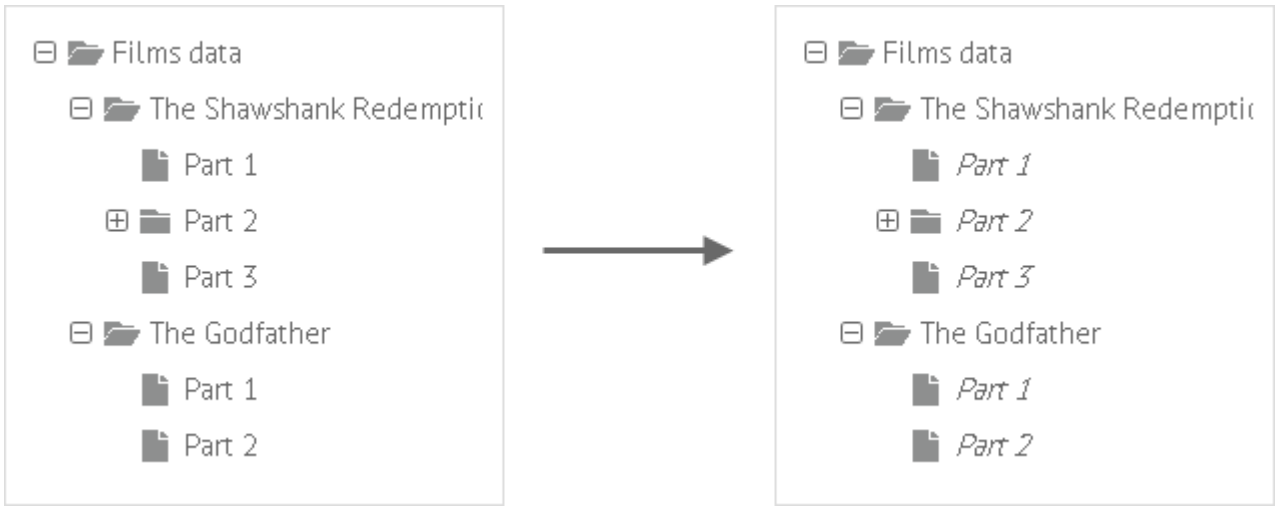
The input parameters of the helpers function are:

- **item object** with the following properties:
  - **item properties** from the dataset;
  - **helper properties** that are added to tree items during data loading and exist only on client-side to maintain tree integrity:
    - **$level** - (*number*) the level of nesting (one-based numbering);
    - **$parent** - (*string*) the ID of thr item's parent;
    - **$count** - (*number*) the number of childs;
    - **open** - (*boolean*) defines whether the node is collapsed or expanded (only for parent nodes).
- **common** object that provides 4 functions:
  - **icon(obj, common)** - draws the '+'/'-' icons for collapsed/expanded states of nodes
  - **folder(obj, common)** - draws an icon of the folder
  - **checkbox(obj, common)** - draws a check box
  - **treetable(obj, common)** (treetable only) - combines *common.icon()* and *common.folder()* in one line.

Benefits of function definition

Template function add more freedom in item template definition as numerous conditions can be taken into an account.

Let's assume, you want to show items that have the nesting level greater than 2 in italic font. In thic case you can specify the data template as in:




```
webix.ui({
  view:"tree",
  template: function(obj, common){
    if (obj.$level>2) {
       return
common.icon(obj,common)+common.folder(obj,common)+"<i>"+obj.value+"</i>";
    } else {
       return common.icon(obj,common)+common.folder(obj,common)+obj.value;
    }
  }
});
```

You can also define additional common elements within the component type.

Predefined Templates: Use Cases

{common.icon()}




```
webix.ui({
    view:"tree",
    template:"{common.icon()} #value#"
```

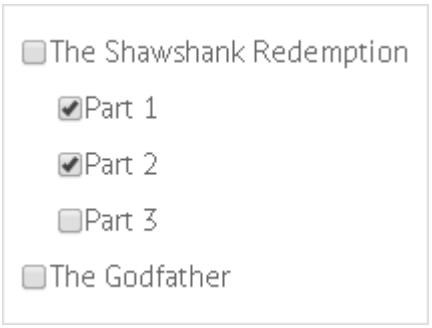
```
})
```

{common.folder()}



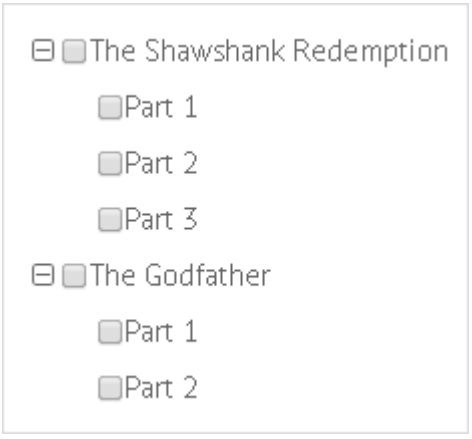

```
webix.ui({
    view:"tree",
    template:"{common.folder()} #value#"
});
```

{common.checkbox()}




```
webix.ui({
    view:"tree",
    template:"{common.checkbox()} #value#"
});
```

{common.icon()} {common.checkbox()}



```
webix.ui({
    view:"tree",
    template:"{common.icon()}{common.checkbox()} #value#"
});
```

{common.icon()}{common.folder()} or {common.treetable()}




```
webix.ui({
    view:"tree",
    template:"{common.icon()}{common.folder()} #value#"
});

The same for [treetable](desktop/treetable.md) can be rendered via:

webix.ui({
  view:"treetable",
  columns:[
    { id:"value",
      template:"{common.treetable()} #value#"
    },
    {...}
  ]
});
```

{common.icon()}{common.folder()}{common.checkbox()}




```
webix.ui({
    view:"tree",
    template:"{common.icon()}{common.folder()}{common.checkbox()} #value#"
```

```
});
```

Changing Predefined Templates

The above mentioned template methods are parts of the tree type, a set of methods used for rendering tree items.

Tree type can be extended or altered:

```
webix.ui({
    view:"tree",
    type:{
        folder:function(obj){
            if(obj.$level == 1)
                return "<span class='webix_icon fa-folder-open'></span>";
            if (obj.$level == 2)
                return "<span class='webix_icon fa-video-camera'></span>";
        }
    }
});

webix.ui({
    view:"tree",
    type:{
        my_folder:function(obj){
            //code
        }
    }
});
```

RELATED SAMPLE: Custom Type

If you need to apply this type to several trees, you can define the type object separately and then refer to it by its **name**:

```
webix.type(webix.ui.tree, {
   name:"awesome",
   folder:function(obj){
       //code
    }
});

var tree2 = webix.ui({
   view:"tree",
   type:"awesome",
});
```

RELATED SAMPLE: Custom Type

More about type implementation for data components can be found in the Type implementation article.

Checkboxes in the tree

In this chapter we will talk about checkboxes in tree nodes: how to add, customize them or provide the 3-state behaviour.

Adding checkboxes to nodes

Tree provides a predefined template for check boxes, which is - *{common.checkbox()}*
(read more about templates in chapter Node templates).



So, to add checkboxes to tree nodes you should specify the template property like this:

```
Adding checkboxes to tree nodes

tree = new webix.ui({
    view:"tree",
    template:"{common.icon()} {common.checkbox()} {common.folder()}
#value#"
    ...
});
```

{common.icon()} template adds '+'/'-' icons to the nodes, {common.folder()} - adds folder icons.

RELATED SAMPLE: 2-state Checkboxes

3-state checkboxes

In addition to standard 2-state behaviour, Tree supports 3-state checkboxes.

Table 1 Types of checkbox behaviour

Behaviour	Description
2-state (standard)	◦ When the user checks/unchecks a parent node - this only node is checked/unchecked ◦ When the user checks/unchecks a child node - this only node is checked/unchecked
3-state	◦ When the user checks/unchecks a parent node - this parent node and all its child nodes (of each nesting level) are checked/unchecked ◦ When the user checks/unchecks a child node - this only node is checked/unchecked

To provide 3-state behaviour for tree checkboxes you should set the threeState property to *true*:

```
Activating 3-state behaviour for checkboxes

tree = new webix.ui({
    view:"tree",
    template:"{common.icon()} {common.checkbox()} {common.folder()}
#value#",
    threeState: true
    ...
});
```

RELATED SAMPLE: 3-state Checkboxes

Manipulations with checkboxes

Checkboxes can be programmatically checked and unchecked as well as it is possible to get checked items and define, whether the item is currently checked or not.

- **checkItem**(id)/ **uncheckItem**(id) - checks/unchecks tree node with the specified ID;
- **getChecked()** - returns an array of IDs of the checked items;
- **isChecked(id)** - helps find out whether the specified node is checked at the moment.

```
tree.checkItem(tree.getSelectedId());

tree.uncheckItem(tree.getSelectedId());
```

Checkboxes can be used for **component refreshing** just from browser.

Generally, components are refreshed by applying the **refresh()** method to them, yet it can be called each time tree checkbox changes. The item wth this checkbox will be refreshed:

```
webix.ui({
    view:"datatable",
    checkboxRefresh:true
```

```
});
```

Sorting in the Tree

Tree allows you to sort nodes on the client side. Note, along with the parent nodes, child nodes are sorted as well.

Data is sorted as strings and by levels, starting from the latest nesting level.

Common

Sorting in Tree can be invoked by calling the sort method. The method can be called on some event or action, i.e. button click or page load.

```
Sorting Tree on button click

<input type='button' onclick='tree.sort("#value#", "asc");'>

<script type="text/javascript" charset="utf-8">
    tree = webix.ui({ view:'tree', ...})
</script>
```

RELATED SAMPLE: Sorting

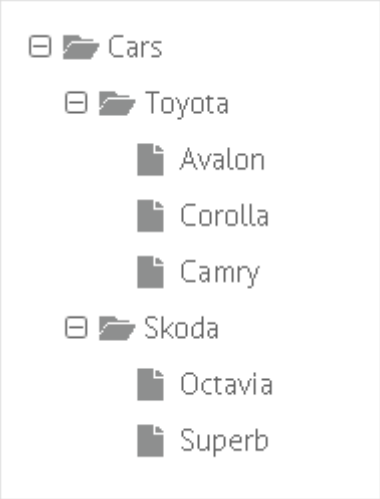
Custom sorting functions

If you want to apply a custom sorting behaviour, you should define the related logic in a function and invoke this function in the sort method.

This function is called for each pair of adjacent values and return 1,-1 or 0:

- **1** - an object with the first value in pair must go before the second one;
- **-1** - the second object goes before the first one;
- - the order of both objects doesn't change.

Let's assume you have a tree like this:



And want to sort only car makes and keep the childs as they are. As the default sorting affects all nodes you need to provide a custom logic. It may look as in:

```
Sorting the tree by one level

<input type='button' value='sort car makes'
onclick='tree.sort(sortMakes)'>

<script>
 function sortMakes(a,b){
         if (a.$level == 2)
         {
             a = a.value;
             b = b.value;
             return a>b?1:(a<b?-1:);
         }
         else{
             return ;
         }
 }
 ...
</script>
```

Note, Tree sorts data by levels starting from the lastest nesting level, i.e. at first car models of each make are sorted, then car makes themselves.

RELATED SAMPLE: Custom Sorting

Filtering in the Tree

Tree allows you to filter nodes on the client side.

Filtering is Tree can be invoked by calling the filter method. The method can be called on some event or action, i.e. button click or page load.

```
Sorting Tree on button click

<input type='button' onkeypress='tree.filter("#value#", this.value);'>

<script type="text/javascript" charset="utf-8">
     tree = webix.ui({ view:'tree', ...})
</script>
```

RELATED SAMPLE: Filtering

Filtering modes

The library provides special property **filterMode** to define how the tree items will be filtered.

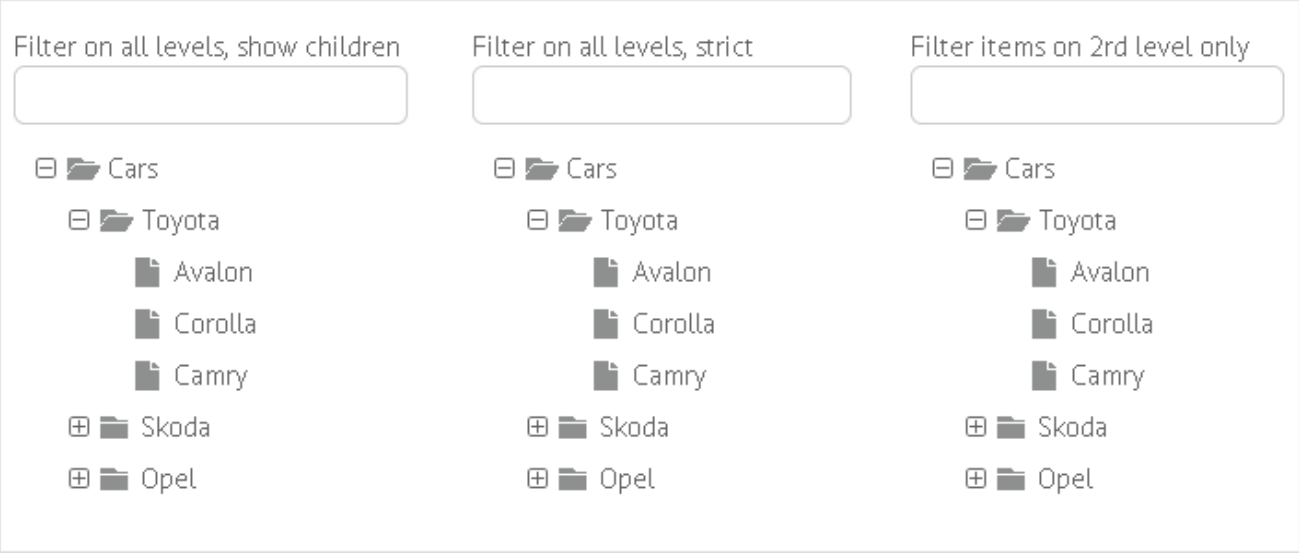
The **filterMode** property is an object that can contain 3 attributes:

- **showSubItems** - (*boolean*) defines whether the tree should display children of the filtered items (*true*) (even if they don't match the filtering condition). The default value - *true*.
- **openParents** - (*boolean*) - defines whether parents of the filtered items should be expanded (even if they don't match the filtering condition) so that user can effortlessly see the result of filtering. *True* by default.
- **level** - (*number*) sets the nesting level to filter items from (one-based numbering).

```
Using the filterMode parameter

webix.ui({
    view:"tree",
    filterMode:{
        showSubItems:false,
        level:2
    }
});
```

RELATED SAMPLE: Filtering



Custom filtering function

For example, if you add an input and button to the page and want to filter Tree by clicking on it, you code may look like this:

```
Custom filtering function
```

```
<input type="text"><input type="button" value='filter'
onclick='filterText(this);'>

<script>
    function filterText(node){
        var text = node.previousSibling.value;
        if (!text) return tree.filter();

        tree.filter(function(obj){  //tree is a Tree instance
            return obj.year == text;
        })
    }
</script>
```

Editing in the Tree

Initially, Tree is a non-editable component. But you can easily make Tree editable by extending it with a special class - EditAbility.

Generally, the technique of making Tree editable looks like this:

- Create a new component and inherit it from ui.tree and EditAbility classes.
- Specify 3 parameters (inherited from EditAbility) in the constructor:
  - **editable** - enables editing in the component
  - **editor** - sets the type of the editor
  - **editValue** - the data property that will be edited

By default, editors are opened on a single click.

```
Making Tree editable

webix.protoUI({
    name:"edittree"
}, webix.EditAbility, webix.ui.tree);


var tree = new webix.ui({
    view:"edittree",
    ...
    editable:true,
    editor:"text",
    editValue:"value"
});
```

RELATED SAMPLE: Editing in Tree

Types of editors

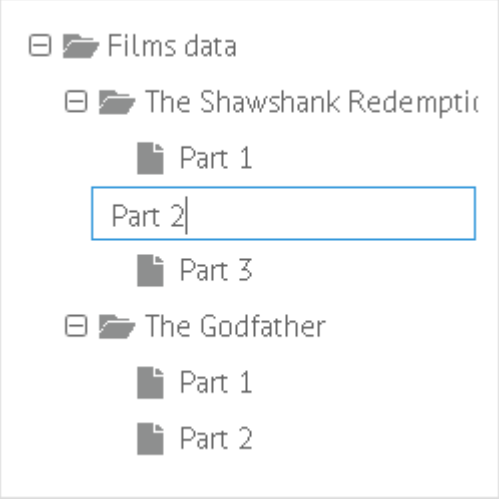
There are various types of data editors in the Webix library. You can use any of them in different components. To get more details read the article Data editors.

Below we'll describe 2 most frequently used editors for the tree:

- text
- select

text

A basic text editor.




```
Specifying the 'text' editor for the tree

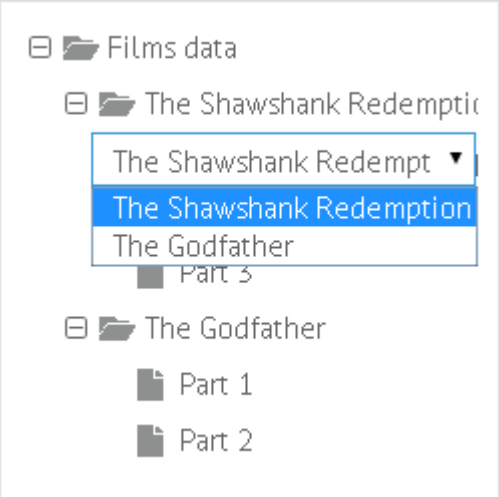
webix.protoUI({
    name:"edittree"
}, webix.EditAbility, webix.ui.tree);

var tree = new webix.ui({
    view:"edittree",
    editable:true,
    editValue:"value",
    ...
    editor:"text"
});
```

RELATED SAMPLE: Editing in Tree

select

A dropdown list.



Specifying the 'select' editor for the tree

```
webix.protoUI({
    name:"edittree"
}, webix.EditAbility, webix.ui.tree);


var tree = new webix.ui({
    view:"edittree",
    editable:true,
    editValue:"value",
    ...
    editor:"select",
    options:["The Shawshank Redemption", "The Godfather"]
});
```

Denying editing specific items

To deny editing specific items, branches, levels etc. you can use the onBeforeEditStart event inherited from EditAbility. The event fires right before the user opens the editor and gets the id of the edited item as a parameter.

To deny editing of a item - return *false* for it within the appropriate event handler.

Making editable only the 2nd nesting level

```
webix.protoUI({
    name:"edittree"
}, webix.EditAbility, webix.ui.tree);

var tree = new webix.ui({
    view:"edittree",
    ...
    editable:true,
    editor:"select",
    editValue:"value",
    data: [
        { id:"1", open:true, value:"The Shawshank Redemption", data:[
            { id:"1.1", value:"Part 1" },
            { id:"1.2", value:"Part 2" },
            { id:"1.3", value:"Part 3" }
        ]},
        { id:"2", value:"The Godfather", data:[
            { id:"2.1", value:"Part 1" },
            { id:"2.2", value:"Part 2" }
        ]}
    ]
});
//only film parts (i.e. 'Part1', 'Part2') can be edited
tree.attachEvent("onBeforeEditStart", function(id){
```

```
    if ( this.getItem(id).$level != 1)
        return false;
});
```

Copy-pasting data

To enable the possibility to copy/paste Tree data by the CTRL+C/CTRL+V keyboard shortcuts you should use parameter clipboard.

The parameter can have one of 3 values:

'modify' (or true. It's the default value);

'insert';

'custom'.

Each of the types has its specificity and define its behavior of copying.

```
Setting the desired behavior of copying

tree = new webix.ui({
    view:"tree",
    ...
    clipboard:"insert"
});
```

RELATED SAMPLE: Pasting New Items

'Modify' copying

It's the default type and can be also specified by setting the *true* value.

The type of copying can be characterized as in:

- When you copy the selected node in the tree, you copy the node title (doen't matter it's a parent or child node);
- When you paste a text to the tree, it will be pasted as the title of the selected node.

```
Setting the 'modify' behavior of copying

tree = new webix.ui({
    view:"tree",
    ...
    clipboard:"insert"
});
```

RELATED SAMPLE: Pasting Items' Titles

'Insert' copying

- When you copy the selected node in the tree, you copy the node title (doen't matter it's a parent or

child node);

- When you paste a text to the tree, it will be added as a child item of the selected node.

```
Setting the 'insert' behavior of copying

tree = new webix.ui({
    view:"tree",
    ...
    clipboard:"insert"
});
```

RELATED SAMPLE: Pasting New Items

Custom copying

The '*custom*' value allows you to specify a custom logic for the paste operation.

To apply a custom 'paste' behaviour to the tree:

- Set the clipboard parameter to *true*.

```
tree = new webix.ui({
    view:"tree",
    ...
    clipboard:"custom"
});
```

This command will cancel the predefined behaviour for the paste operation.

- Specify custom logic in the onPaste event handler.

```
//the code does nothing but alerts messages
tree.attachEvent("onPaste", function(text) {
    webix.message("Node is pasted: " + text);
});
```

Tree invokes the onPaste event when the user presses CTRL+V keys combination.

RELATED SAMPLE: Custom Clipboard Operations

Styling

You can use CSS in conjunction with Tree nodes to archieve the desired look-and-feel.

In this article you will know how:

- to fine-tune the style of a whole tree or specific nodes;
- to add tree lines;
- apply templates;
- customize icons.

Styling a whole tree

To apply some style to a whole tree you can use the css parameter.




```
<style>
    .my_style  {
        font-style: italic;
        color:blue;
    }

</style>

<script>
    webix.ui({
        view:"tree",
        css:"my_style",
        ...
    })
</script>
```

Styling specific nodes

Generally, to style a specific item you can use the **css** attribute inside the data source.

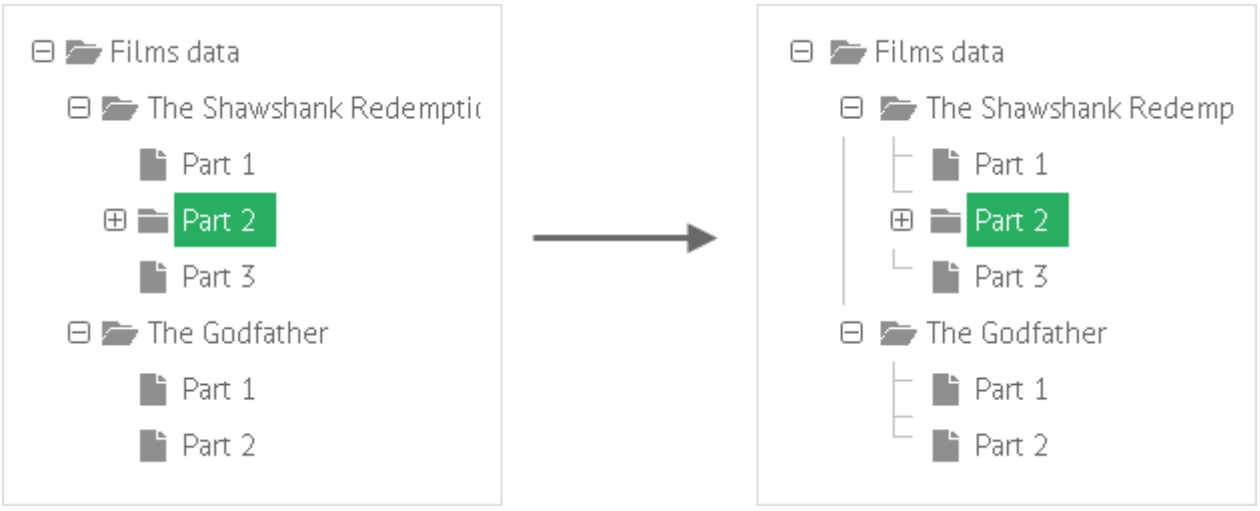



```
<style>
    .my_style{
        background-color:#FFAAAA;
    }
</style>

<script>
grid = new webix.ui({
    view:"tree",
    data: [
       {id:"root", value:"Films data", open:true, data:[
       { id:"1", open:true, value:"The Shawshank Redemption", data:[
            { id:"1.1", value:"Part 1" },
                { id:"1.2", value:"Part 2", $css:"my_style" },
                { id:"1.3", value:"Part 3" }
            ]},
            { id:"2", value:"The Godfather", open:true, data:[
                { id:"2.1", value:"Part 1" },
                { id:"2.2", value:"Part 2", $css:"my_style" }
            ]}
        ]}
    ]
});
</script>
```

Adding lines

To add lines to tree nodes, set the type parameter to the '*lineTree*' value.




```
Adding lines to tree nodes

tree = new webix.ui({
    view:"tree",
    ...
    type:"lineTree"
});
```

RELATED SAMPLE: Lines

Note, you still have a possibility to define your custom style for lines and apply it through the css parameter (see details in the first part).

Templates

One of ways to fine-tune the style of the tree is define some template for it.




```
Using templates for styling

webix.ui({
    view:"tree",
    template:"{common.icon()}{common.folder()}<div
style='color:blue;'>#value#</div>"
});
```

Read more about using templates in chapter Node templates.

Custom Folder and Files Icons

The look-and-feel of tree icons is set via **CSS classes** applied to **definite parts** of it and you can change it in the same way by:

- **redefining existing Webix CSS classes**. It changes icons for all tree nodes;
- **setting custom images** as icons to **desired tree nodes**. For these needs, icons are stored in the dataset and applied via
  - custom CSS classes;
  - node templates.
- modifying tree type to add extra helpers that will render the needed node elements (including FontAwesome Icons).

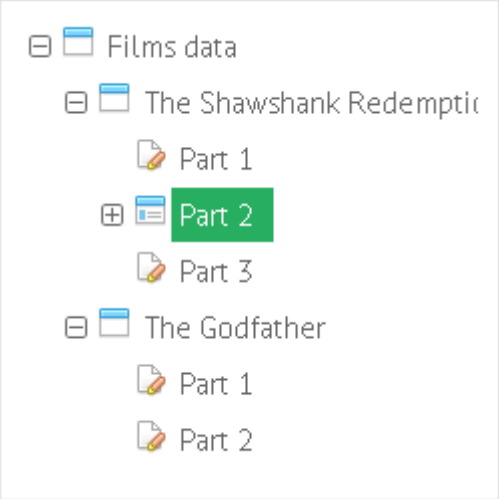
Redefining existing Webix CSS classes

Study Tree CSS image map to learn how classes are distributed. There you'll find that folder and file feature the following CSS:

- **webix_tree_folder** - applied to folders in a close state;
- **webix_tree_folder_open** - applied to folders in the open state;
- **webix_tree_file** - applied to files (elements with no children).

The simplest way to change folder and file icons for all tree nodes is to **redefine default** ones setting new **background image** and **position**:

```
//customizing all folders and files in a tree
.webix_tree_folder{
    background-image: url("icons/folder.png");
    background-position: 0px 4px;
}
.webix_tree_file{
    background-image: url("icons/file.png");
    background-position: 0px 4px;
}
```



RELATED SAMPLE: Custom Folders

Using Custom Images as Icons

To set icons to **desired tree nodes**, you should store their titles in the data source:

If icon titles are stored under an **icon** property in the dataset, they are applied as **webix_icon__name** CSS class:

```
var data_with_icon = [
    { id:"root", value:"Films data", icon:"home", open:true, data:[
        { id:"1", icon:"puzzle", value:"Part 1" }]
    }
];
webix.ui({
    view:"tree",
    data:data_with_icon
});
```

The root item gets **webix_icon_home** CSS class. All you have to do is to set the rule for this class:

```
.webix_tree_home{
    background-image: url("icons/home.png")
}
```

RELATED SAMPLE: Icons

If icon titles are stored under an **image** property in the dateset, they are applied within a node template:

```
var data_with_icon = [
    { id:"root", value:"Films data", image:"home", open:true, data:[
        { id:"1.1", image:"puzzle", value:"Part 1" }]
    }
];

webix.ui({
    view:"tree",
```

```
    template:"<img src='icons/#image#.png' style='...'> #value#",
    data:data_with_icon
});
```

RELATED SAMPLE: Custom Icons

You can study Webix **tree node templates** in detail separately.

Drag-and-drop support

Tree has built-in support for internal/external drag-and-drop.

Common use

To enable drag-and-drop support for a tree, set the drag parameter to *true*:

```
Enabling drag-and-drop support for Tree

tree = new webix.ui({
    view:"tree",
    ...
    drag:true
})
```

RELATED SAMPLE: Enabling Drag-and-drop in Tree

The 'order' mode

There is a special drag-and-drop mode - 'order'.

In such a mode, items can't be dragged out of the tree container boundaries. Also, while dragging an item the remaining items are automatically rearranged so that when you drop the item to the final destination the remaining items won't need to do any rearrangements.

To enable the d-n-d support and activate the 'order' mode, just set the drag parameter to *order*.

```
Activating the 'order' mode for items

webix.ui({
        view:"tree",
        ...
        drag:'order'
});
```

RELATED SAMPLE: The 'order' DnD Mode

The 'source-target' mode

When you have several trees on the page, you may want to set specific tree(s) as the source of dnd (to drag items from) and specific tree(s) as the target of dnd (to drag items to).

- To set a tree as the dnd source (the user can drag items from the tree but can't drag to it), you should set the drag parameter to *source*.

- To set a tree as the dnd target (the user can drag items to the tree but can't drag from it), you should set the drag parameter to *target*.

```
Using the 'source-target' mode

//you can drag items only from treeA to treeB.
//Dragging within treeA is denied. Dragging from treeB to treeA is denied.
treeA = new webix.ui({
    view:"tree",
    ...
    drag:"source"
});

treeB = new webix.ui({
    view:"tree",
    ...
    drag:"target"
});
```

RELATED SAMPLE: Custom Dropping Behaviour

Advanced

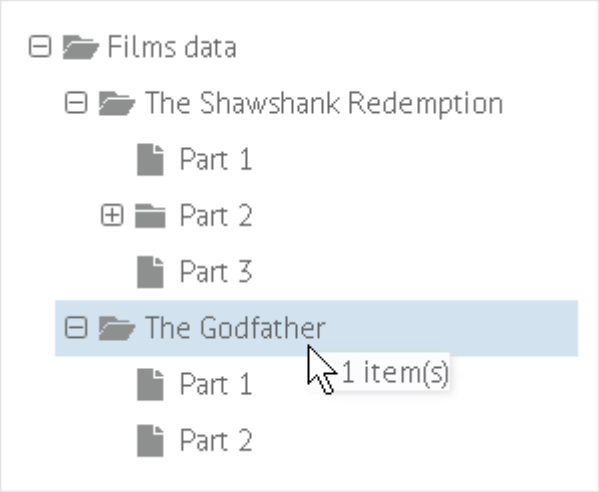
Information stated in the Common part is enough in most cases. The current part should be used just if you want to customize existing drag-and-drop behaviour.

The part will describe the next things:

- Custom text of the dragging items;
- Denying dragging specific items;
- Custom dropping behaviour;
- Related events and the order in which these events are invoked.

Custom text of the dragging items

To redefine the text displaying for the dragging item(s), you should use the onBeforeDrag event.
The desired template is set through the **context.html** property. You can use any HTML while specifying the value for the property.




```
Displaying a custom text for dragging items

treea.attachEvent("onBeforeDrag", function(context, ev){
    context.html = "  "+context.source.length+" item(s)";
});
```

RELATED SAMPLE: Custom Text of the Dragging Items

Denying dragging specific items

To deny dragging specific items you can use the onBeforeDrag event and return *false* each time you want to block the operation.

```
Denying dragging even items

tree.attachEvent("onBeforeDrag", function(context, ev){

    if (tree.getItem(context.source).$level == 2){
        return true;     // allows dragging items with the 2nd nesting
level
    }
    return false;        // denies dragging in any other case
});
```

From now on, each time d-n-d is started, the item level will be checked, and the item can be dragged only if its level equal to 2.

Custom dropping behaviour

To specify custom behaviour for the 'drop' operation you can use the onBeforeDrop event.

Let's assume you want to specify the following behaviour:

- When you drag item(s) to the closed folder, the item is inserted as a subling;
- When you drag item(s) to the opened folder, the item is inserted as a child.

```
Custom dropping behaviour
```

```
tree.attachEvent("onBeforeDrop", function(context, ev){
    if (this.getItem(context.target).$count &&
this.getItem(context.target).open){
        //drop as the first child
        context.parent = context.target;
        context.index = ;
    } else {
        //drop as the next subling
        context.index++;
    }
});
```

RELATED SAMPLE: Custom Dropping Behaviour

RELATED SAMPLE: Custom Dropping Behaviour. Denying Dropping

Related events

In the default scenario there is no need to use any of the mentioned events, because Tree will process all the operations on its own. Use the events only when the default behavior needs customizing.

The following events are generated while d-n-d process:

- onBeforeDrag - fires before the mouse button is pressed and moved over a draggable item
- onBeforeDragIn - fires before a dragged element is moved over the droppable area
- onDragOut - fires when a dragged element is moved outside of the droppable area
- onBeforeDrop - fires before a dragged element is released over the droppable area
- onAfterDrop - fires after drag-n-drop was finished

Events order

Let's assume you have 2 same trees and drag an item from the 1st tree to the 2nd one.



So, here is the order in which events are invoked in trees:

onBeforeDrag fires in **tree1**.

*Returning '*false*' from the event handler will block the current dnd operation.*

onBeforeDragIn / onDragOut fire in **tree1** as you drag the item within the **tree1**'s container.

*Returning '*false*' from the event handler will deny dropping to the related tree.*

onBeforeDragIn / onDragOut fire in **tree2** as you drag the item within the **tree2**'s container.

*Returning '*false*' from the event handler will deny dropping to the related tree.*

onBeforeDrop fires in **tree2**.

*Returning '*false*' from the event handler will block the current dnd operation.*

onAfterDrop fires in **tree2**.

Event handling with Tree

Tree supports various events that can be used to provide a custom behaviour for your tree.

There are 2 ways you can add a handler to the event:

- through the method attachEvent();
- through the parameter on.

Event names are case-insensitive

Method attachEvent()

You can attach several handlers to the same event and detach them using two respective methods:

```
A general way to attach/detach the event handler

// to attach event
var myEvent = $$("treeId").attachEvent("onItemClick", function () {
// event handler code
})

//to detach event
$$("treeId").detachEvent(myEvent);
```

Parameter 'on'

With the help of parameter on you can also attach any event(s) to a Tree object. But in contrast to using the attachEvent method you can't detach attached events later.

```
Attaching the event handler through parameter 'on'

webix.ui({
  view:"tree",
  ...
  on: {"itemClick": function () {alert("item has just been clicked");}}
);
```

Cancelable Events

All events with subword '**onbefore**' can be cancelled.

To cancel some event you should return **false** within the appropriate event handler.

```
Cancelling the event handler

var myEvent = $$("treeId").attachEvent("onBeforeSelect", function () {
  ... // some event handler code
  return false;
})
```

Accessible objects and data

Inside the event handler you can refer to the holder component through keyword **this**. Besides, most event handlers get incoming argument(s), like the **id** of a data item (see tree events to know exactly what arguments are passed inside event handler).

Btw, using the **id** of a data item you can access this item itself and all its properties. For example:

```
Referring within the event handler

$$("treeId").attachEvent("onAfterSelect",function(id){
  parentId = this.getItem(id).parent;
})
```

Treetable

API Reference

- Methods, properties and events
- Samples

Overview

Ui-related treetable inherits from view. Being a hybrid between DataTable and tree it shares a number of their properties and methods.

	Film title	Mode
1	The Shawshank Redemption	
1.1	Part 1	alpha
1.2	Part 2	beta
1.2.1	Part 1	beta-twin
1.2.2	Part 1	beta-twin
1.3	Part 3	gamma
2	The Godfather	

Initialization

Note that you need to set **{common.treetable()}** template for a column that should display the tree. Otherwise, you'll get standard datatable.

```
webix.ui({
    view:"treetable",
    columns:[
        { id:"id",  header:"", width:50},
        { id:"value", header:"Film title", template:"{common.treetable()}
#value#"},
        { id:"chapter", header:"Mode",  width:200}
    ],
    data: "..." //dataset, variable or path
```

```
})
```

RELATED SAMPLE: Basic initialization

Tree table is populated with hierarchical data coming in **XML** and **JSON** data formats. IDs of the coulmns must coincide with data keys from the initial dataset

Working with Treetable

Common tasks here include the following and coincide with those of the datatable and tree:

1 . Data Loading including Dynamic Loading of large datasets inportions;

RELATED SAMPLE: Static and dynamic loading

2 . Filtering - built-in and custom filtering in various filter modes;

RELATED SAMPLE: Filtering in TreeTable

3 . Selection by cells, row and columns in multiselect and blockselect modes;

RELATED SAMPLE: Selecting items in TreeTable

4 . Item Editing by click or dblclick or any custom action;

RELATED SAMPLE: Editing data in TreeTable

5 . Using Math in Treetable - built-in logic for storing data as mathematic formulas as well as editing these formulas directly in treetable.

RELATED SAMPLE: Using formulas

6 . Data Grouping - built-in logic for preseting hierarchical data in a tree-like manner;

RELATED SAMPLE: Aggregating data while grouping

7 . Data Export to Excel and PDF

RELATED SAMPLE: Export to PDF, Excel

8 . Copy-Paste Operations

RELATED SAMPLE: Different copy-paste modes

Related Articles

- Data Object Operations
- Adding Active Elements to Nodes
- Tree

- Node templates
- Sizing Components
- Resizing
- Redefinition of the Components
- Treetable CSS Image Map

Unitlist

API Reference

- Methods, properties and events
- Samples

Overview

Unitlist presents data in groups derived from initial non-hierarchical dataset. Data items are sorted and then united by common values.



Initialization

```
List items are groupped by title

webix.ui({
    view:"unitlist",
    sort:{
        by:"#title#",
        dir: 'asc'
    },
    uniteBy:function(obj){
        return obj.title.substr(,1);
    },
    type:{//setting item properties, optional
```

```
        height:50,
        headerHeight:30,
    },
    template:"#title#</br>#year#<br/>#votes#",
    data: set
});
```

RELATED SAMPLE: Unitlist

- The **uniteBy** property groups data according to the specified criterion, the unit value (here it's the first letter to the data item title);
- In addition it sets **unit headers** (unit values) and, additionally, sets template for them;
- **Sorting** is applied to unit headers as well as to data items within the units.

```
Unit header template

view:"unitlist",
uniteBy:function(obj){
    return "<span style='background-color:"+obj.color+";'>"+
            obj.title.substr(,1)+"</span>";
}
```

Working with Unitlist

Working with Unit Values

Unitlist API makes it possible to get text values of unit headers (unit values) as well as get data items that comply with this or that unit criterion.

- **getUnits()** - returns JS array of unit values defined by the **uniteBy** property. The values are arranged in the alpabetical order, numbers first;
- **getUnitList(id)** - returns the ID (or, more often, IDs) of data records united by "unit" criterion.

```
var units = $$('list').getUnits(); // -> // -> [1, C, M, T]

var unit = $$('unit').getInputNode().value; //getting value of the combo
box text field
var id = $$("list").getUnitList(unit);
```

RELATED SAMPLE: Working with UnitList Items

If you know the ID of any data item, you can easily get to all its properties:

```
var title = $$("list").getItem(id).title;
```

If there're several data items that comply to the unit value criterion, they are returned in a **JS array**, and should be treated accordingly.

Common Functionality:

- Data Loading.
- Defining Data Template.
- Adding Active Elements to Unitlist Items
- Adding/Deleting Items.
- Editing Data.
- Data Filtering and Sorting
- Selection
- Paging

Note that there's no built-in possibility to edit data with unitlist. You should create a prototype **editlist** object beforehand.

Related Articles

- Data Object Operations
- Sizing Components
- Resizing
- Changing Component Properties
- List
- Grouplist
- X-list
- Unitlist CSS Image Map

Uploader

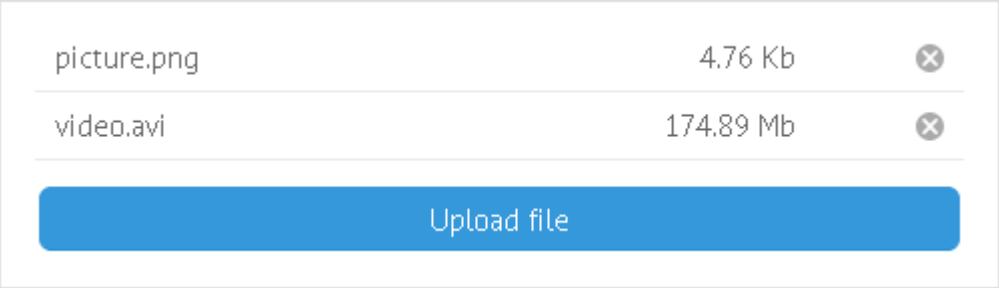
API Reference

- Methods, properties and events
- Samples

Overview

Uploader is a component that allows users to upload files. It allows browsing directories on your machine to choose the necessary file(s) and send them to server.

Uploader can be used **standalone** as well as integrated into **form** or **UI component** (in an "invisible" mode for the latter).



Initialization

```
webix.ui({
    view:"uploader",
    id: "uploader_1",
    value:"Upload file",
    link:"mylist",
    upload:"php/upload.php"
})
```

RELATED SAMPLE: Basic List Uploader

Working with Uploader

- Uploading Files
- Configuring Uploader
- Progress Bar for Uploader
- Uploader and Form Integration

- Binding Uploader to Any Page Element

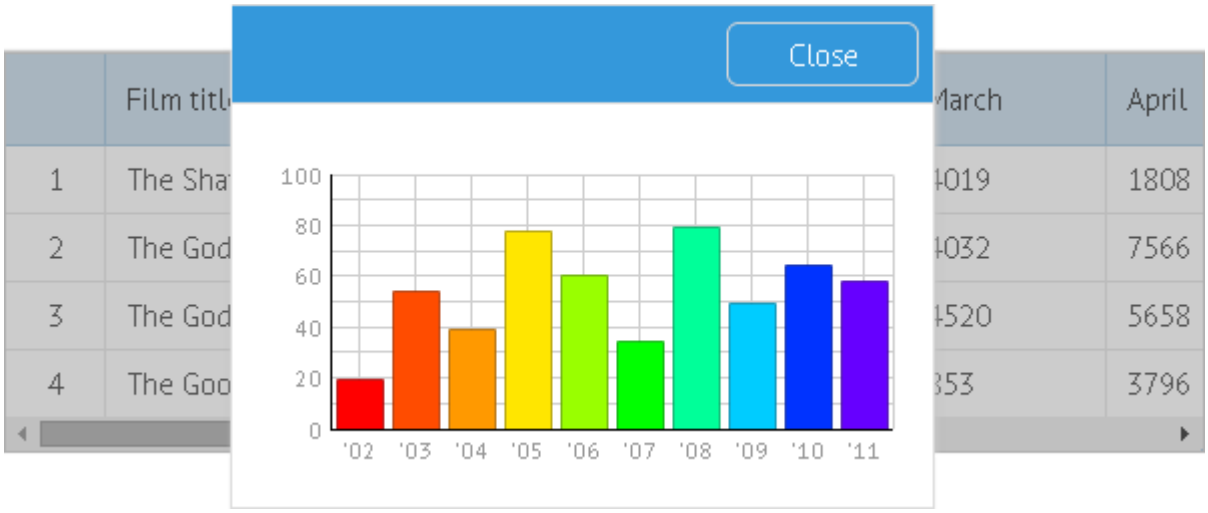
Window

API Reference

- Methods, properties and events
- Samples

Overview

Window is a ui-related component that looks like a standard on-screen window. It inherits from view but differs from other components in its character. It lies not within your app's layout but as if above other components. The same is true about popup.



Initialization

```
webix.ui({
    view:"window",
    id:"my_win",
    head:"My Window",
    body:{
        template:"Some text"
    }
})
```

RELATED SAMPLE: Windows: Basic Initialization

Comments:

- Standard window contains **head** and **body** sections. However, head can be omitted if you don't need it;

- To make window visible, you should apply **show()** method after its initialization;
- A movable window contains **move:true** property in its constructor;
- By default windows are **non-modal**. When such window is focused, it can be closed with an 'Esc' hotkey. Modality is described below.

Working with Window

Head

Head differs from window body in style and may contain:

- a text header that is set as string value of head property;
- a toolbar with controls. The most common are text **label** and **"Close" button**.

```
{view:"button", label:"Close", width:70, click:("$$('my_win').close();")}
        // "my_win" is your window's ID
```

Please note that there exist two functions to remove a window from the screen - **close()** and **hide().** The first one completely removes the window from the app and there will be no way to show it back.

If you want to change the window's visibility rather than remove it, use **hide().** Later on, it can appear on the page on calling **show()** method for it.

Body

This section may contain any view to match your needs.

Text template is set by initing **template** component. It can be defined without direct initialization: {template:"some text"}.

```
Window with Datatable

webix.ui({
    view:"window",
    body:{
        view:"datatable",
        ...
    }
})
```

RELATED SAMPLE: Window with a Component Inside

Window Sizing and Positioning

The must-have parameters for the window are **width** and **height**. If you don't set tham, the window it will occupy the whole screen.

Furthermore, Webix window features various methods for setting its **position**. All of them are described in a

separate documentation article.

Window Modality

Modality is a feature of the ui-component that can affect the work of the application and control user-app interaction. Alert and Confirm message boxes are modal windows in their essence, as they **prevent the workflow** on the application main window.

The appearance of such window on the page prevent users to work with the app until they perform action required by this window (enter data, press the button, etc..).

UI-related Window is non-modal by default. Its modality is set with the help of the same-name property:

```
{view:"window", modal:true}
```

When a modal window is inited, the app area is greyed out to indicate that it is disabled while you "communicate" with the window.

Related articles

- Popup
- Window and Popup Positioning
- Message Boxes
- Changing Visibility of Components
- Window CSS Image Map

Popup and Window Positioning

By default window and popup (provided that they don't occupy the whole screen) will be placed in the upper left corner of the screen sticked to browser borders. To avoid this, position should be set.

Absolute Positioning

Left and **top** offset can be defined right in the component constructor:

```
webix.ui({
    view:"window"
    width:300,
    height:200,
    top:100,
    left:100
}).show();
```

At the same time, position can be set dynamically, by passing these parameters into its **show();** function:

```
webix.ui({
    view:"window",
    ...
}). show({
    x:300, //left offset from the right side
    y:50 //top offset
});
```

RELATED SAMPLE: Window Offset

Another way is to use **setPosition()**:

```
$$('my_window').setPosition(100, 100); //left and top offset
```

Relative Positioning

Relative positioning is achived with the help of **position** property. It takes the following values:

- **center** - to place window or popup at the center of the screen;
- **top** - to place window or popup at the top of the screen with center horizontal alignment. Also, "top" positioning adds slide **animation**: the window will slide from the top of the screen when shown and slide back when closed.

```
webix.ui({
    view:"window",
    position:"center" //or "top"
```

```
    //..window config

})
```

RELATED SAMPLE: Window: Centralization

Positioning Relative to HTML Node

Show() method allows specifying an HTML node near which a window will be shown.

```
$$("window").show(node);
```

By default, such pattern places the window below the node. To redefine this, pass position object as well:

```
$$("window").show(node, {pos:"top"});
```

Possible position values here are **top**, **bottom** (default), **left** and **right**.

To shift a window in relations to the node, pass its vertical and horizontal positions together with pos:

```
$$("window").show(node, {pos:"top", x:10, y:10});
```

Setting Position via State Object

By default, position is set in two way depending on window size:

- If a window or popup is **non-sized**, it occupies the whole screen with *top*, *bottom*, *left* and *right* offsets being ;
- If a window or popup features custom size (**width** and **height** properties in its constructor), it is placed in the left upper corner of the screen with its *top* and *left* offsets being .

This default position can be **changed** or **amended** with the help of **state** object used in the function of the **position** property:

```
webix.ui({
    view:"window",
    head:"",
    body:{...},
    position:function(state){
        state.left = 20; //fixed values
        state.top = 20;
        state.width -=60; //relative values
        state.height +=60;
    }
});
```

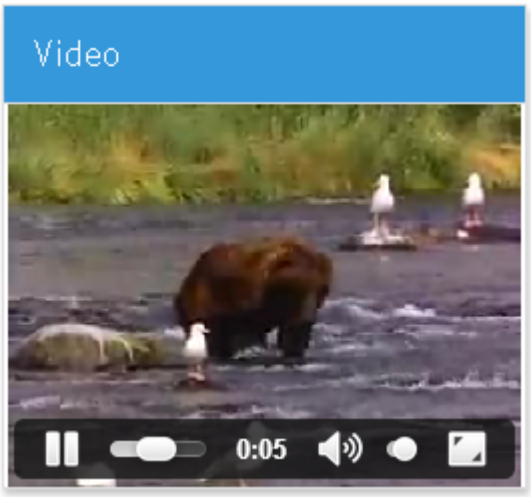
Video

API Reference

- Methods, properties and events
- Samples

Overview

Webix **video** is a UI component designed for embedding video files into the application. It is based on **HTML5 video tag** to provide support for most popular video formats: **MP4**, **WebM** and **Ogg**.



RELATED SAMPLE: Video view init

Initialization

The component inherits from Webix view and is initialized with the help of **webix.ui()** constructor.

```
//<div id="video_div" style="width:300px; height:200px;"></div>

webix.ui({
    view:"video",
    id:"video1", //used to work with the component later
    container:"video_div"
    src: ["data/movie.ogv","data/movie.mp4"]
});
```

Video source defined by **src** property can be either a single video file or an array of several ones. In case of **multiple src** browsers will use the first supported format, which ensures that your video will be recognized regardless of the client side browser.

Video component can be inited inside an HTML container (as above) as well as inside Webix window:

```
webix.ui({
    view:"window",
    head:"My Video",
    body:{
        view:"video",
        src:"..." //or [...]
    }
});
```

RELATED SAMPLE: Video view init

Customizing Video Player

Not much customization is possible, but still:

- **controls** property can be used to disable player controls by setting its value to *false*. By default controls are enabled (*true*);
- **autoplay** property can be used to start playing the video right after the component initialization. Default value is *false*.

```
view:"video",
src:[],
autoplay:true
```

RELATED SAMPLE: Autoplay

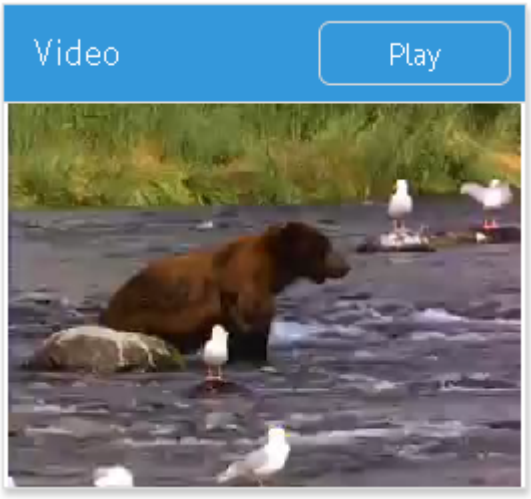
Working with Video

HTML5 video can be controlled programmatically using methods and properties of video DOM element:

- **play()** and **pause()** methods to control the video;
- **paused** - read-only property that stores video state.

To get the object of video DOM element, use the **getVideo()** method of Webix video component.

Let's get rid of prebuilt controls and assign **play-and-pause** function to a Webix button:



RELATED SAMPLE: Video API

UI - window with toolbar

```
webix.ui({
    view:"window",
    body:{
        view:"video",
        id: "video1",
        src:".."
        controls: false //removing default controls
    },
    head:{
        view:"toolbar", elements:[ //assigning a custom function to the
button
            {view: "button", id: "playButton", value: "Play", click:
playPause}
        ]
    },
}).show();
```

Logic

```
function playPause(){
    //getting video object
    var video = $$("video1").getVideo();

    //palying video if it's paused
    if (video.paused){
        video.play();
        $$("playButton").setValue("Pause")
    }
    else{
        //otherwise - pausing video
        video.pause();
        $$("playButton").setValue("Play")
    }
}
```

Alongside with playing and pausing the video, we change **button text value** with its setValue to correspond video state.

UI Extensions

Except for a main line of UI components, the library provides a number of helpful extensions to simplify your development. Most of the extensions are task-oriented and intended to accomplish a certain purpose. They are already quite configurable and need only initializing on the page to start serving.

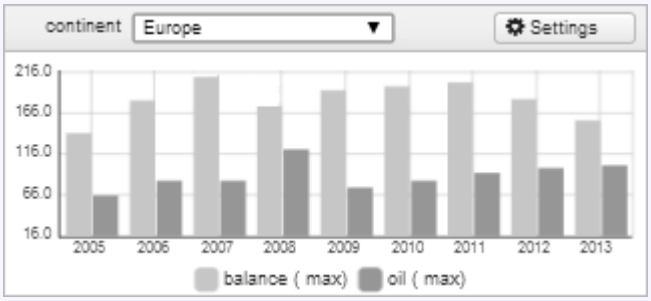
Pivot Table

A pivot table that helps you to summarize large numeric data sets in one spot.



Pivot Chart

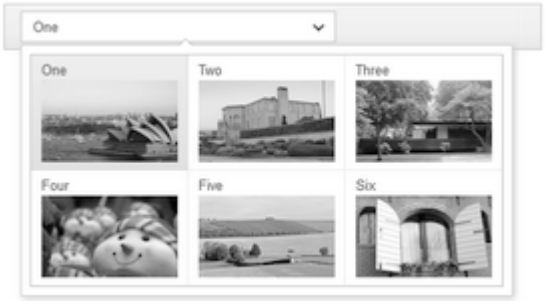
A pivot chart that helps you to research and analyze large quantities of data and the relationships between them.



Extended Select Controls

An integration extension that allows you to add extended options for select controls:

- Dataview Suggest
- Grid Suggest



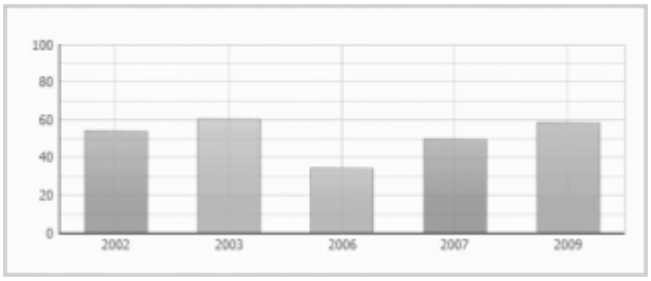
Map

An integration extension that allows you to add 3-rd party web maps to the app: *Google Maps, Yandex Maps, Here Maps*.



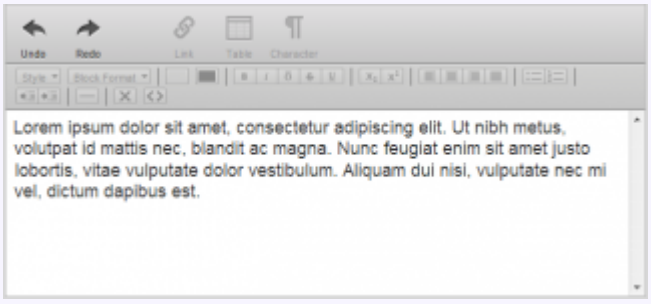
Chart

An integration extension that allows you to add 3rd-party charts to the app: *Raphael, D3*.



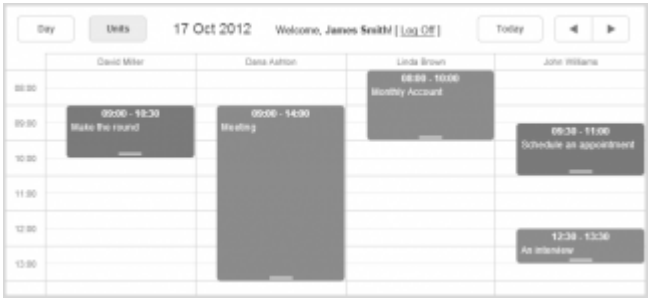
Text Editor

An integration extension that allows you to add 3rd-party editors to the app: *Mercury, TinyMCE, Code Mirror, NicEdit*.



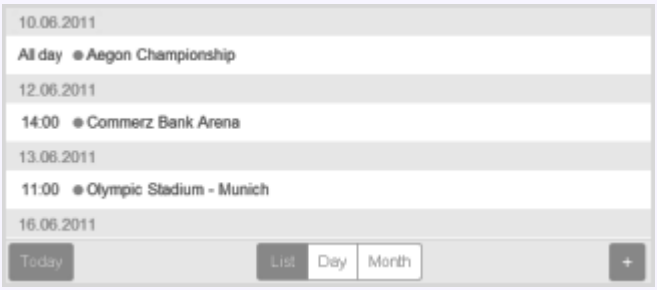
Scheduler

An integration extension that allows you to add a 3rd-party event calendar to the app: *dhtmlxScheduler of the DHTMLX library*.



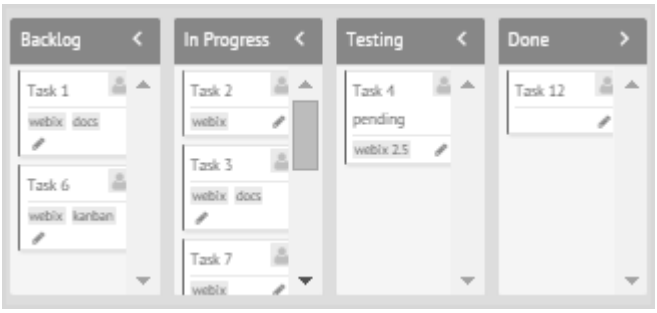
Mobile Scheduler

An integration extension that allows you to add the mobile version of the scheduler to the app.



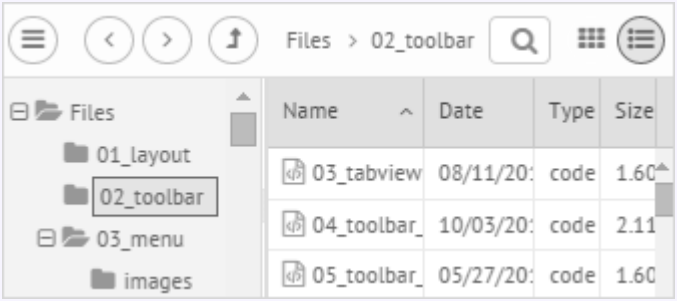
Kanban Board

An integration extension that helps you to visualize and optimize your workflow.



File Manager

An integration extension that helps you to navigate folders and files in a handy way.



Pivot Table

Webix Pivot is fully client-side Javascript datatable with **extended filtering capabilities** to allow for **well-organized visual reports** made over large and complex datasets at a high speed. With Pivot you can easily compare, filter and sort data within one table as well as change the analysis pattern on the flow.

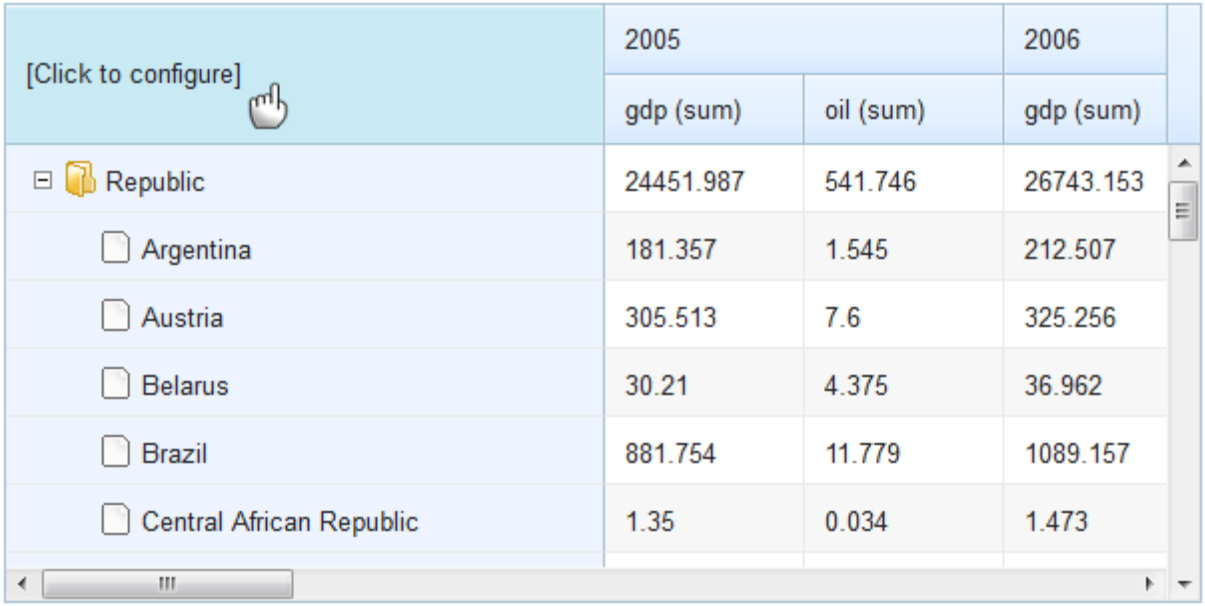
External resources

- You can download Webix Pivot from http://webix.com/pivot/
- There are online samples at docs.webix.com/samples

Design and Usage

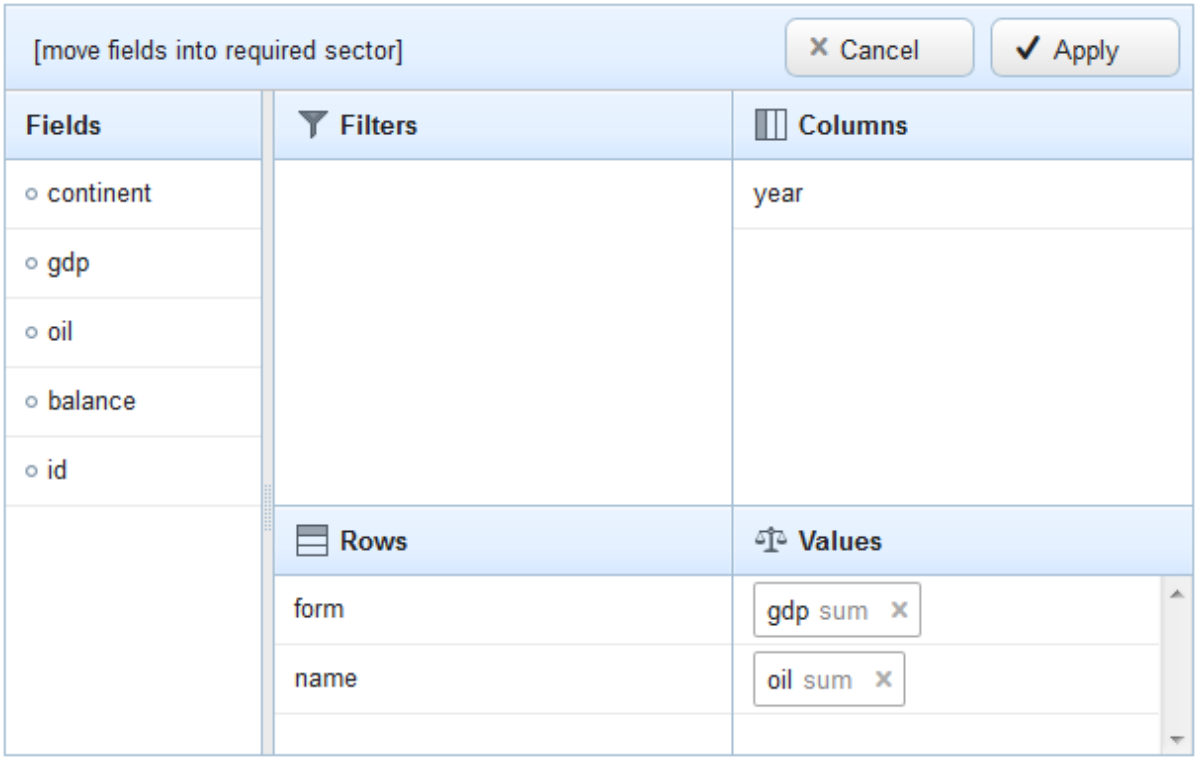
Webix Pivot consists of two functional parts:

- Webix **datatable** with configurable columns, rows and filters;



Pay attention to the **[click to configure]** area of the datatable upper left corner. It is highlighted each time you mouse over it while clicking this area will make a **configuration window** show up.

- Configuration **window**.



The window reflects current Pivot structure and contains five sections. The elements inside sections are based on the **data item properties** from the dataset and can be dragged to change Pivot structure:

```
Sample data

[
  {"name": "China", "year": 2005, "continent": "Asia", "form": "Republic",
  "gdp": 2256.919, "oil": 59.615, "balance": 134.098},
  ...
]
```

- **Fields** contain all data properties (such as *name*, *year*, *continent*, *form*, etc. from the sample dataset) EXCEPT for those in Rows, Columns or Filters.
- Fields dragged or initially set to **Columns** area define the **x-scale** of Pivot, namely the upper headers of the datatable;
- Fields dragged or initially set to **Rows** define the y-scale of Pivot, namely the treetable to the left;
- Fields dragged or initially set to **Values** define which data will be loaded as well as set the lowest row of datatable headers;
- Fields dragged or initially set to **Filters** define data properties that will be used as filters for the datatable.

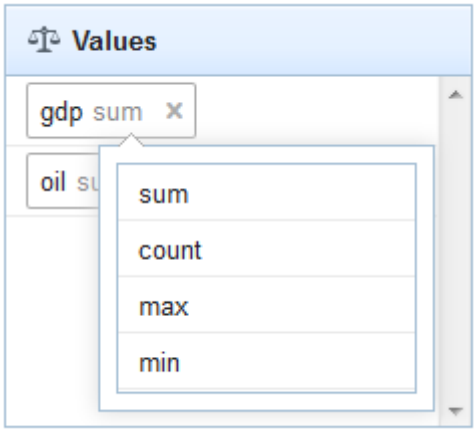
Balance amoung Sections

You cannot drag one and the same element to different sections; instead, use each element either for setting a **row**, or a **column** or for **filtering** or don't use it at all (it remains in **Fields**). However, **Values** section is more indendent since elements dragged to it are still available in

other sections except **Filters**.

Data Operations

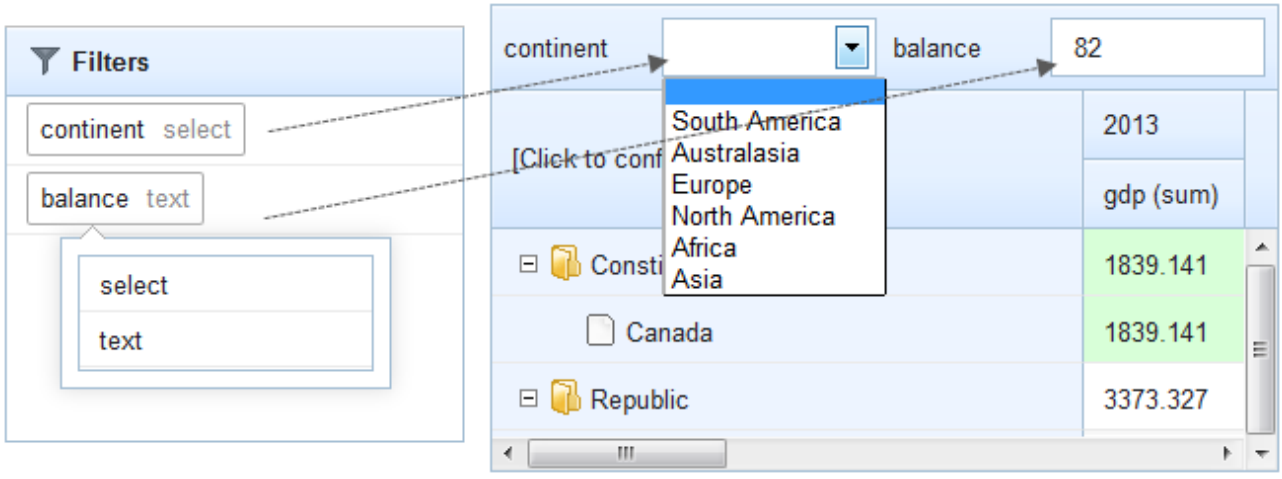
Current data operations come together with values they are assigned to in the **Values** section. Right there they can be changed with a couple of clicks.



- **Sum** - sums all values of this property and shows the sum;
- **Max** - picks and shows the max value of this property;
- **Min** - picks and shows the min value of this property;
- **Count** - counts the number of occurrencies of this property and shows it.

In addition to the above-mentioned prebuilt operation types, you can as well create a custom one.

Filtering Options



Webix Pivot uses two filters:

- **text** - filtering is performed by symbols you type in the text field. It supports base **math comparison operators**, so you can type something like "**<** 100", "**>** 2.5" or "**=** 12".

  If there is no operator, a filter will use text match for filtering;

- **select** - filtering is performed by options that are automatically gathered from the dataset (all unique values of this property). For instance, if you choose *Continents*, then all unique continent names from the dataset become options.

Initialization

Since Pivot tool comes as an add-on to Webix library you need to include both **Webix and Pivot scripts** and **CSS files** on your page to make everything work. Make sure you specify relative paths to these files:

```
<html>
 <head>
  <!-- js files -->
  <script src="codebase/webix.js" type="text/javascript"></script>
  <script src="codebase/pivot.js" type="text/javascript"></script>
  <!-- css files -->
  <link rel="stylesheet" href="codebase/webix.css" type="text/css"
charset="utf-8">
  <link rel="stylesheet" href="codebase/pivot.css" type="text/css"
charset="utf-8">
 </head>
 <body>
   <div id = "testA"></div>  <!-- containter for Pivot -->
 </body>
</html>
```

The initialization of Pivot doesn't differ from that of other Webix components and is done with **webix.ui()** constructor function where you enumerate all the necessary properties:

```
webix.ui({
    view: "pivot",
    container: "testA",
    id: "pivot",
    max: true,
    structure: {
        rows: ["form", "name"],
        columns: ["year"],
        values: [
            { name: "gdp", operation: "sum"},
            { name: "oil", operation: "sum"}
        ],
        filters: []
    }
});
```

Properties

- **id** - *string, number* - component ID, should be unique within a page. In Webix, any component is

referred to by its ID, **$$("id")**, to perform various operations with it;

- **container** - *string* - ID of an HTML container Pivot will be put to (optional);
- **fieldMap** - *object* - can be used to define custom labels for fields (optional). Described in detail below;
- **max** - *boolean* - enables or disables GREEN highlighting of a cell(s) with the the biggest value in a row (optional);
- **min** - *boolean* - enables or disables RED highlighting of a cell(s) with the least value in a row (optional);
- **data** or **url** - *string* - defines dataset (data loading is described in detail below);
- **structure** - *object* - defines initial display and analysis pattern;

Structure Object

- **rows** - array of fields that will be used as pivot's rows and displayed in the treetable on the left;
- **columns** - array of fields that will be used as pivot's columns and will be displayed above the datatable area;
- **values** - array of fields that will be used as pivot's data (will be displayed in datatable cells);
- **filters** - array of fields that will be used as filters and will be displayed above the datatable columns.

Loading Data

Pivot supports both inline and external (including serverside) data in any of the supported data types: **XML**, **JSON**, **JSArray** and **CSV**.

Inline Data

```
Inline Data (JSON)

var pivot_dataset = [
    {"name": "China", "year": 2005, "form": "Republic", "gdp": 181.357,
"oil": 1.545 },
    {"name": "China", "year": 2006, "form": "Republic", "gdp": 212.507,
"oil": 1.732 },
    //...
]
```

To load inline data during component init, make use of data property:

```
webix.ui({
    view: "pivot",
    id: "pivot",
    data: pivot_dataset
});
```

To load inline data after component init on some event, for instance, use the parse function:

```
$$("pivot").parse(pivot_dataset);
```

External Data

Either you get data from an external file or by a serverside script, you should use the following pattern

If you load the data during component init, specify the path to this file/script as value of url

```
{
    view: "pivot",
    url: "../load.php" // or "../data.json"
}
```

If you load data after the component has been inited (e.g. on some event), use the load function:

```
$$("pivot").load("../load.php");
//or
$$("pivot").load("../data.json");
```

In essence, Pivot complies to standard Webix Data Loading rules.

Configuring Pivot

Defining Operation on Data

Operations are set within Pivot structure object in **values** array. **Name** refers to data item property:

```
view: "pivot",
structure: {
    values: [
        { name: "gdp", operation: "sum" }, //gdp values will be summed
        { name: "oil", operation: "max" } //max oil value will be shown
    ]
}
```

There are **4 prebuilt operations** over data:

- **sum** - default, shows the sum of values of this property;
- **max** - shows the max value of this property found in the dataset;
- **min** - shows the min value of this property found in the dataset;
- **count** - shows the number of occurencies of this property in the dataset.

If needed, you can **add your own operation**: {#operation}

```
pivot.operations.abssum = function(data) {
    // data - array of values which need to be processed
    var sum = ;
    for (var i = ; i < data.length; i++) {
        var num = window.parseFloat(data[i], 10);
        if (!window.isNaN(num))
```

```
        sum += Math.abs(num);
    }
    return sum;
};
```

And use it as:

```
values:[ name:"oil", operation:"abssum"]
```

Filters

Filters are set within Pivot structure object in **filters** array. **Name** refers to data item property used for filtering:

```
view:"pivot",
structure: {
    values: [
        { name: "name", type:"select" },
        { name: "continent", type: "text" }
    ]
}
```

There are two types of filters:

- **select** - filters by options automatically gathered from a dataset (all unique values of this property). For instance, if you choose *Continents*, then all unique continent names from the dataset become options;
- **text** - filters by symbols in the text field. It supports base **math comparison operators**, so you can type something like "< 100", "> 2.5" or "= 12"
  If there is no operator, filter will use text match for filtering.

Renaming fields

By default Pivot uses data item properties for column names. But if you have dataset structured like below, the default pattern becomes unsuitable:

```
[{ a1:100, a2:1.34 }, ...]
```

In this case, you can use **fieldMap** property to set beautiful names for columns instead of *a1* and *a2*:

```
webix.ui({
    view: "pivot",
    fieldMap: { "a1" : "GDP", "a2" : "Grow ratio" },
    //...
});
```

Setting readonly mode

In order to disable changing of the pivot configuration settings, you can specify the *readonly* config property with the "true" value:

```
webix.ui({
    view: "pivot",
    readonly:true,
    //...
});
```

RELATED SAMPLE: Readonly mode

Localization

By defaut all names and titles in Pivot are defined in English, but you can set any other language by specifying a custom locale for the page.

```
webix.i18n.pivot = {
    columns: "Columns",
    count: "count",
    fields: "Fields",
    filters: "Filters",
    max: "max",
    min: "min",
    operationNotDefined: "Operation is not defined",
    pivotMessage: "[Click to configure]",
    rows: "Rows",
    select: "select",
    sum: "sum",
    text: "text",
    values: "Values",
    windowTitle: "Pivot Configuration",
    windowMessage: "[move fields into required sector]"
};

//and then init Pivot and see you custom names
webix.ui({
    view:"pivot",
    //...
});
```

API

Getting and Setting Configuration Object

```
//get current configuration
```

```
var config = pivot.getStructure();

//set configuration
pivot.setStructure(config);
```

Format of a **config** object is the same as "structure" parameter of the constructor:

```
var config = {
    rows: ["form", "name"],
    columns: ["year"],
    values: [
        { name: "gdp", operation: "sum"},
        { name: "oil", operation: "sum"}
    ],
    filters:[]
}
```

Data export

You can export result to PDF or Excel:

```
$$("pivot").toPDF();
$$("pivot").toExcel();
```

Getting Datatable Object

You can access the DataTable object by using the next code:

```
var datatable = $$("pivot").$$("data");
```

This allows you use any of datatable events or API methods. At the same time, it's not recommended to redefine datatable properties.

```
//attach event to selection
datatable.attachEvent("onAfterSelect", function (id) {
    webix.message("selected row: "+id);
});

//or get the ID of the currently selected item.
var sel = datatable.getSelectedId();
```

Pivot Chart

Webix Pivot Chart is fully client-side Javascript tool for graphical representation of data. A plenty of configuration options allows users to use the tool for understanding, analyzing and researching large quantities of data and the relationships between them.

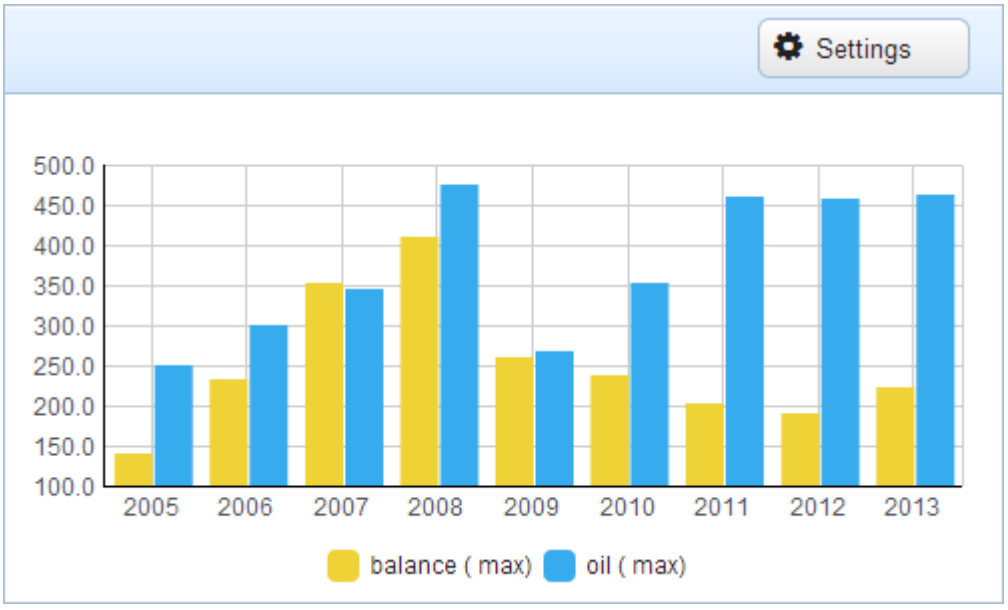
External resources

- You can download extensions from pivot.webix.com
- There are online samples at [docs.webix.com/samples](http://docs.webix.com/samples/61_pivot/index.html

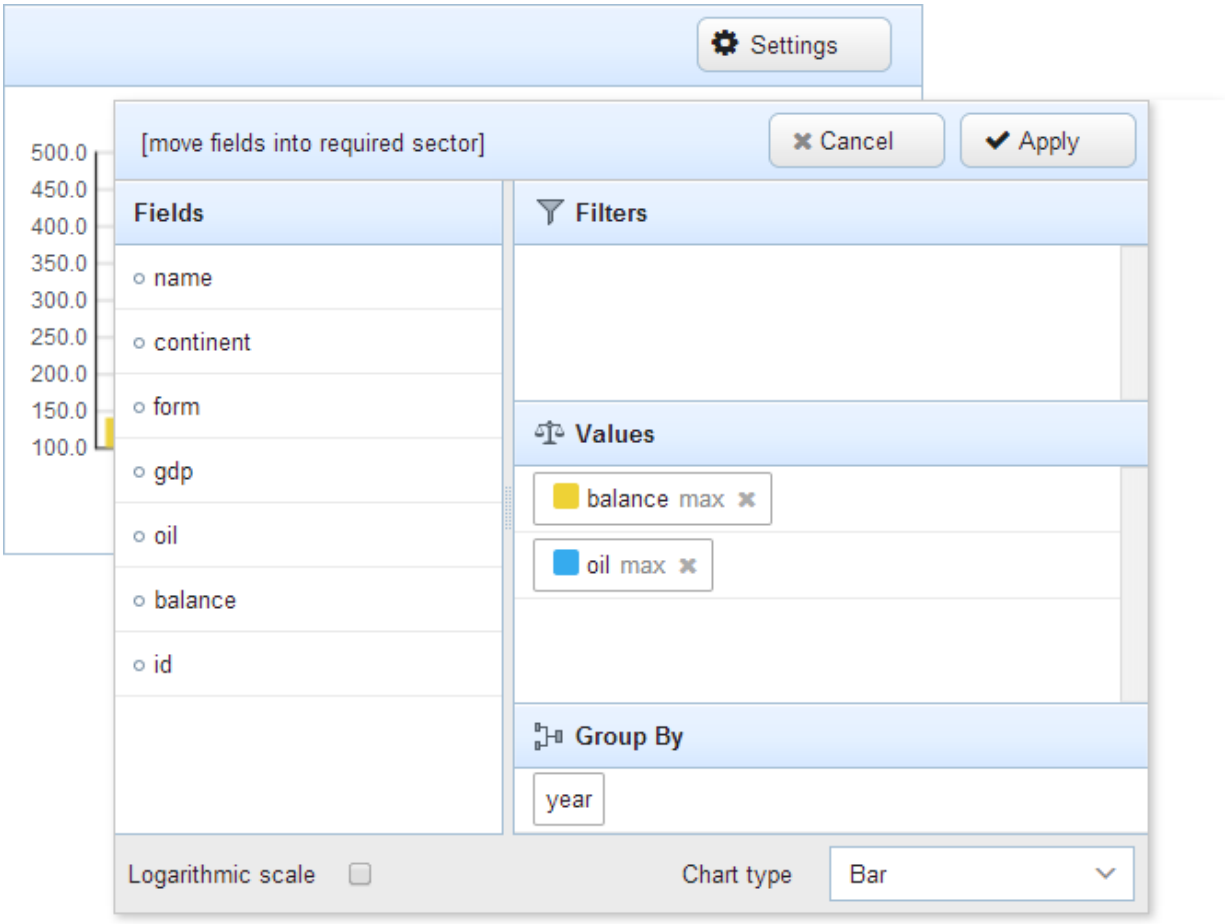
Design and Usage

Webix Pivot Chart consists of two functional parts:

Webix **chart** reflecting specified data properties and conditions:



Configuration window (appeared when the user clicks the 'Settings' button):



Configuration Window

The window reflects the current Pivot Chart structure and contains four sections. The elements inside sections are based on the data item's properties from the dataset and can be dragged to change Pivot Chart structure:

```
Sample data

[
    {"name": "China", "year": 2005, "continent": "Asia", "form":
"Republic",
    "gdp": 2256.919, "oil": 59.615, "balance": 134.098},
    //...
]
```

- **'Fields'** block presents all data properties of the dataset (such as name, year etc.) EXCEPT for those that are selected in the **Filters** and **Group By** blocks.
- Fields dragged or initially set in the **Values** block define which data will be loaded to the chart. Each data property in the **Values** block presents an individual graph (see details).
- A field dragged or initially set in the **Group By** block defines a data property that **Values**'s properties will be grouped by. For example, if the user sets the "balance" property as **Values**'s value (the 'sum' operation) and the "year" property as **Group by**'s value then the chart will display the total balance of all contries for each year. Note, only one data property can be specified in the **Group By** block.
- Fields dragged or initially set in the **Filters** block define data properties that will be used as filters for

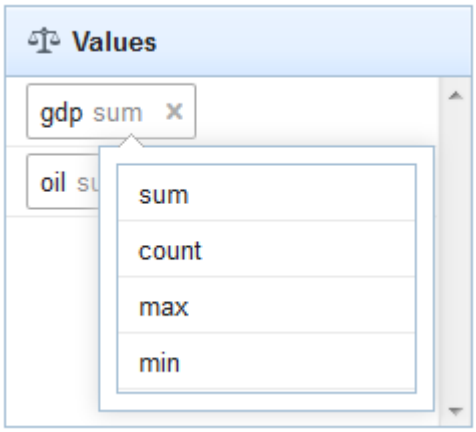
the chart. For each data property in the **Filters** block, Pivot Chart creates an individual input in the header of the chart (nearby the 'Settings' button) (see details).

Balance amoung Sections

You cannot drag one and the same element to different sections; instead, use each element either for **grouping** or for **filtering** or don't use it at all (it remains in **Fields**). However, **Values** section is more indendent since elements dragged to it are still available in other sections except **Filters**.

Data Operations

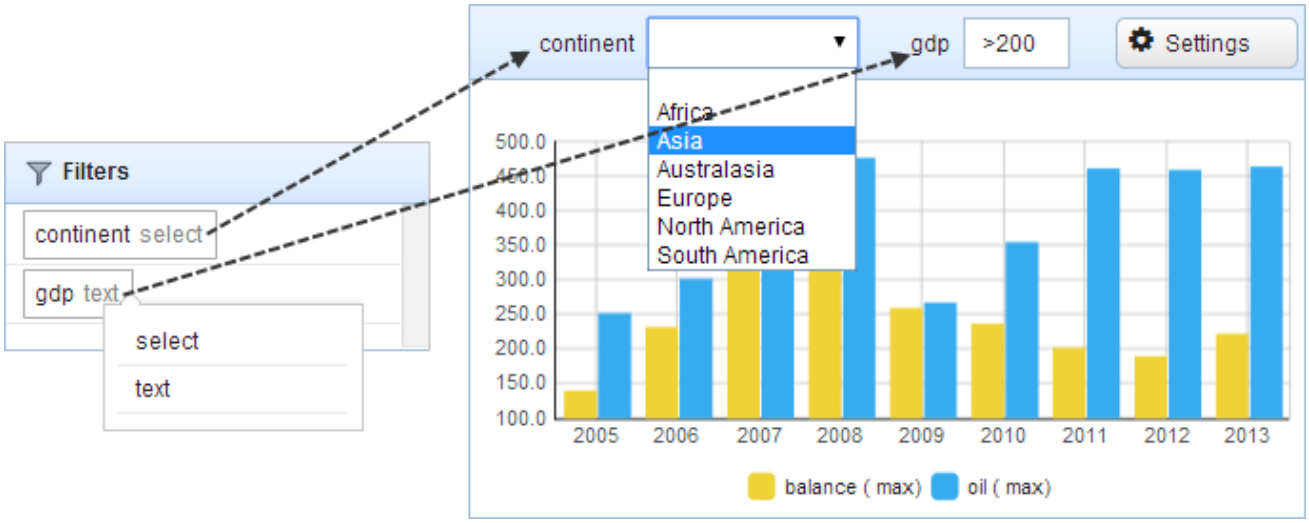
Current data operations come together with values they are assigned to in the **Values** section. Right there they can be changed with a couple of clicks.



- **Sum** - sums all values of this property and shows the sum;
- **Max** - picks and shows the max value of this property;
- **Min** - picks and shows the min value of this property;
- **Count** - counts the number of occurrencies of this property and shows it.

In addition to the above-mentioned prebuilt operation types, you can as well create a custom one.

Filtering Options

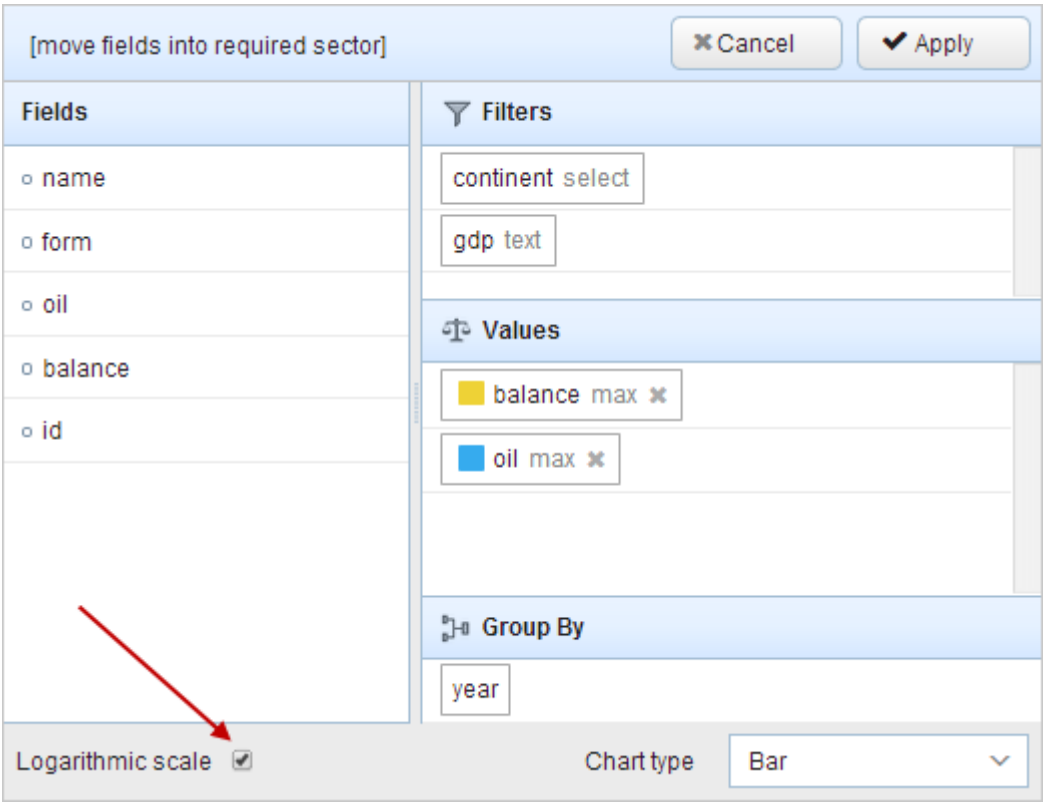


Webix Pivot Chart uses two filters:

- **text** - filtering is performed by symbols you type in the text field. It supports base **math comparison operators**, so you can type something like "**<** 100", "**>** 2.5" or "**=** 12".
  If there is no operator, a filter will use text match for filtering;
- **select** - filtering is performed by options that are automatically gathered from the dataset (all unique values of this property). For instance, if you choose *Continents*, then all unique continent names from the dataset become options.

Logarithmic scale

When difference between compared values is significant, smaller values can be hardly recognizable on the chart. To prevent it and make all values available for analysis, you can use logarithmic scale instead of default linear one.
To enable the logarithmic scale, check the "Logarithmic scale" checkbox in the configuration window.



Initialization

Since Pivot Chart tool comes as an add-on to Webix library you need to include both **Webix and Pivot Chart scripts** and **CSS files** on your page to make everything work. Make sure you specify relative paths to these files:

```
<html>
 <head>
  <!-- js files -->
  <script src="codebase/webix.js" type="text/javascript"></script>
  <script src="codebase/pivotchart.js" type="text/javascript"></script>
  <!-- css files -->
  <link rel="stylesheet" href="codebase/webix.css" type="text/css"
charset="utf-8">
  <link rel="stylesheet" href="codebase/pivotchart.css" type="text/css"
charset="utf-8">
 </head>
 <body>
   <div id = "testA"></div>  <!-- containter for Pivot Chart-->
 </body>
</html>
```

The initialization of Pivot Chart doesn't differ from that of other Webix components and is done with **webix.ui()** constructor function where you enumerate all the necessary properties:

```
webix.ui({
    container:"testA",
```

```
    id:"pivot",
    view:"pivot-chart",
    structure:{
        groupBy: "continent",
        values: [
            {name:"balance", operation:"max", color: "#eed236"},
            {name:"oil", operation:"max", color: "#36abee"}
        ],
        filters:[{name:"year", type:"select"}]
    },
    url: "../common/data.json"
});
```

Properties

- **id** - *string, number* - component ID, should be unique within a page. In Webix, any component is referred to by its ID, **$$("id")**, to perform various operations with it;
- **container** - *string* - ID of an HTML container Pivot Chart will be put to (optional);
- **fieldMap** - *object* - can be used to define custom labels for fields (optional) (details);
- **data** or **url** - *string* - defines dataset (details);
- **structure** - *object* - defines initial display and analysis pattern (details);
- **chart** - *object* - defines the chart configuration options that you want to redefine, i.e. object with a set of redefined properties (details);
- **chartType** - *string* - sets the default chart type;
- **chartMap** - *object* - can be used to redefine default types of a chart ('bar', 'line', 'radar') or add a new one ('stacked bar' or 'area') (details);
- **filterWidth** - *number* - sets the width of filters (both input and label) in the header of the chart;
- **filterLabelWidth** - *number* - sets the width of the filters' labels;
- **filterLabelAlign** - *string* - sets the horizontal alignment of the filters' labels;
- **editButtonWidth** - *number* - sets the width of the "Settings" button;
- **singleLegendItem** - *boolean* - specifies whether the legend should be displayed if it contains only 1 item;
- **palette** - *array* - specifies the pop-up color pallete of the **Values**' values `palette:[ ["#e33fc7","#a244ea",...],["#d3ee36","#eed236",.. ],.. ]` - each inner array specifies a row in the pallete

Structure Object

- **values** - array of fields that will be used as pivot's data (will be displayed in datatable cells);
- **filters** - array of fields that will be used as filters and will be displayed above the datatable columns;
- **groupBy** - the name of the field that will be used to group values by.

Loading Data

Pivot Chart supports both inline and external (including server-side) data in any of the supported data types: **XML**, **JSON**, **JSArray** and **CSV**.

Inline Data

```
Inline Data (JSON)

var pivot_dataset = [
    {"name": "China", "year": 2005, "form": "Republic", "gdp": 181.357,
"oil": 1.545},
    {"name": "China", "year": 2006, "form": "Republic", "gdp": 212.507,
"oil": 1.732},
    //...
]
```

To load inline data during component init, make use of data property:

```
webix.ui({
    view:"pivot-chart",
    id:"pivot",
    data:pivot_dataset
});
```

To load inline data after component init on some event, for instance, use the parse function:

```
$$("pivot").parse(pivot_dataset);
```

External Data

Either you get data from an external file or by a server-side script,use the following pattern:

- If you load the data during component init, specify the path to this file/script as value of url:

```
{
    view:"pivot-chart",
    url:"../load.php" // or "../data.json"
}
```

- If you load data after the component has been inited (e.g. on some event), use the load function:

```
$$("pivot").load("../load.php");
//or
$$("pivot").load("../data.json");
```

In essence, Pivot Chart complies to standard Webix Data Loading rules.

Configuring Pivot Chart

Defining chart properties

To redefine the default properties of the chart, such as a legend or width of bars, you can use the **chart** object. Any property that you specify in the **chart** object will redefine the analogous one in the default configuration object:

```
view:"pivot-chart",
chart: {
    scale: "logarithmic",
    barWidth: 25,
    legend: {
        layout: "x",
        align: "center",
        valign: "bottom"
    }
}
```

Note, you can use any property available for the Chart component (a list of properties).

Defining Operation on Data

Operations are set within Pivot Chart structure object in **values** array. **Name** refers to data item property, **color** - to the color of the related graph:

```
view:"pivot-chart",
id:"pivot",
structure:{
    values:[
        {name:"gdp",operation:"sum",color:"#eed236"},//gdp values will be
summed
        {name:"oil",operation:"max",color:"#36abee"}//max oil value will
be shown
    ]
}
```

There are **4 prebuilt operations** over data:

- **sum** - default, shows the sum of values of this property;
- **max** - shows the max value of this property found in the dataset;
- **min** - shows the min value of this property found in the dataset;
- **count** - shows the number of occurencies of this property in the dataset.

If needed, you can add your own operation:

```
view:"pivot-chart",
groupMethods:{
```

```
    abssum: function(template, data){
        data = data || this;
        var summ = ;
        for (var i = ; i < data.length; i++)
            summ+= Math.abs(template(data[i])*1);
        return summ;
    }
},
```

And use it as:

```
values:[ name:"oil", operation:"abssum"]
```

Filters

Filters are set within Pivot Chart structure object in **filters** array. **Name** refers to data item property used for filtering:

```
view:"pivot-chart",
structure:{
    values:[
        {name:"name",type:"select"},
        {name:"continent", type:"text"}
    ]
}
```

There are two types of filters:

- **select** - filters by options automatically gathered from a dataset (all unique values of this property). For instance, if you choose *Continents*, then all unique continent names from the dataset become options;
- **text** - filters by symbols in the text field. It supports base **math comparison operators**, so you can type something like "< 100", "> 2.5" or "= 12"
  If there is no operator, filter will use text match for filtering.

Renaming fields

By default Pivot Chart uses data item properties for column names. But if you have dataset structured like below, the default pattern becomes unsuitable:

```
[{ a1:100, a2:1.34 }, ...]
```

In this case, you can use the **fieldMap** property to set beautiful names for columns instead of *a1* and *a2*:

```
webix.ui({
    view:"pivot",
    fieldMap:{ "a1" : "GDP", "a2" : "Grow ratio" },
```

```
    //...
});
```

Adding chart types

By default, Pivot Chart gives users a possibility to present data in 3 types of a chart: 'bar', 'line', 'radar'. If you need you can redefine the default types or add a new one.

Note, just types 'stacked bar' and 'area' can be additionally added to the Pivot Chart.

To add a new chart's type to Pivot Chart, use the **chartMap** property:

```
Adding a new chart's type

view:"pivot-chart",
chartMap: {
    "Area Radar": function(color){ //adds a new chart type 'Area Radar'
        return {
            type: "radar",
            alpha: 0.4,
            disableItems: true,
            fill: color,
            line:{
                color: color,
                width:1
            }
        }
    },
    "Line": function(color){ //redefines the default 'Line' type
        type:"line",
        offset:false,
        preset:"plot"
    }
}
```

The chart's type is identified by its name ("Area Radar" and "Line" in the code above). The names of the default types you can check in the 'Chart type' control of the Configuration window.

Setting readonly mode

In order to disable changing of the Pivot Chart configuration settings, you can specify the *readonly* config property with the "true" value:

```
webix.ui({
    view: "pivot-chart",
    readonly:true,
    //...
```

```
});
```

RELATED SAMPLE: Read-Only

Localization

By defaut all names and titles in Pivot Chart have English names, but you can change it by setting a custom locale for the page.

```
webix.i18n.pivot = {
    apply: "Anwenden",
    bar: "Balken",
    cancel: "Abbrechen",
    groupBy: "Gruppieren nach",
    chartType: "Diagramm Typ",
    count: "zahl",
    fields: "Felder",
    filters: "Filtern",
    logScale: "Logarithmischen Skala",
    line: "Linien",
    max: "max",
    min: "min",
    operationNotDefined: "Operation ist nicht definiert",
    radar: "Netzdiagramm",
    select: "auswahl",
    settings: "Einstellungen",
    sum: "summe",
    text: "text",
    values: "Werte",
    valuesNotDefined: "Werte oder Gruppenfeld ist nicht definiert",
    windowMessage: "[ziehen Felder auf gewunschten Sektor]"
};


//and then init Pivot Chart and see you custom names
webix.ui({
    view:"pivot",
    //...
});
```

API

Getting and Setting Configuration Object

```
//get current configuration
var config = $$("pivot").getStructure();
```

```
//set configuration
$$("pivot").setStructure(config);
```

Format of a **config** object is the same as "structure" parameter of the constructor:

```
var config = {
    groupBy: "year",
    values: [{name:"balance", operation:"sum", color: "#eed236"}],
    filters:[{name:"name", type:"select"}]
}
```

Data export

You can export result to PDF or Excel:

```
$$("pivot").toPDF();
$$("pivot").toExcel();
```

Getting Chart Object

You can access the Chart object by using the next code:

```
var chart = $$("pivot").$$("chart");
```

This allows you use any of Chart events or API methods.

Available only in **PRO Edition**

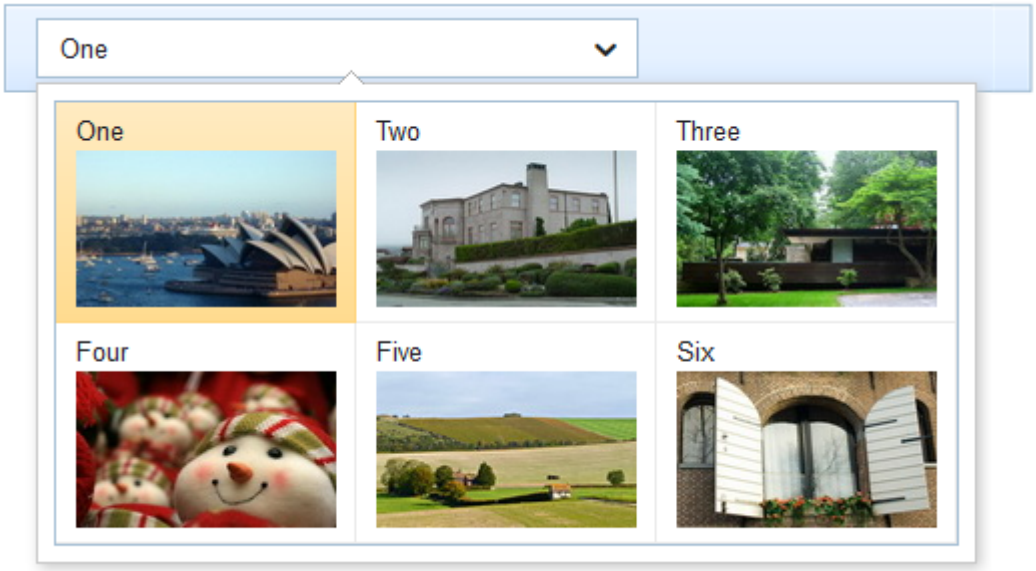
Dataview Suggest

Dataview suggest, or datasuggest, is used to add extra functionality to Webix select controls, namely richselect and combo and the same-name editors. It is based on Webix suggest list.

Datasuggest extension makes it possible to use Webix dataview in a dropdown menu which is great for inserting pictures while by default Webix list is used.

Using Datasuggest for Form Controls

The extension can be used to **redefine dropdown menu** of richselect and combo controls.



RELATED SAMPLE: Data Suggest

Default combo and richselect constructor looks as follows:

```
{
    view:"combo", //or "richselect"
    value:1,
    options:[...] //options data
}
```

When extended with datasuggest, the constructor allows setting Webix dataview for a dropdown menu and defining its **configuration** right among the component's **options**.

```
{ view:"richselect", value:1, options:{
    view:"datasuggest",
    data:options
```

```
}},
```

Possible configuration parameters:

- **data** (array) - defines data for the options to choose from;
- **template** (function) - sets a complex template for control input, for instance, the one with a picture. By default, the first data value after the ID field is shown;
- **textValue** (string) - sets data value to display in the input (acts as a string template) and, in case of combo, data value subject to filtering;
- **body** (object) - defines complex config for dataview.

```
Predefined Input Template

{ view:"richselect", options:{
    view:"datasuggest",
    template:"#value# (#color#)",
    data:options
}}
```

Body allows for setting the following parameters for a dropdown dataview:

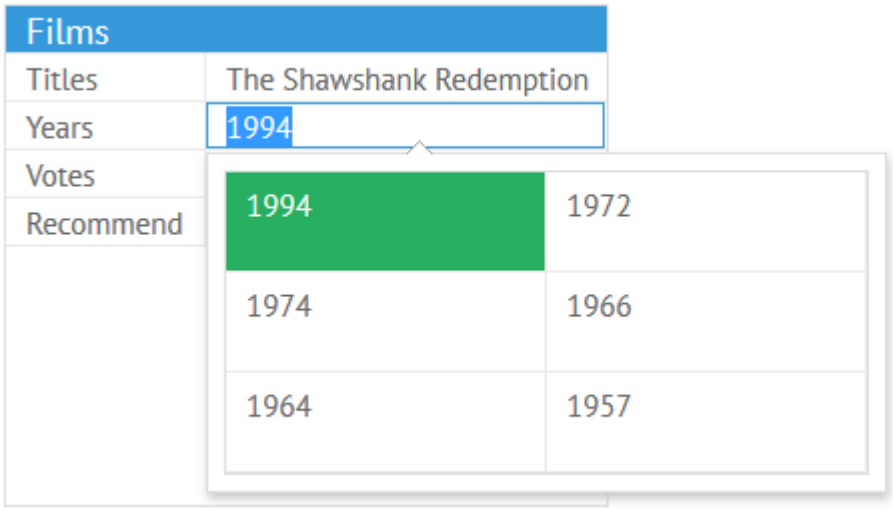
- **template** (string, function) - defines template for dataview items. By default, the first data value after the ID field is shown;;
- **type** (obj) - sets dimensions for dataview cells as an object *{width:150, height:100}*;
- **data** (array) - data for the options to choose from. If you use **body** property for datasuggest configuration, data should be defined inside it.
- **xCount** (number) - the number of items on the X scale;
- **yCount** (number) - the number of items on the Y scale;
- **autoheight** (boolean) - if *true*, set auto height for the dataview. *False* by default.

```
Item template, predefined item dimensions

{ view:"richselect", options:{
    view:"datasuggest",
    body:{
        template:function(obj){
            return obj.value + "</br><img
src='data/image00"+obj.id+".jpg'>";
        },
        type:{
            width:270, height:180
        },
    data:options
    }
}}
```

Using Datasuggest as Editor

Datasuggest can be used as a component editor as part of **combo** or **richselect** editors.



For these needs, a popup should be previously initialized, either with default or with customized settings:

```
Default - that's enough

var popup = webix.ui({
    view:"datasuggest"
});
```

With the custom configuration all the above-mentioned settings can be used:

```
Customized

var popup = webix.ui({
    view:"datasuggest",
    body:{
        type:{
            height:100,
            width:160
        },
        template:"<b>#value#</b><br>#year#"
   }
});
```

Then, this popup should be attached to the column in question with the help of its **popup** property:

```
{ id:"title",  editor:"richselect", collection:options, popup:popup }
```

RELATED SAMPLE: Data Suggest::Editor

Related Articles

- Webix Data Loading Pattern
- Templates for Webix Data Components
- Data Editing

Available only in **PRO Edition**

Grid Suggest

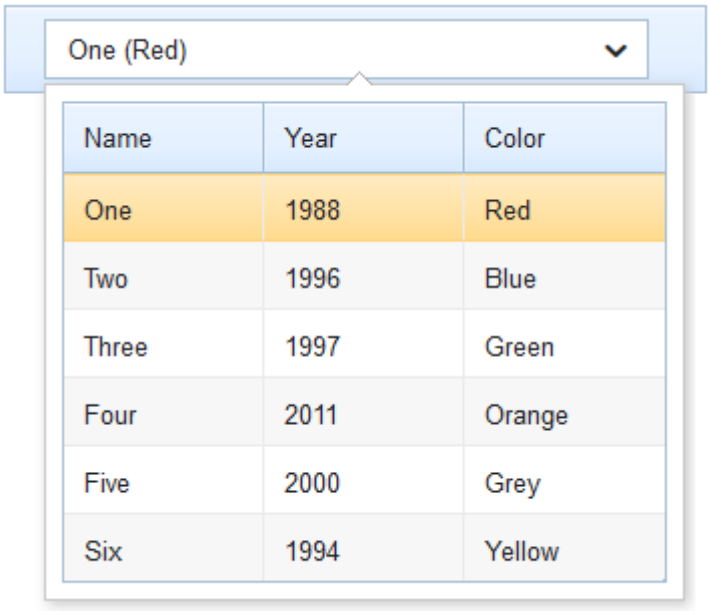
The functionality is a feature of **Webix Pro** edition.

The extension is based on Webix suggest list. It redefines the **default dropdown menu** for combo and richselect controls as well as the same-name editors.

Gridsuggest extension makes it possible to use Webix datatable in a dropdown menu while by default Webix list is used.

Using Gridsuggest for Form Controls

Gridsuggest can be used with combo and richselect controls.



RELATED SAMPLE: Grid Suggest

By default combo and richselect are inited with the following constructor:

```
{
    view:"combo", //or richselect
    value:1,
    options:[..]//option data
}
```

With gridsuggest extension you get the possibility to define view used for a dropdown menu as well as set its

configuration, both for combo and richselect.

```
{ view:"richselect", value:1, options:{
    view:"gridsuggest",
    data:options
}}
```

Possible properties here are as follows:

- **view** (string) - name of the view acting as a dropdown menu (here - gridsuggest);
- **data** (array) - data for the options to choose from;
- **textValue** (string) - data value shown in the input (acts as a string template) and, in case of combo, data value used for filtering. By default, the first data value after the ID field is taken;
- **template** (function) - allows for setting a complex template for the input field;
- **width** (number) - width of a dropdown popup;
- **body** (object) - set of configuration options for a suggest datatable.

```
{ view:"richselect", options:{
    view:"gridsuggest",
    template:function(item){
        return item.name + " (" + item.color + ")";
    },
    //textColor:#name#
    data:options
}},
```

Such code will show a datatable with borders while the columns will be generated for all data values from the dataset except for the ID field.

Datatable can be further configured inside **body** object property and can take the following parameters:

- **autoConfig** (boolean) - if *true*, enables default configuration for datatable (described above);
- **columns** (array) - used instead of autoConfig, allows for setting an array of columns for the datatable.
- **header** (boolean) - if *false*, hides datatable header. *True* by default;
- **borderless** (boolean) - if *true*, removes borders around the datatable. *False* by default;
- **data** (array) - data for the options to choose from. If you use **body** property for gridsuggest configuration, data should be defined inside it.

```
Auto config, no headers, no borders

{ view:"combo", options:{
    view:"gridsuggest",
    body:{
        autoConfig:true,
        header:false,
        borderless:true,
        data: options
```

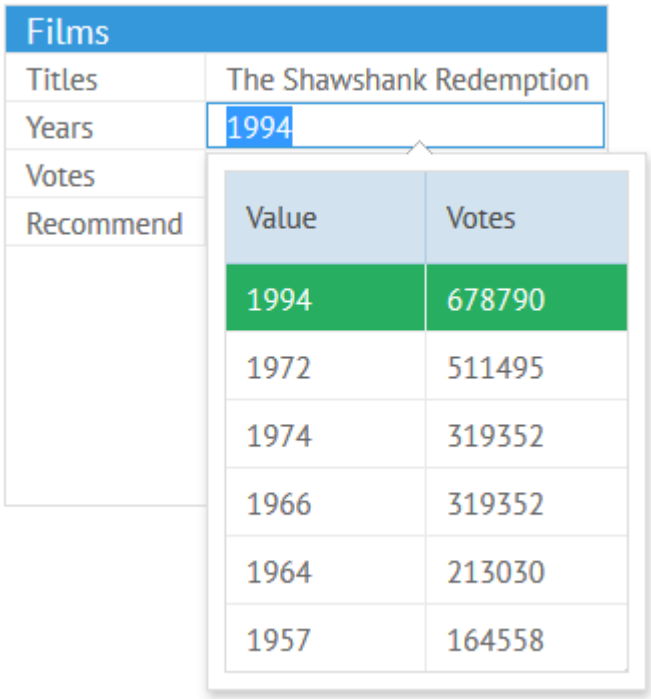
```
    }
}}
```

```
Preset Columns

{ view:"combo", options:{
    view:"gridsuggest",
    body:{
        columns:[
            { id:"name" }, { id:"color" }
        ],
        data:options
    }
}}
```

Using Gridsuggest as Editor

Gridsuggest can be used as a component editor as part of **combo** or **richselect** editor.



For these needs, a popup should be previously initialized, either with default or with customized settings:

```
Default - that's enough

var popup = webix.ui({
   view:"gridsuggest"
});
```

With the custom configuration all the above-mentioned settings can be used:

```
Customized

var popup = webix.ui({
```

```
    view:"gridsuggest",
    body:{
        columns:[
            {id:"value", header:"Value", width:250},
            {id:"year", header:"Year", width:100}
        ]
    }
});
```

Then, this popup should be attached to the column in question with the help of its **popup** property:

```
{ id:"title",  editor:"combo", collection:options, popup:popup }
```

RELATED SAMPLE: Grid Suggest::Editor

Enabling scrolling

In order to enable scrolling functionality in a gridsuggest extension, you should make 2 steps:

- Set the scroll property to true
- Disable the autoheight parameter

```
var form = { view:"form", elements: [
    { view:"combo", name:"names", label:"Code", options:{
        view:"gridsuggest",
        data: ["A", "B", "C", "D", "E", "F", "G", "H", "J"],
        body: {
            scroll:true,
            autoheight:false,
            autofocus:true,
            yCount:5
        }
    }}
  ]
};

webix.ui(form);
```

Related Articles

- Webix Data Loading Pattern
- Templates for Webix Data Components
- Data Editing

Intermediate

Text Editor Integration

JS files for these components aren't included in the lib package and should be taken from https://github.com/webix-hub/components

Webix library supports integration of popular text editors in the application with the help of its own components. For now, the following text editors can be included:

- Mercury;
- NicEdit;
- Tinymce;
- CodeMirror;
- CKEditor.

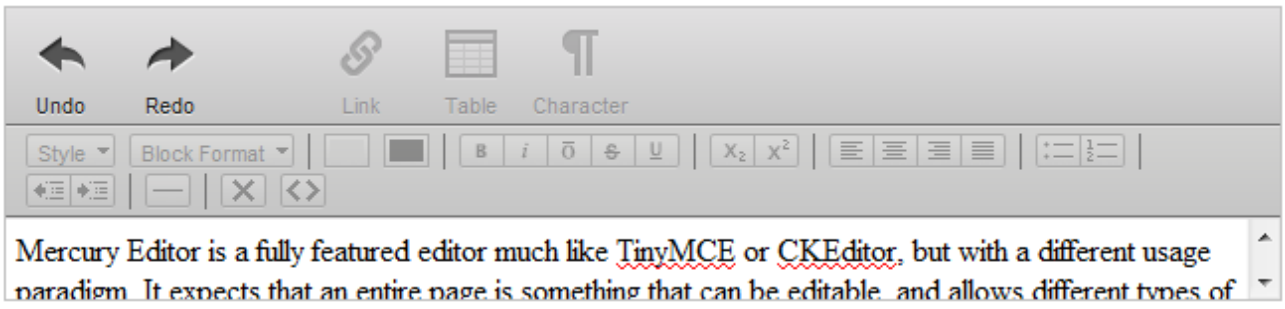
To embed any of the editors into your web page, you should link not only to Webix, but also to a **special JavaScript file** from the components folder. This file will connect you to the chosen editor as well load extra required files for it.

Note that in documentation samples files are linked in another way, but in your apps you should follow the patterns described below.

Mercury Text Editor

Mercury is a fully featured HTML5 editor with **full HTML, simple, markdown, snippet** and **image** areas to get the utmost from in-browser content editing.

Learn more about Mercury text editor



RELATED SAMPLE: Mercury text editor

Component Initialization:

```
Link to the library
```

```
<script type="text/javascript" src="./mercury.js"></script>
```

```
JS code

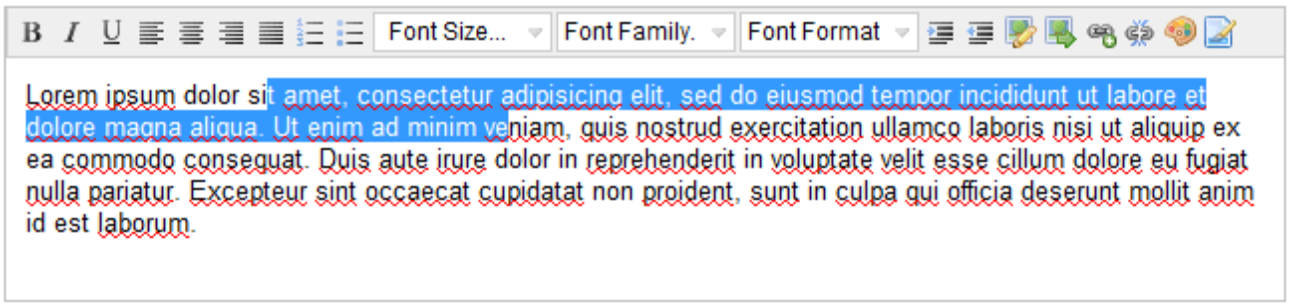
//path from which extra libraries are autoloaded
webix.codebase = "./";

webix.ui({
    id: "editor",
    view: "mercury-editor",
    value: "..." //text and HTML markup
});
```

NicEdit Text Editor

NicEdit is a simple and fast **WYSIWYG** editor for websites. Being extremely lightweight it makes any element/HTML container editable or converts standard textareas to rich text editing fields.

Learn more about NiceEdit text editor



RELATED SAMPLE: NiceEdit text editor

Component Initialization

```
Link to the library

<script type="text/javascript" src="./nicedit.js"></script>
```

```
JS code

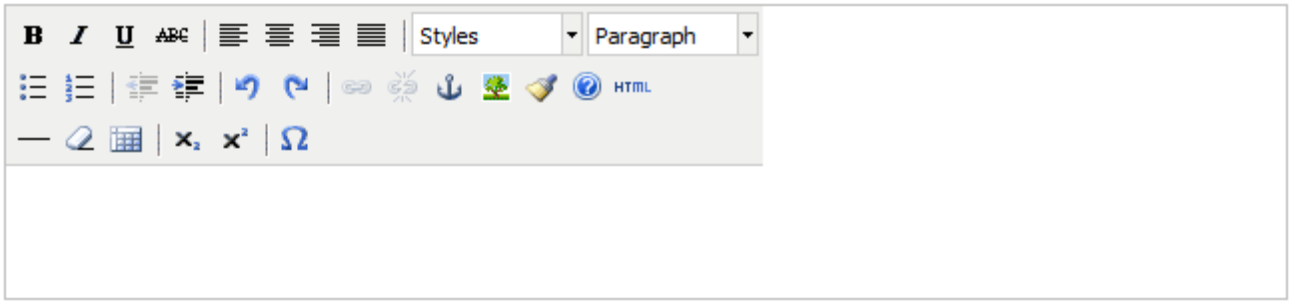
//path from which extra libraries are autoloaded
webix.codebase = "./";

webix.ui({
    id: "editor",
    view: "nic-editor",
    value: "..." //text and HTML markup
});
</script>
```

TinyMCE Text Editor

TinyMCE is a platform independent web based **Javascript HTML WYSIWYG** editor control. It has the ability to convert HTML TEXTAREA fields or other HTML elements to editor instances.

Learn more about TinyMCE text editor



RELATED SAMPLE: TinyMCE text editor

Component Initialization

```
Link to the library

<script type="text/javascript" src="./tinymce.js"></script>
```

```
JS code

//path from which extra libraries are autoloaded
webix.codebase = "./";

webix.ui({
    id: "editor",
    view: "tinymce-editor",
    value: "..." //text and HTML markup
});
</script>
```

Codemirror Text Editor

CodeMirror is a JavaScript component that provides a **code editor** in a browser. When a mode is available for the language you are coding in, it will color your code, and optionally help with indentation.

Learn more about Codemirror text editor

```
webix.ui({
        type:'space', rows:[
            { id:'editor', view:'codemirror-editor'},
            { height:250 },
            { id:'editor2', view:'codemirror-editor'}
        ]
});
```

RELATED SAMPLE: Codemirror text editor

Component Initialization

```
Link to the library

<script type="text/javascript" src="./codemirror.js"></script>
```

```
JS code

//path from which extra libraries are autoloaded
webix.codebase = "./";

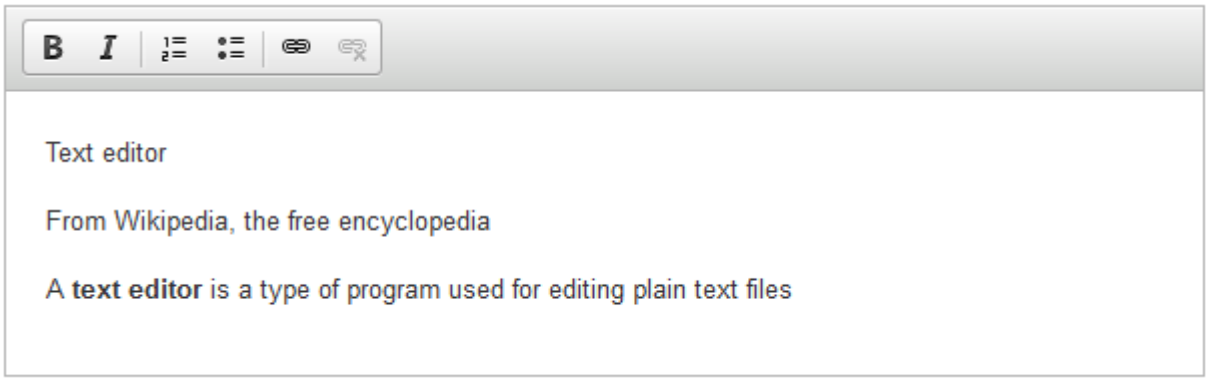
webix.ui({
    id: "editor",
    view: "codemirror-editor",
    mode: "javascript",
    value: "..." //text and HTML markup
});
```

Note that here an extra property appears - **mode** that denotes a language you are coding in.

CKEditor Text Editor

CKEditor is a ready-for-use WYSIWYG editor that brings word processor features directly to your web pages. It is suitable for editing plain texts as well as HTML markup.

Learn more about TinyMCE text editor



Related Sample: Ckeditor Text Editor

Component Initialization

```
Link to the library

<script type="text/javascript" src="./ckeditor.js"></script>
```

```
JS code

//path from which extra libraries are autoloaded
webix.codebase = "./";

webix.ui({
    id: "editor",
    view: "ckeditor",
    value: "..." //text and HTML markup
});
```

Common Methods:

- **setValue(**value**);** - sets text for the editor (equal to **value** property, but allows doing it dynamically). The function is called from the component referring to its ID.

```
$$("editor").setValue("<p>some text or code snippet</p>");
```

- **getValue();** - gets the current editor text with current formatting. Formatted text pieces are wrapped into corresponding HTML tags;

```
$$("editor").getValue(); // returns e.g "some<b> text</b> "
```

- **getEditor();** - gets the editor object so that you can use its specific API (exept for Mercury editor).

```
$$("editor").getEditor();
```

- **focus();** - in case of several editors on the page this method allows placing cursor mark into the desired editor.

```
$$("editor").focus();
```

Intermediate

Chart Integration

JS files for these components aren't included into the lib package and should be taken from https://github.com/webix-hub/components

Except for built-in charts, Webix offers a possibility to integrate with third-party charts of the following libraries:

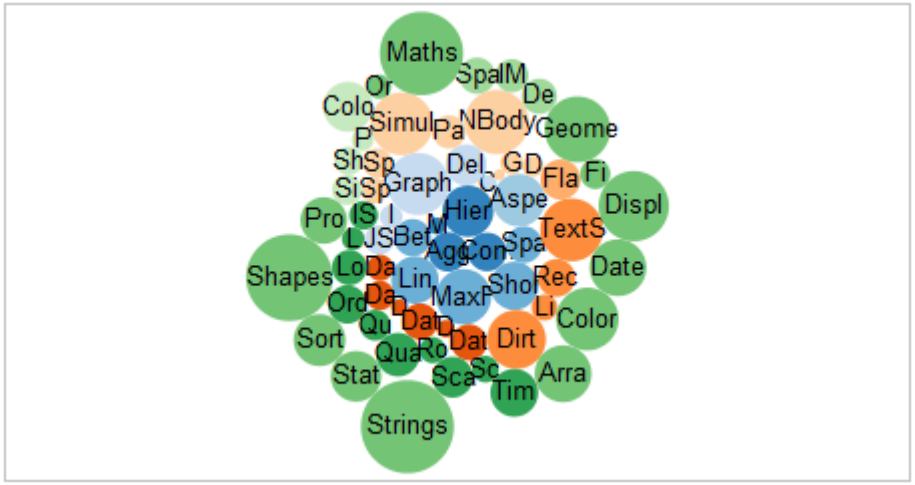
- D3 JS Library
- JastGage Javascript Plugin
- Raphael JavaScript Library
- SigmaJS Library

Note that third-party scripts are included to documentation samples in another way, but you should follow the linking pattern described below.

D3 Charts

D3 charts helps manipulate documents based on data and offers powerful visualization tools and a data-driven approach to DOM manipulation.

One of its implementations, **Bubble chart**, displays data in the area of circles that pack hundreds of values into a small space. It looks like a bundle of multicolored baloons.



RELATED SAMPLE: D3 Bubble Chart

Component Initialization

Firstly, you should include D3 JS file from the package into your document in addition to Webix files:

```
<script type="text/javascript" src="./d3.js"></script>
```

And then init the view:

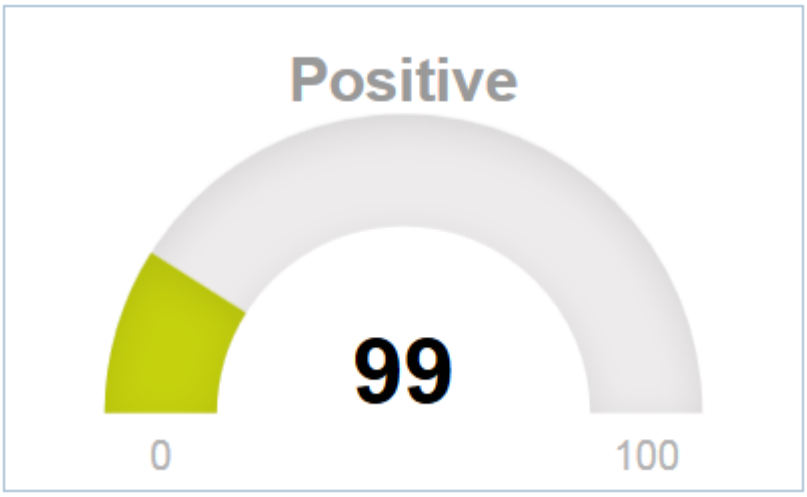
```
//path from which extra libraries are autoloaded
webix.codebase = "./";

webix.ui({
    view:"d3-chart",
    url:"...",
    ready:function(){...};
});
```

- **url** - specifies data source for the chart (data file or script);
- **ready** - function that is executed when the component is fully loaded. It contains the code of the needed chart (available on their site).

Justgage Gauges

JustGage JS is a handy JavaScript plugin for generating and animating nice gauges. It is based on Raphaël library for vector drawing.



Related Sample: JustGage Chart

Component Initialization

Firstly, you should include JustGage JS file from the package into your document in addition to Webix files:

```
<script type="text/javascript" src="./justgage.js"></script>
<!-- Raphael JS will be autoloaded -->
```

And then init the view:

```
//path from which extra libraries are autoloaded
webix.codebase = "./";

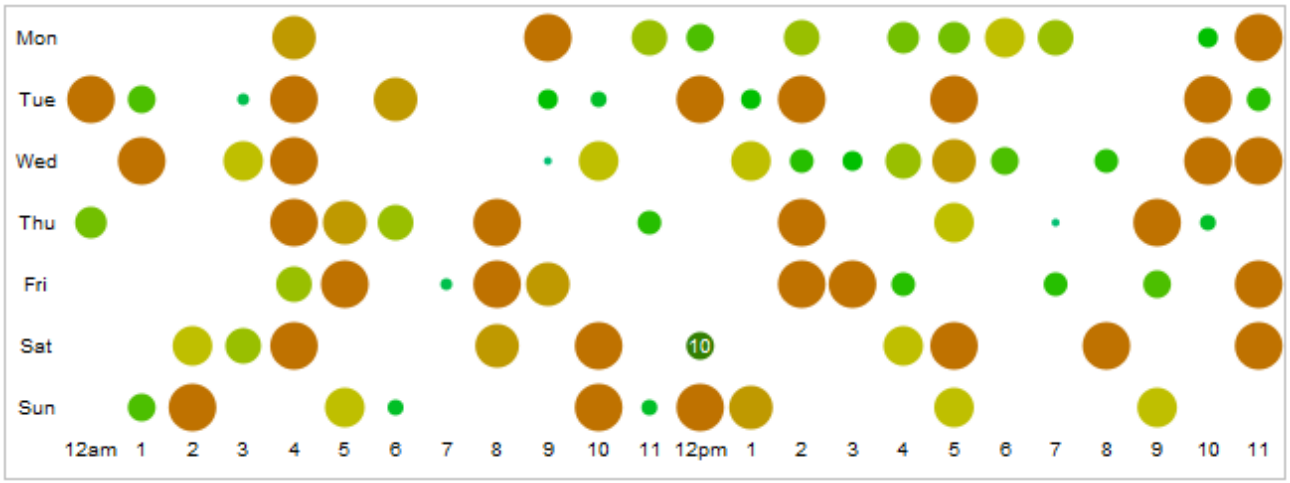
webix.ui({
    view:"justgage-chart",
    value:25,
    title:"Positive",
    height:300,
    min: ,
    max: 100
});
```

- Full list of a **Justgage gauge object properties** can be found on their site.

Raphael Charts

Raphael JS lib offers a number of viasualization tools that help working with with vector graphics on the web.

For instance, its dot chart encodes data in the area of circles placed on the coordinate system.



RELATED SAMPLE: Raphael Chart

Component Initialization

Firstly, you should include Raphael JS file from the package into your document in addition to Webix files:

```
<script type="text/javascript" src="./raphael.js"></script>
```

And then init the view:

```
//path from which extra libraries are autoloaded
webix.codebase = "./";

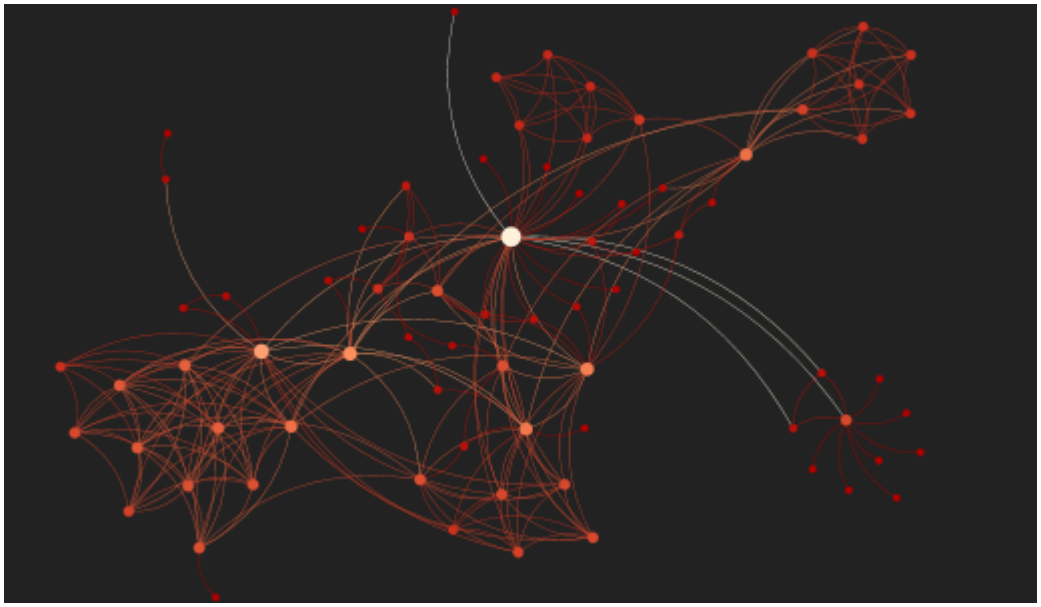
webix.ui({
    view:"raphael-chart",
```

```
    url:"...",
    ready:function(){...};
});
```

- **url** - specifies data source for the chart (data file or script);
- **ready** - function that is triggered when the component is fully loaded. It contains the chart's source code available on the Raphael library site.

Sigma Charts

Sigma is JS library that aids in drawing graphs using the HTML canvas element. It offers several graph types.



RELATED SAMPLE: Sigma Chart

Component Initialization

Firstly, you should include Sigma JS files from the package into your document in addition to Webix files:

```
<script type="text/javascript" src="./sigma.js"></script>
```

And then init the view:

```
//path from which extra libraries are autoloaded
webix.codebase = "./";

webix.ui({
    view:"sigma-chart",
    url:"./data/les_miserables.gexf"
});
```

- **URL** property specifies a path to a data file or script that will generate data.

The look-and-feel of a Sigma chart is defined by a currently used **plugin**. By default, Webix-integrated Sigma chart uses Fisheye plugin described in a tutorial on Sigma JS site. At the same time, you can easily change the plugin within the component source code in a private **_after_render()** function:

```
_render_once:function(){
    webix.require([
        "sigma/sigma.js", //sigma lib file
        "sigma/plugins/sigma.parseGexf.js", //GEXF extension
        "sigma/plugins/sigma.fisheye.js" //plugin
   ],function(first_init){ ...}); //chart initializing
}
```

In our Github package you'll also find **forceAtlas2** Sigma plugin as well as you can write your own one and place it to the plugin folder and include into the **sigma-chart** Webix component in the way described above.

In addition, the support for GEXF data is provided by default.

Advanced

Maps Integration

JS files for these components aren't included in the lib package and should be taken from https://github.com/webix-hub/components.

Webix library allows to embed geographical maps into the app with the help of its own components. Four maps are supported:

- **Google Map**
- **Nokia Map**
- **OpenStreet Map**
- **Yandex Map**

Ui-related maps inherit from view.

Initialization

1 . First of all, you need to **add a special file** to the head section of your document. The script helps connect to thirdparty applications without linking to them directly:

The extensions are taken from https://github.com/webix-hub/components. Note that in documentation samples they are linked in another way, but you should follow the patterns below:

- For **Google Map**




```
<script type="text/javascript" src="./googlemap.js"></script>
```

RELATED SAMPLE: Google Map

- For **Nokia Map**




```
<script type="text/javascript" src="./nokiamap.js"></script>
<script src="http://js.api.here.com/se/2.5.3/jsl.js"
type="text/javascript" charset="utf-8"></script>
```

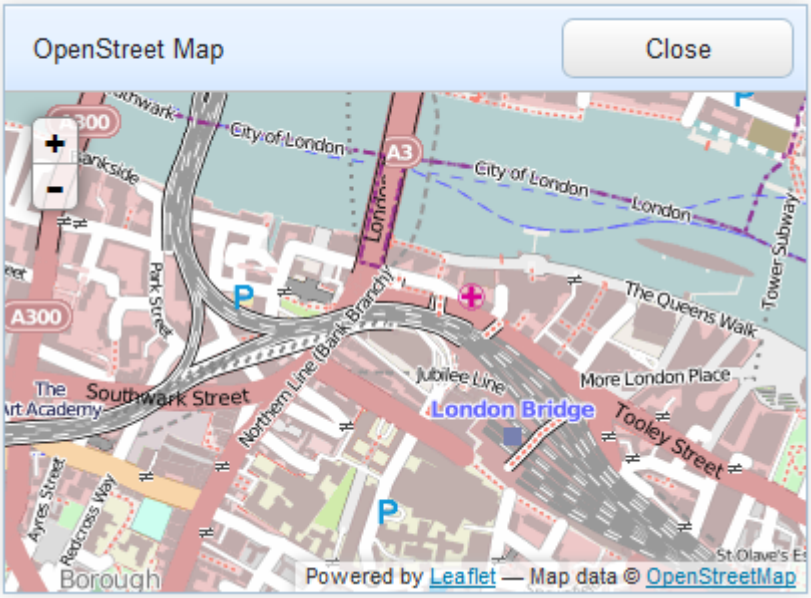
Additionally, you should have a Nokia Account registered. Once you register, they send you **"appID"** and **"authenticationToken"** keys that you need to specify:

```
webix.ready(function(){
    nokia.Settings.set( "appId", "..."); //instead of dots enter your
keys!
    nokia.Settings.set( "authenticationToken", "...");

    webix.ui({
        //then init map
    });
});
```

RELATED SAMPLE: Nokia Map

- For **OpenStreet Map**




```
<script type="text/javascript" src="./openmap.js"></script>
```

RELATED SAMPLE: OpenStreet Map

- For **Yandex Map**




```
<script type="text/javascript" src="./yandexmap.js"></script>
```

RELATED SAMPLE: Yandex Map

2 . Secondly, choose where on page the map will be inited. You can render it into any **HTML element** defined as the map's **container**:

```
// <div id="mydiv"></div>
webix.ui({
    container:"mydiv",
    ..//map config
});
```

Additionally, you can put the map into Webix Window:

```
webix.ui({
    view:"window", id:"mapwin",
    body:{ ... }, //map config
    head:{....}, //place for toolbar
    top:50, left:50,
    width:600, height:500 // dimensions and positioning of the window
}).show();
```

3 . Thirdly, **create a map** and assign properties to it according to your needs. All map constructors identical differing only in the map **view name**.

```
Map Initialization

{
    view:"yandex-map", // or "nokia-map", "google-map", "open-map"
    id:"map",
    zoom:6,
    center:[ 48.724, 8.215 ]
}
```

Map Properties

Map's properties define the initial position of the map:

- **id** - *string* - defines the component unique ID;
- **zoom** - *number* - defines how close you'd like to be to the Earth surface;
- **center** - *array* - sets the center of the map. It's an array of two elements (latitude and longtitude) with comma delimiter.

RELATED SAMPLE: Google Map

Working with Maps

To make online maps interactive you should refer to the API reference each of the services provides.

For instance, the GoogleMap API for javascript-based applications offers a rich colection of map options, controls, overlays, services, layers, etc. that you can use while working with a **google-map** intergrated into our library.

For instance, to show the necessary map piece on request, you can use the following function with the desired **lat**, **lng** and **zoom**:

```
function show_position(lat, lng, zoom) {
    var myLatlng = new google.maps.LatLng(lat, lng);
    $$('map').map.setOptions({
        zoom: zoom,
        center: myLatlng,
```

```
        mapTypeId: google.maps.MapTypeId.ROADMAP
    });
}
```

- **google.maps.map** class features lots of methods, one of them being **setOptions()** that allows for defining the area on the map;
- **google.maps.LatLng** class is a point on the map defined by latitude and longtitude coordinates;
- **mapTypeId** is a property of *google.maps.mapOptions* object, that allows setting a map type (HYBRID, ROADMAP, SATELLITE, TERRAIN).

Related Articles

- Setting Dimensions for the Components
- Redefinition of the Components

Intermediate

Scheduler and Gantt Integration

Both Scheduler and Gannt are nice planning tools from DHTMLX JS library. While Scheduler is an event calendar, the Gantt is a chart to display a long-term project schedule.

JS files for these components aren't included into Webix lib package and should be taken from https://github.com/webix-hub/components.

DHTMLX Scheduler

Scheduler is Javascript feature-rich event calendar that can create and display events as well as show a schedule for a day, week and month. Full info about it can be found in the dedicated section of DHTMLX site.



Scheduler Initialization

To include Scheduler into your app, you need to add a link to a dedicated library file into your document head section. Note that in a documentation sample this file is included in another way, but in your real app you should the pattern below:

```
<script type="text/javascript" src="./scheduler.js"></script>
```

And then init the view:

```
//path from which extra libraries are autoloaded
webix.codebase = "./";

webix.ui({
    view:"webix-scheduler",
```

```
    date:new Date(2010,,5),
    mode:"week",
    init:function(){...}, //scheduler config
    ready:function(){
        scheduler.parse("..events data..")
    }
});
```

RELATED SAMPLE: Scheduler

Scheduler properties

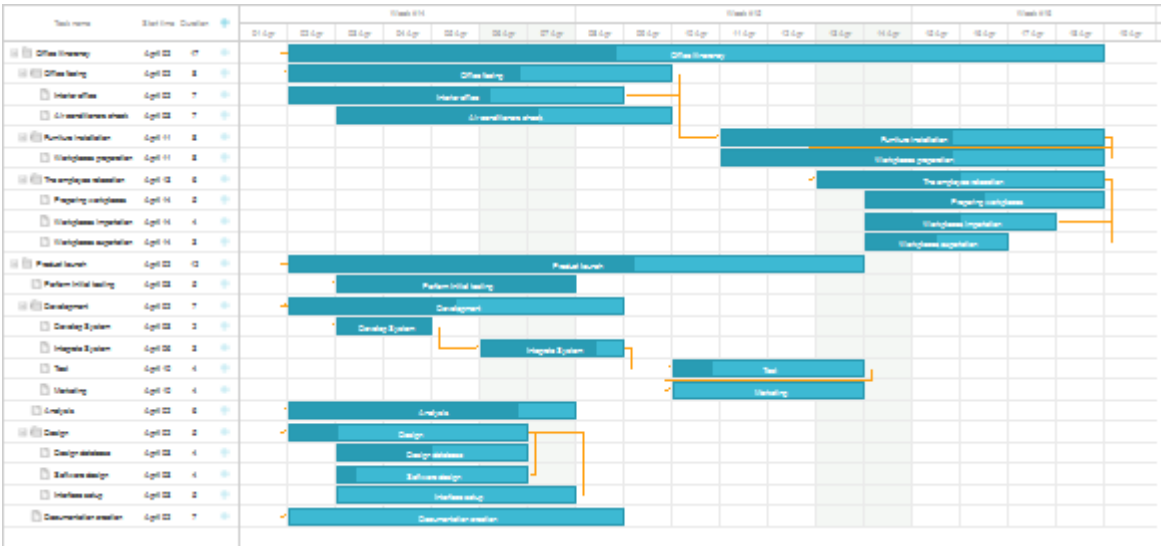
- **date** (function) - defines the start date of the scheduler. In the sample above, it is th 5the of January, 2010; If you pass no parameters into the *new Date()* constructor, the component will show current date;
- **mode** (string) - defines the type of presentation - day, week or month;
- **init** - specifies a function to be executed on component initializing. The property may include:
  - *scheduler.config.xml_date* - defines data format, e.g. "%Y-%m-%d %H:%i";
  - *scheduler.config.first_hour* (integer) - specifies an hour to start the day;
  - *scheduler.config.multi_day* (boolean) - allows/forbids creating events that last several days.
- **ready** - specifies a function to be executed when the component is fully loaded;
- **parse** (string) - XML string with predefined events.

Check Scheduler Documentation for more details.

Read more about date formatting methods in the corresponding chapter of the manual.

DHTMLX Gantt Chart

DHTMLX Gantt is a chart for visualizing project schedule that allows for easy setting and showing dependencies between tasks. It's highly customizable and features flexible API.
Full into about it can be found in the dedicated section of DHTMLX site.



Gantt Initialization

To include Scheduler into your app, you need to add a link to a dedicated library file into your document head section.

```
<script type="text/javascript" src="./gantt.js"></script>
```

And then init the view:

```
webix.ui({
    view:"dhx-gantt",
    init:function(){...},
    ready:function(){
        gantt.parse(tasks); //adding tasks
    }
});
```

Related Sample: DHTMLX Gantt

Gantt Properties

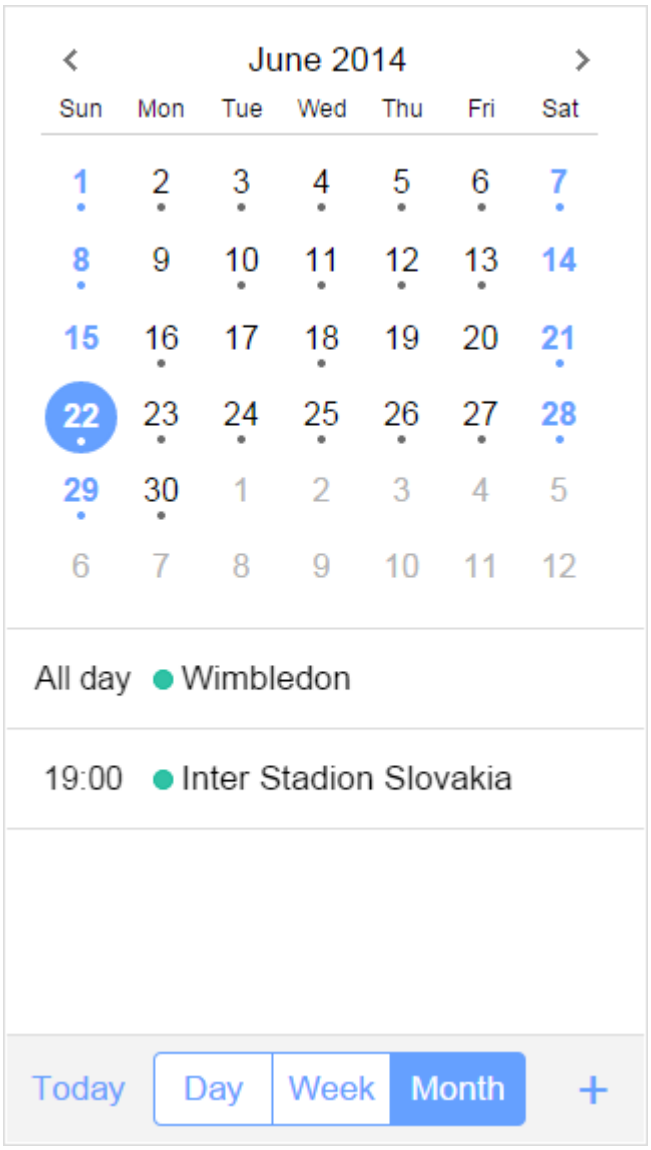
- **init** - functions to execute on initialization;
- **ready** - specifies a function to be executed when the component is fully loaded.

Related Articles

- Text Editor Integration
- Chart Integration
- Integration with Geographical Maps

Mobile Scheduler

Webix Scheduler was developed to be used in mobile web apps that run in Webkit-based browsers on mobile devices (iOS, Android, Blackberry). You can download Webix Scheduler by this link.



Initializating & Configuring Scheduler

- Creating Scheduler on a Page
- Loading and Saving Data
- Customizing "Edit" Form
- Specifying Templates
- Sizing Elements
- Date Formats
- Styling Scheduler

Managing Scheduler

- Localizing Scheduler
- Read-only Mode
- Recurring Events
- Dynamic Loading
- Changing Scheduler's Structure
- Handling Events

Other

- General How-Tos
- Tips for dhtmlxScheduler's users

Creating Scheduler

To use Scheduler in your application you should make 2 steps:

Include related CSS and JS files into your HTML file.

Call the object constructor.

Included files

You need to include 4 files. They are:

```
<!-- js files -->
<script src="codebase/webix.js" type="text/javascript"></script>
<script src="codebase/scheduler.js" type="text/javascript"></script>
<!-- css files -->
<link rel="stylesheet" href="codebase/webix.css" type="text/css"
charset="utf-8">
<link rel="stylesheet" href="codebase/scheduler.css" type="text/css"
charset="utf-8">
```

Initialization

To initialize the component you should use the following code:

```
//webix.ready() function ensures that the code will be executed when the
page is loaded
webix.ready(function(){
    //the method allows to adjust a main view to screen size
    webix.ui.fullScreen();
    //object constructor
    webix.ui({
    view: "scheduler",
        id: "scheduler"
    });
    //the method load() lets you populate the event calendar with data
    $$("scheduler").load("../common/mobile.xml","scheduler");
});
```

RELATED SAMPLE: Initialization

Basic configuration properties

Here is a list of configuration properties with examples of usage. All of them have default values, but you can redefine them if you need.

- **scheduler.config.init_date** - (date object) the initial date of the scheduler (by default, the current date)

```
scheduler.config.init_date = new Date(2014,5,1);
```

- **scheduler.config.multi_day** - (boolean) enables rendering of multi-day events (true by default)

```
scheduler.config.multi_day = true;
```

- **scheduler.config.multi_day_limit** - (numeric) sets the maximum number of visible items in a multi-day list (3 by default)

```
scheduler.config.multi_day_limit = 4;
```

- **scheduler.config.server_utc** - (boolean) enables converting server-side dates from utc to a local time zone (and backward) while sending data to the server (false by default)

```
scheduler.config.server_utc = true;
```

- **scheduler.config.separate_short_events** - (boolean) allows preventing short events from overlapping (true by default)

```
scheduler.config.separate_short_events = false;
```

- **scheduler.config.recurring** - (boolean) enables the recurring view (false by default)

```
scheduler.config.recurring = true;
```

Loading and Saving Data

Mobile Scheduler supports data loading from JSON and XML data sources. By default, data is loaded into Scheduler in JSON format.

```
{
    id: 7,
    text: "some_text",
    start_date:"2014-05-11 00:00",
    end_date:"2014-05-18 00:00"
    ...
}
```

Each data item (event) should have a unique *id* property and the following obligatory parameters:

- text - {string} the item's (event's) title
- start_date - {string} the start date of the event
- end_date - {string} the end date of the event

XML formats

There are two possible xml data types:

- "xml" - event items are described as <item> tags and properties as tag attributes

```
$$("scheduler").load("events.php","xml");
```

Here is an example of "xml" data source:

```
<data>
<item id="2" start_date="2014-05-24 00:00:00"
       end_date="2014-06-08 00:00:00" text="French Open"></item>
<item id="3" start_date="2014-06-10 00:00:00"
       end_date="2014-06-13 00:00:00" text="Aegon Championship"></item>
<item id="4" start_date="2014-06-21 00:00:00"
       end_date="2014-07-05 00:00:00" text="Wimbledon"></item>
</data>
```

- "scheduler" - event items are described as <event> tags and properties as subtags

You can use "scheduler" data type (xml datasource), if you are going to use Connector for data loading and saving.

```
$$("scheduler").load("events.php","scheduler");
```

In case of "scheduler" data type, the xml has the following format:

```
<data>
    <event id="2">
        <start_date>
            <![CDATA[ 2014-05-24 00:00:00 ]]>
        </start_date>
        <end_date>
            <![CDATA[ 2014-06-08 00:00:00 ]]>
        </end_date>
        <text>
            <![CDATA[ French Open ]]>
        </text>
        <details>
            <![CDATA[ Philippe-Chatrier Court Paris, FRA ]]>
        </details>
    </event>
    <event id="3">
        <start_date>
            <![CDATA[ 2014-06-10 00:00:00 ]]>
        </start_date>
        <end_date>
            <![CDATA[ 2014-06-13 00:00:00 ]]>
        </end_date>
        <text>
            <![CDATA[ Aegon Championship ]]>
        </text>
        <details>
            <![CDATA[ The Queens Club London, ENG ]]>
        </details>
    </event>
</data>
```

RELATED SAMPLE: Initialization

If you have a big data source, you can enable Dynamic Loading and data will be loaded by parts: months or years.

Saving data

There are several ways of saving data in Webix. The details on saving data are given in the Data Saving article.

To save data in scheduler you can make use of DataProcessor and set the save property:

```
webix.ready(function(){
    webix.ui({
        view: "scheduler",
        id: "scheduler",
        save: ".../events.php"
```

```
    });
    $$("scheduler").load("events.php");
});
```

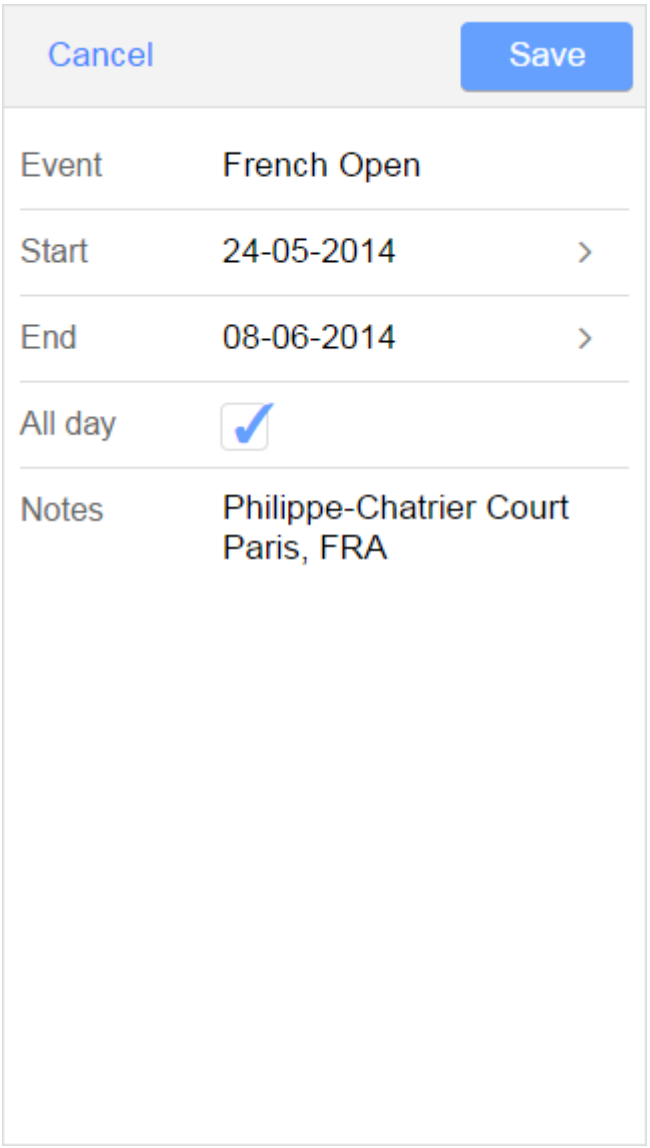
You can find more information in the chapter Saving Data with Connectors.

RELATED SAMPLE: Data saving

Or you can implement your own saving solution by using events.

Customizing the "Edit" Form

Whenever you need to edit events' details, make use of the "Edit" form.



For customizing the "Edit" form you should use the **'scheduler.config.form'** object. In this object you must specify all the controls you want the form to contain inside.

Into **'scheduler.config.form'** you can put any Form control (read about the controls in the related article Form Controls).

By default, **'scheduler.config.form'** looks like:

```
scheduler.config.form = [
    {
        view:"text",
        id:'text',
        label:scheduler.locale.labels.label_event,
        labelWidth: 90,
        name:'text'
```

```
    },
    {
        view:"datetext",
        id:'start_date',
        label:scheduler.locale.labels.label_start,
        name:'start_date',
        labelWidth: 90,
        dateFormat:scheduler.config.form_date
    },
    {
        view:"datetext",
        id:'end_date',
        label:scheduler.locale.labels.label_end,
        name:'end_date',
        dateFormat:scheduler.config.form_date
    },
    {
        view:"checkbox",
        id: "allDay",
        name: "allDay",
        label: scheduler.locale.labels.label_allday,
        labelWidth: 100,
        value:},
    },
    {
        view:"textarea",
        id: "details",
        name:"details",
        label: scheduler.locale.labels.label_details,
        labelWidth: 90,
        height:150
    }
];
```

There are 3 mandatory controls which you can't remove (you are also not allowed to change or remove their parameters' 'ids' as they're used in the scheduler code files):

- **text**
- **start_date**
- **end_date**

Two last default controls are optional and you can remove them if needed:

- **allDay**
- **details**

RELATED SAMPLE: Custom form

Specifying Templates

You can redefine any content in the scheduler in the following way:

```
scheduler.templates.day_event = function(obj,type){
    return obj.text;
}
```

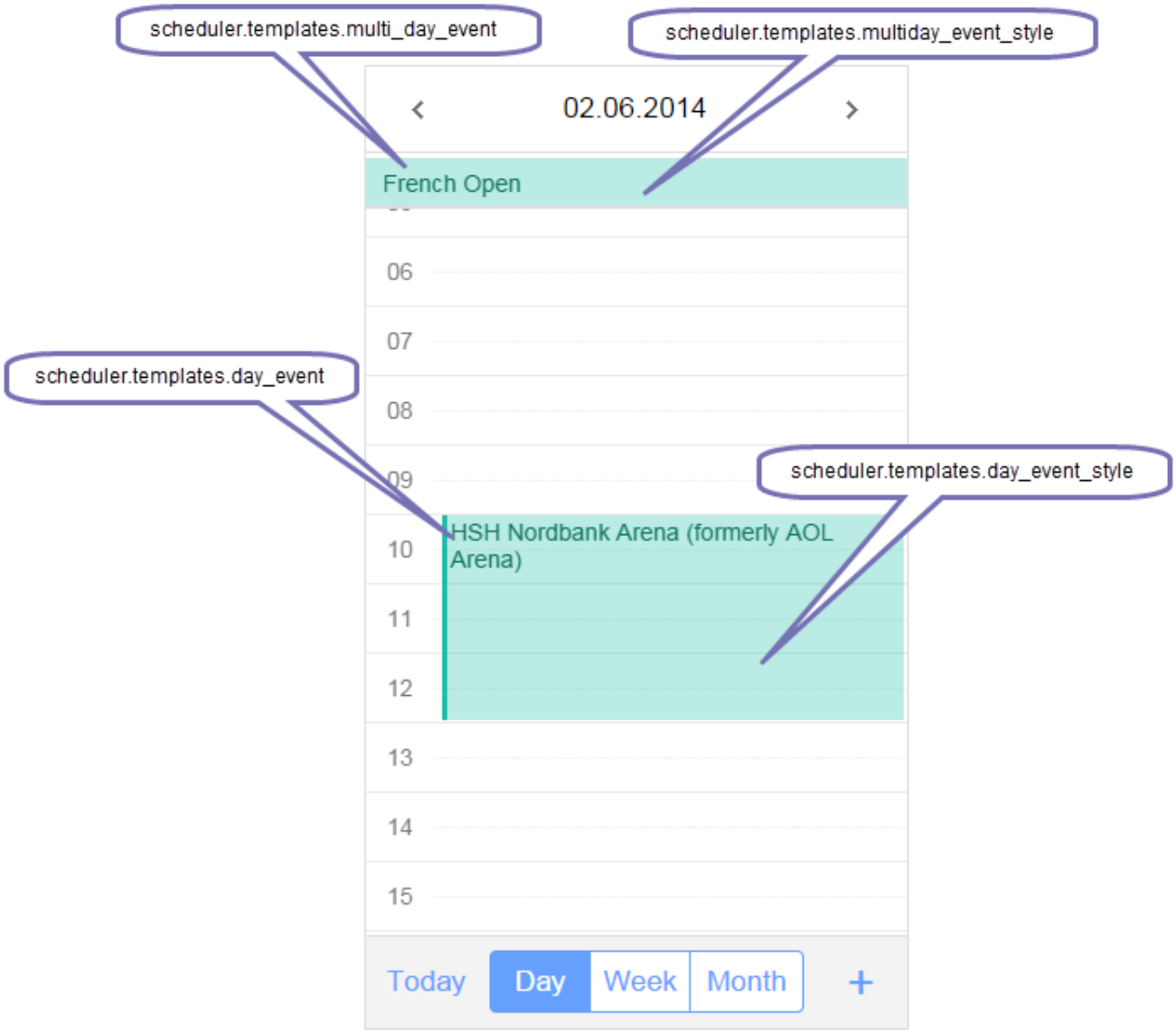
There is a set of templates that can be used to change the display of dates and titles.
Beware! The templates' definitions should go before the code line with scheduler initialization.

Below you can find the list of all available templates and their default definitions.

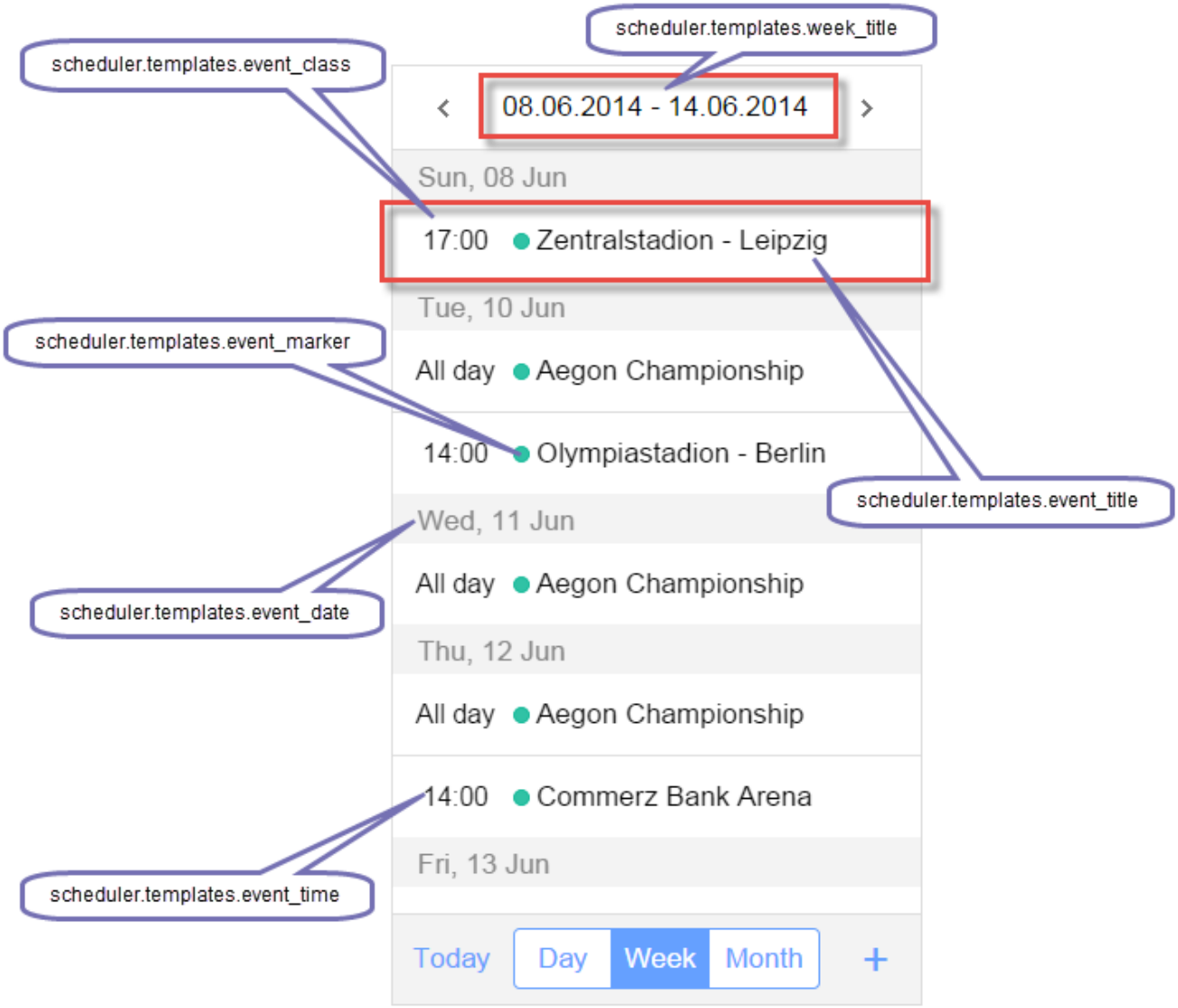
Available templates:

- scheduler.templates.day_event - specifies an event in the 'Day' view;
- scheduler.templates.day_event_style - specifies a css class for events in the 'Day' view;
- scheduler.templates.multi_day_event - the template for events in the multi-day list of the 'Day' view;
- scheduler.templates.multi_day_event_style - specifies a css class for multi-day events in the "Day" view;
- scheduler.template.week_title - specifies the title of the 'Week' view;
- scheduler.templates.event_class - specifies an additional css class that will be applied to the events in the lists of the 'Day','Week' and 'Month' views;
- scheduler.templates.event_date - the content of day headers in the "Week" view;
- scheduler.templates.event_marker - specifies a marker in the "Week" and "Month" views;
- scheduler.templates.event_time - the content of the time part of event items in the "Week" and "Month" views;
- scheduler.templates.event_title - specifies the content of an event item in the "Week" view;
- scheduler.templates.month_event_title - specifies the content of an event item in the "Month" view;
- scheduler.templates.calendar_event - the template for days with events in the "Month" view;
- scheduler.templates.new_event_data - specifies the default event properties. Used when a user creates new events;
- scheduler.templates.selected_event - the template for an event in the "Selected Event" view.

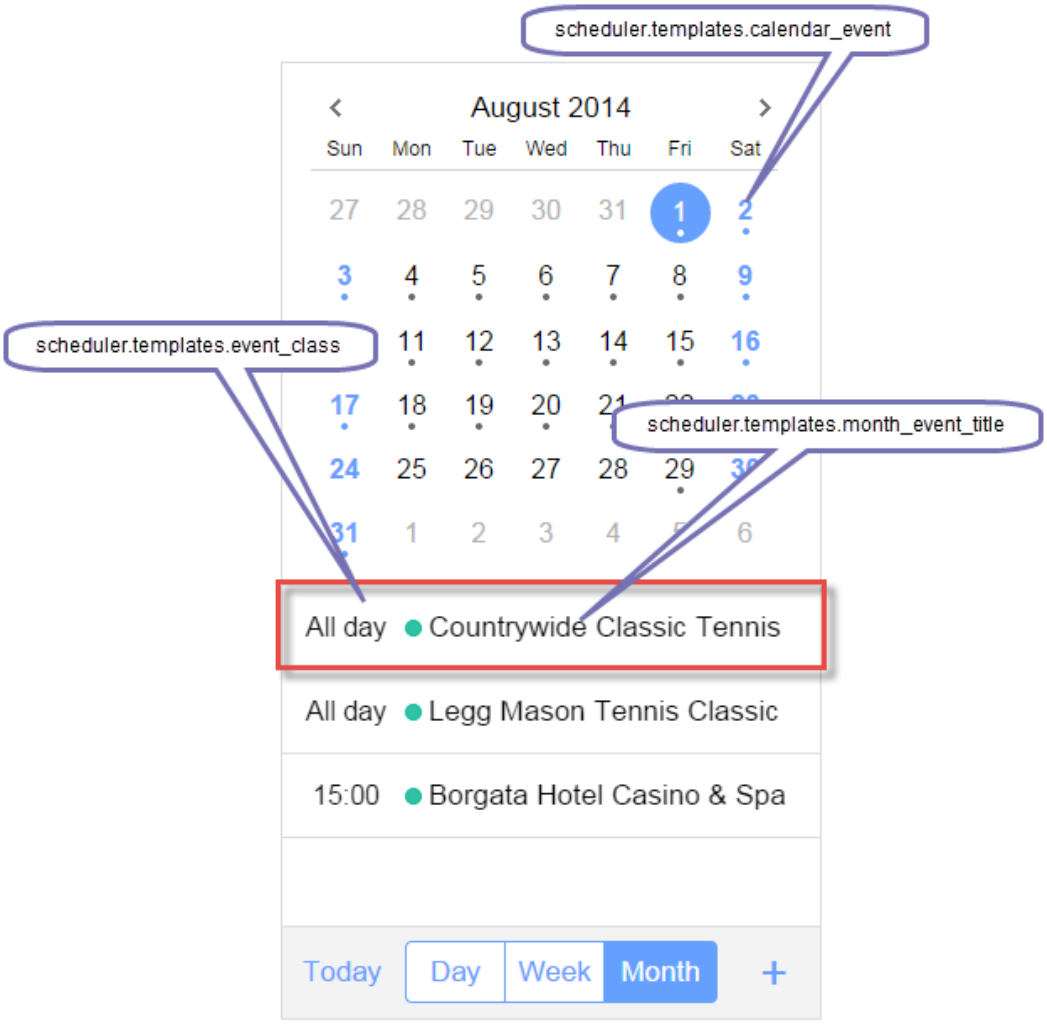
"Day" view


scheduler.templates.multi_day_event
scheduler.templates.multiday_event_style
02.06.2014
French Open
06
07
scheduler.templates.day_event
08
09
scheduler.templates.day_event_style
HSH Nordbank Arena (formerly AOL Arena)
10
11
12
13
14
15
Today
Day
Week
Month
+

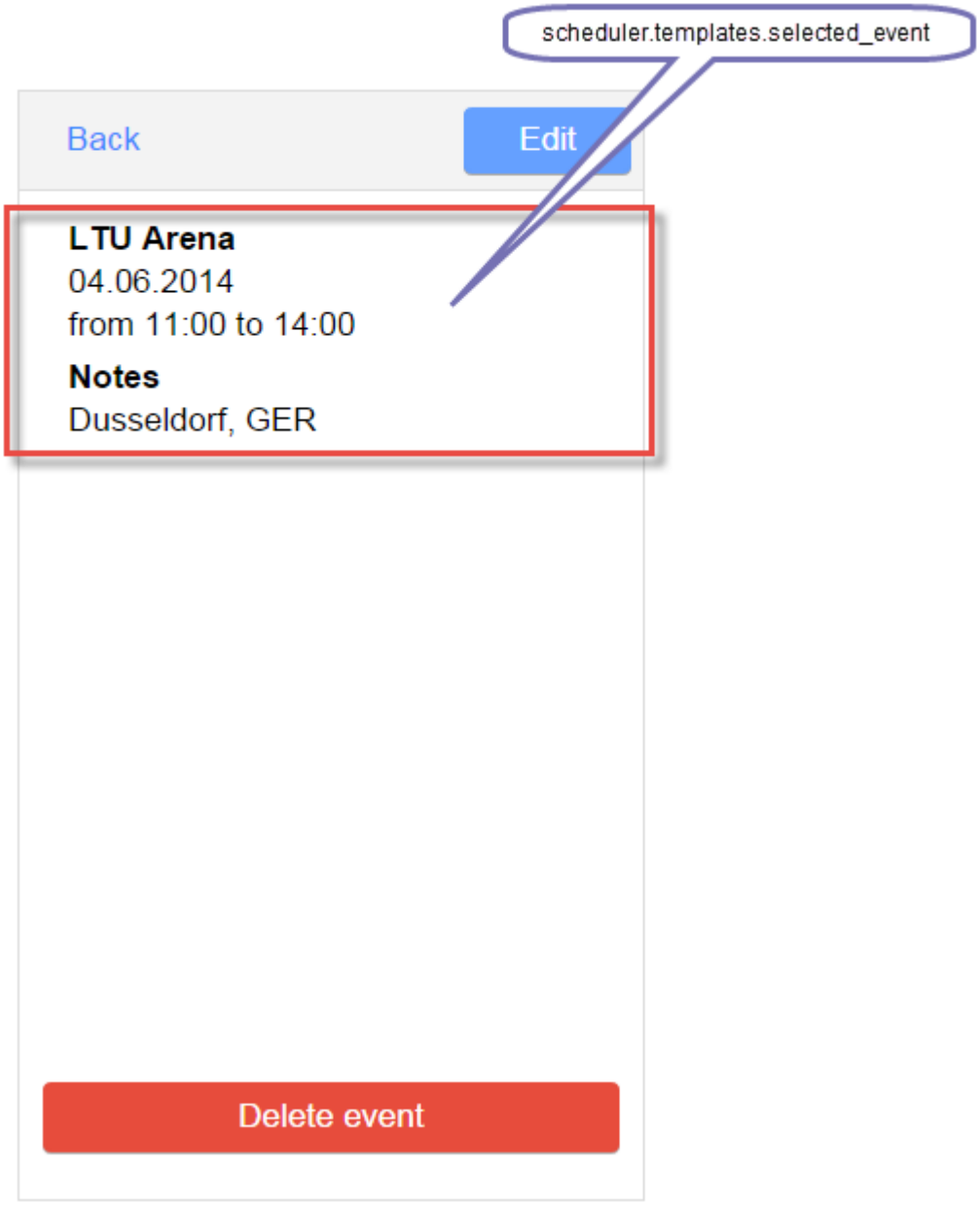
"Week" view


scheduler.templates.week_title
scheduler.templates.event_class
08.06.2014 - 14.06.2014
Sun, 08 Jun
17:00 Zentralstadion - Leipzig
Tue, 10 Jun
scheduler.templates.event_marker
All day Aegon Championship
14:00 Olympiastadion - Berlin
scheduler.templates.event_title
Wed, 11 Jun
scheduler.templates.event_date
All day Aegon Championship
Thu, 12 Jun
All day Aegon Championship
14:00 Commerz Bank Arena
Fri, 13 Jun
scheduler.templates.event_time
Today
Day
Week
Month
+

"Month" view



"Selected event" view



RELATED SAMPLE: Templates

scheduler.templates.day_event

Specifies an event in the 'Day' view.

Parameters:

- obj - the event object
- type - the object that specifies items list presentation.

  For details on the parameter see the API article "type"

```
scheduler.templates.day_event = function(obj,type){
    return obj.text;
}
```

scheduler.templates.multi_day_event

The template for events in the multi-day list of the "Day" view.

Parameters:

- obj - the event object
- type - the object that specifies items list presentation.

  For details on the parameter see the API article "type"

```
scheduler.templates.multi_day_event = function(obj,type){
    return obj.text;
}
```

scheduler.templates.day_event_style

Specifies a css class for events in the 'Day' view.

Parameters:

- obj - the event object
- type - the object that specifies items list presentation.

  For details on the parameter see the API article "type"

```
scheduler.templates.day_event_style = function(obj){
    var style = "";
    if(obj.color){
        var rgb = webix.color.toRgb(obj.color);
        style += ";border-color:"+obj.color+";";
        style += "background-color:rgba("+rgb.toString()+",0.3);";
        var hsv = webix.color.rgbToHsv(rgb[],rgb[1],rgb[2]);
        hsv[2] /= 1.6;
        var color = "rgb("+webix.color.hsvToRgb(hsv[],hsv[1],hsv[2])+")";
        style += "color:"+color;
    }
    return style;
};
```

scheduler.templates.multi_day_event_style

Specifies a css class for multi-day events in the "Day" view

Parameters:

- obj - the event object
- type - the object that specifies items list presentation.

  For details on the parameter see the API article "type"

```
scheduler.templates.multi_day_event_style = function(obj){
    var style = "";
    if(obj.color){
        var rgb = webix.color.toRgb(obj.color);
        style += ";background-color:rgba("+rgb.toString()+",0.3);";
        var hsv = webix.color.rgbToHsv(rgb[],rgb[1],rgb[2]);
        hsv[2] /= 1.6;
        var color = "rgb("+webix.color.hsvToRgb(hsv[],hsv[1],hsv[2])+")";
        style += "color:"+color;
    }
    return style;
};
```

scheduler.templates.week_title

Specifies the title of the "Week" view.

Parameters:

- date - Date object (selected date).

  For details on the parameter see the API article "type"

```
scheduler.templates.week_title = function(date){
    var start = webix.Date.weekStart(date);
    var end = webix.Date.add(start,6,"day",true);
    var format = webix.i18n.headerWeekFormatStr;
    return format(start)+" - "+format(end);
};
```

scheduler.templates.event_class

Specifies an additional css class that will be applied to events in the lists of the 'Week' and 'Month' views.

Parameters:

- obj - the event object
- type - the object that specifies items list presentation.

  For details on the parameter see the API article "type"

```
//The template doesn't have the default definition.
//Below you can find a usage example that illustrates
//how to highlight the text of the TODAY's events
<style>
.today .webix_event_text{
      color:red !important;
}

</style>
```

```
...
scheduler.templates.event_class =  function(obj, type){
  return (webix.Date.datePart(obj.start_date).valueOf()==
        webix.Date.datePart(new Date()).valueOf())?"today":""
}
```

scheduler.templates.event_date

Specifies the content of day headers in the "Week" view

Parameters:

- date - the date which needs to be formatted

```
scheduler.templates.event_date = function(value){
    var date = webix.Date.datePart(new Date(value));
    var today = webix.Date.datePart(new Date());
    var className = ((date.valueOf() == today.valueOf())?" today":"");
    date = webix.i18n.weekDateFormatStr(date);
    return "<span
class='webix_unit_header_inner"+className+"'>"+date+"</span>";
};
```

scheduler.templates.event_marker

Specifies a marker in the 'Week' and 'Month' views.

Parameters:

- obj - the event object
- type - the object that specifies items list presentation.

For details on the parameter see the API article "type"

```
scheduler.templates.event_marker = function(obj,type){
    var style = "";
    if(obj.color){
        var rgb = webix.color.toRgb(obj.color);
        style += ";border-color:"+obj.color+";";
        style += "background-color:rgba("+rgb.toString()+",0.9);";
    }
    return "<div class='webix_event_marker' ><div
style='"+style+"'></div></div>";
};
```

scheduler.templates.event_time

The content of the time part of event items in the "Week" and "Month" views.

Parameters:

- obj - an event object

```
scheduler.templates.event_time = function(obj){
    var start = obj.start_date,
        end = obj.end_date,
        start0 = webix.Date.datePart(start,true).valueOf(),
        end0 = webix.Date.datePart(end,true).valueOf();

    if( start0 == start.valueOf() && end0 == end.valueOf())
        return scheduler.locale.labels.label_allday;
    else
        return webix.i18n.timeFormatStr(start);
};
```

scheduler.templates.event_title

Specifies the content of an event item in the 'Week' view.

Parameters:

- obj - the event object
- type - the object that specifies items list presentation.

For details on the parameter see the API article "type"

```
scheduler.templates.event_title = function(obj,type){
    var className, html, time;

    time = scheduler.templates.event_time(obj);
    html = "<div class='webix_event_time'>"+time+"</div>";
    html += type.marker(obj,type);
    html += "<div class='webix_event_text'>"+obj.text+"</div>";
    className = scheduler.templates.event_class(obj,type);
    return "<div class='webix_event_overall
"+className+"'>"+html+"</div>";
};
```

scheduler.templates.month_event_title

Specifies the content of an event item in the "Month" view

Parameters:

- obj - the event object
- type - the object that specifies items list presentation.

  For details on the parameter see the API article "type"

```
scheduler.templates.month_event_title = function(obj,type){
    var className, html, time;

    time = scheduler.templates.event_time(obj);
    html = "<div class='webix_event_time'>"+time+"</div>";
    html += type.marker(obj,type);
    html += "<div class='webix_event_text'>"+obj.text+"</div>";
    className = scheduler.templates.event_class(obj,type);
    return "<div class='webix_event_overall
"+className+"'>"+html+"</div>";
};
```

scheduler.templates.calendar_event

The template for days with events in the 'Month' view.

Parameters:

- date - the date which needs to be formatted

```
scheduler.templates.calendar_event = function(day){
  var html = "<div class='webix_cal_day_event'>"+day.getDate()+"</div>";
  html += "<div class='webix_cal_event_marker'></div>";
  return html;
};
```

scheduler.templates.new_event_data

Specifies the default event properties. Used when a user creates new events.

```
scheduler.templates.new_event_data= function(){
    var hours = (webix.Date.add(new Date(),1,"hour",true)).getHours();
    //coreData refers to the currently selected date
    var start = webix.Date.copy(this.coreData.getValue());
    start.setHours(hours);
    var end = webix.Date.add(start,1,"hour",true);
    return {start_date:start,end_date:end};
};
```

scheduler.templates.selected_event

The template for an event in the "Selected Event" view.

Parameters:

- obj - the event object

```
scheduler.templates.selected_event : function(obj){
```

```
    var html = "", fts="", fte="", from, to, labels;
    var start = obj.start_date,
        end = obj.end_date,
        start0 = webix.Date.datePart(start,true),
        end0 = webix.Date.datePart(end,true);

    if(!start) return html;
    html = "<div  class='selected_event
"+scheduler.templates.event_class(obj)+"'>";
    html += "<div class='event_title'>"+obj.text+"</div>";

    labels = scheduler.locale.labels;
    from = labels.label_from;
    to = labels.label_to;

    if(start0.valueOf() == end0.valueOf()){
        var fd = webix.i18n.dateFormatStr(start);
        fts = webix.i18n.timeFormatStr(start);
        fte = webix.i18n.timeFormatStr(end);
        html += "<div class='event_text'>"+fd+"</div>";
        html += "<div class='event_text'>"+from+" "+fts+" "+to+"
"+fte+"</div>";
    }
    else{
        var fds = webix.i18n.longDateFormatStr(start);
        var fde = webix.i18n.longDateFormatStr(end);
        /*if not "all-day" event*/
        if(!(start0.valueOf() == start.valueOf() && end0.valueOf() ==
end.valueOf())){
            fts = webix.i18n.timeFormatStr(start)+" ";
            fte = webix.i18n.timeFormatStr(end)+" ";
        }
        html += "<div class='event_text'>"+from+" "+fts+fds+"</div>";
        html += "<div class='event_text'>"+to+" "+fte+fde+"</div>";
    }
    if(obj.details&&obj.details !== ""){
        html += "<div class='event_title'>"+labels.label_details+"</div>";
        html += "<div class='event_text'>"+obj.details+"</div>";
    }
    html += "</div>";
    return html;
};
```

Sizing Elements

You can easily configure the sizes of the scheduler's elements.

To set the desired option, just write it as it's stated in this documentation.

Beware! Configuration options should go before the code line with scheduler initialization.

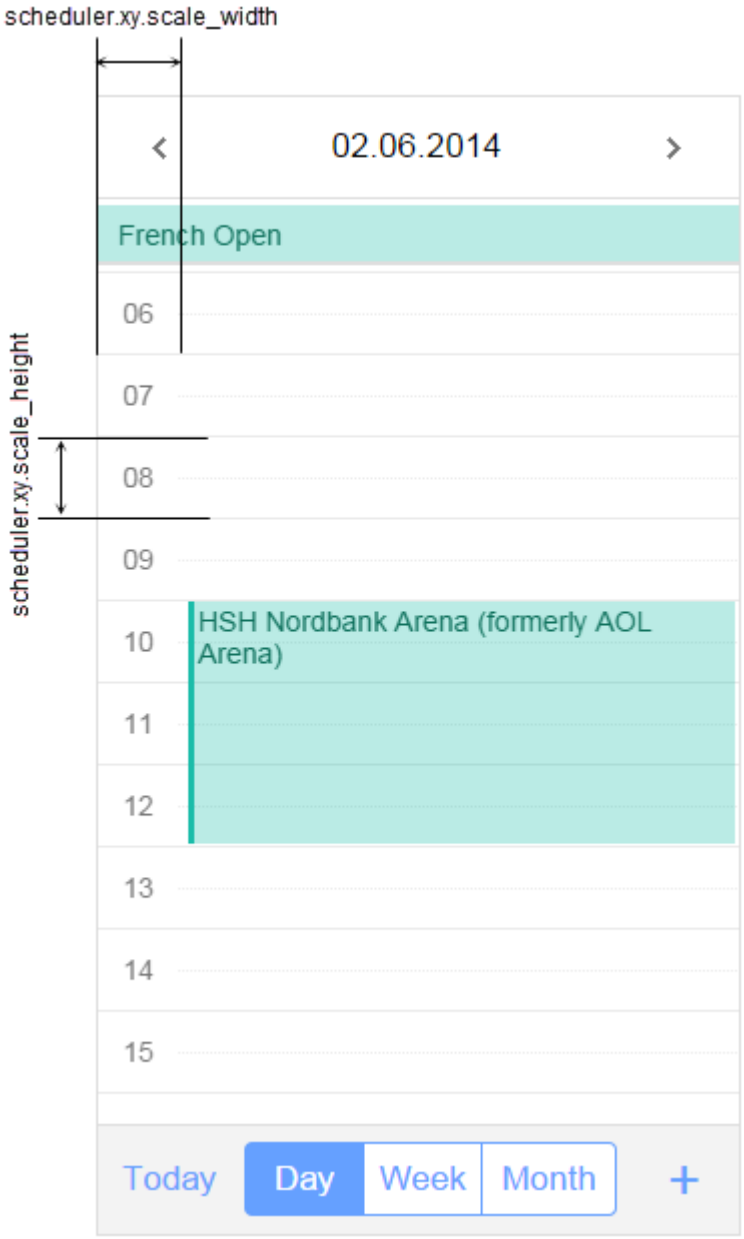
```
scheduler.xy.confirm_height = 300;
scheduler.xy.confirm_width = 300;
//configuration options should go before the scheduler constructor
webix.ready(function(){
    webix.ui.fullScreen();
    webix.ui({
        view: "scheduler",
        id: "scheduler"
    });
    $$("scheduler").load("../scheduler/mobile.xml","scheduler");
});
```

Dimension configuration

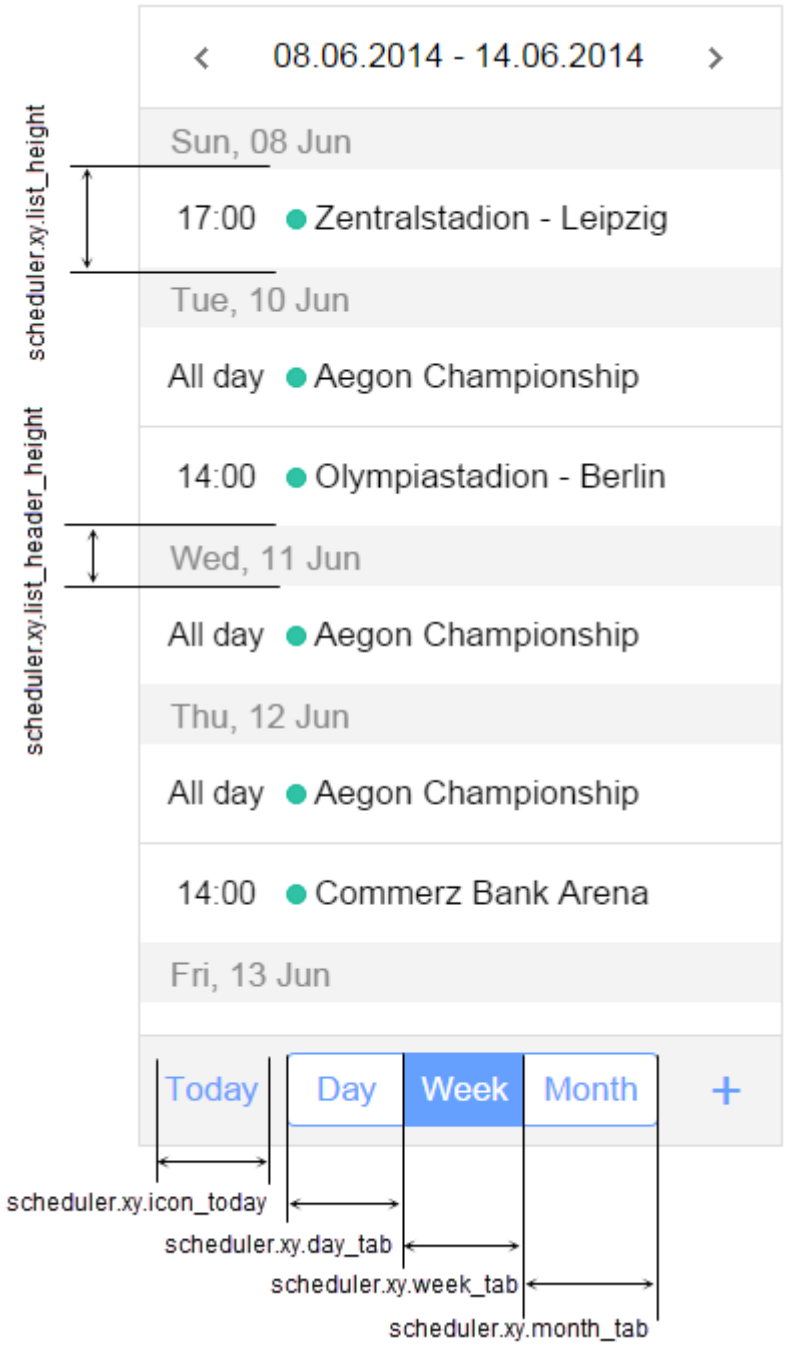
- **scheduler.xy.all_day** - the width of the "All day" label in the "Edit" form
- **scheduler.xy.confirm_height** - (number) the height of the confirm window (231 by default)
- **scheduler.xy.confirm_width** - (number) the width of the confirm window (250 by default)
- **scheduler.xy.day_tab** - (number) the width of the 'Day' tab in the bottom tabbar (60 by default)
- **scheduler.xy.icon_back** - (number) the width of the 'Back' button (70 by default)
- **scheduler.xy.icon_cancel** - (number) the width of the 'Cancel' button (80 by default)
- **scheduler.xy.icon_done** - (number) the width of the 'Done' button (90 by default)
- **scheduler.xy.icon_edit** - (number) the width of the 'Edit' button (90 by default)
- **scheduler.xy.icon_save** - (number) the width of the 'Save' button (90 by default)
- **scheduler.xy.icon_today** - (number) the width of the 'Today' button in the bottom tabbar (55 by default)
- **scheduler.xy.list_header_height** - (number) the height of a grey header with date in the 'Week' view (50 by default)
- **scheduler.xy.list_height** - (number) the height of a row in the 'Week' view (50 by default)
- **scheduler.xy.min_event_height** - (number) the minimal event's height in the 'Day' view
- **scheduler.xy.month_list_height** - (number) the height of a row in the 'month' view (50 by default)
- **scheduler.xy.month_tab** - (number) the width of the 'Month' tab in the bottom tabbar (68 by default)
- **scheduler.xy.scale_height** - (number) the height of a row in the 'Day' view (20 by default)
- **scheduler.xy.scale_width** - (number) the total width of the hour and minute columns in the 'day' view. The width is divided equally between the columns (40 by default)

- **scheduler.xy.week_tab** - (number) the width of the 'Week' tab in the bottom tabbar (60 by default)

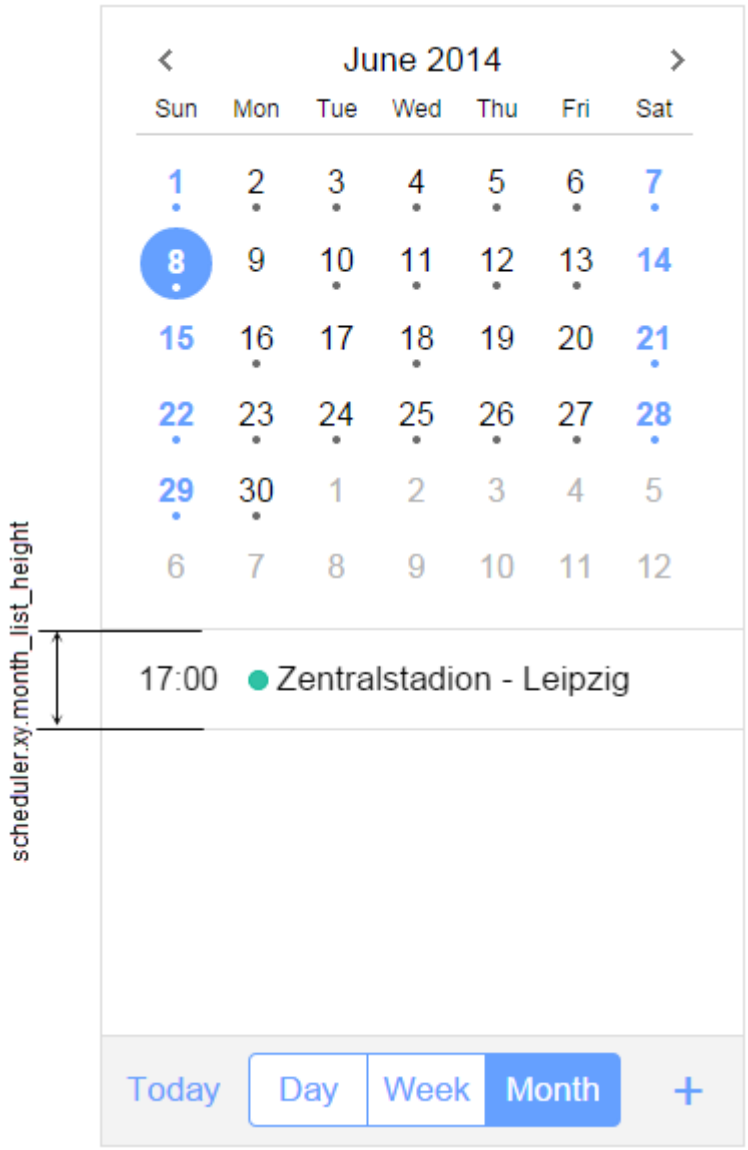
"Day" view



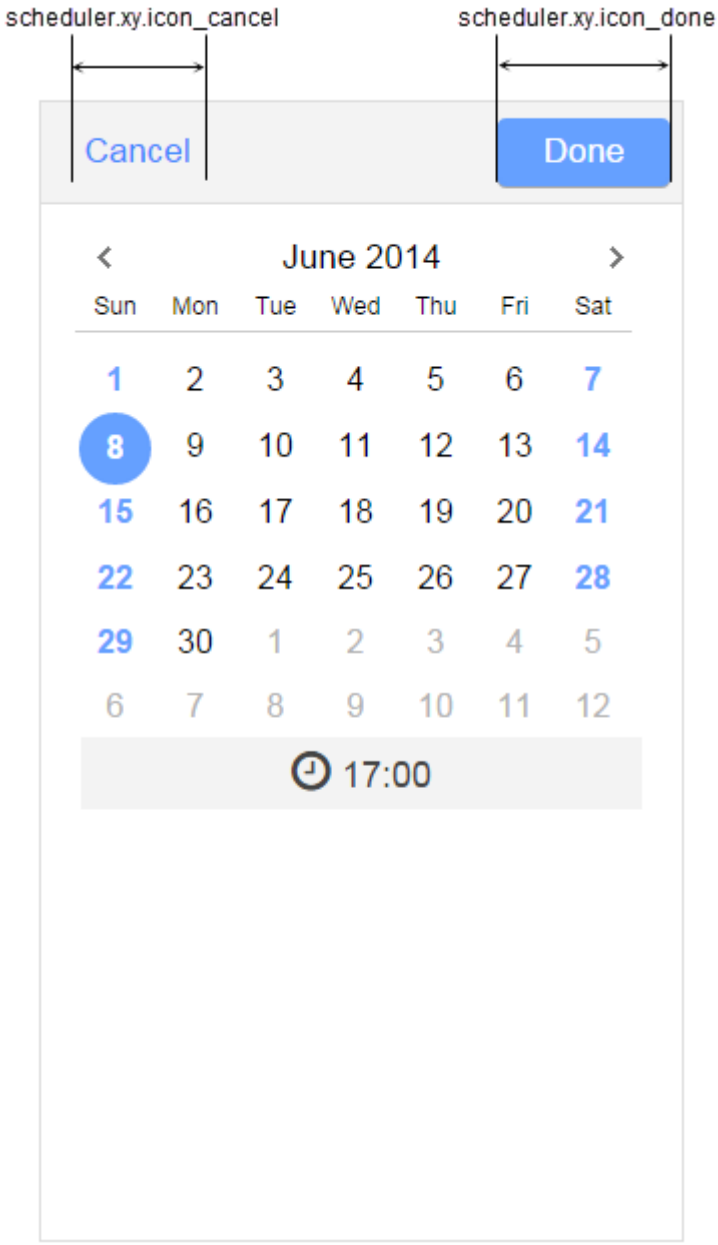
"Week" view


08.06.2014 - 14.06.2014
Sun, 08 Jun
17:00 Zentralstadion - Leipzig
Tue, 10 Jun
All day Aegon Championship
14:00 Olympiastadion - Berlin
Wed, 11 Jun
All day Aegon Championship
Thu, 12 Jun
All day Aegon Championship
14:00 Commerz Bank Arena
Fri, 13 Jun
Today
Day
Week
Month
+
scheduler.xy.list_height
scheduler.xy.list_header_height
scheduler.xy.icon_today
scheduler.xy.day_tab
scheduler.xy.week_tab
scheduler.xy.month_tab

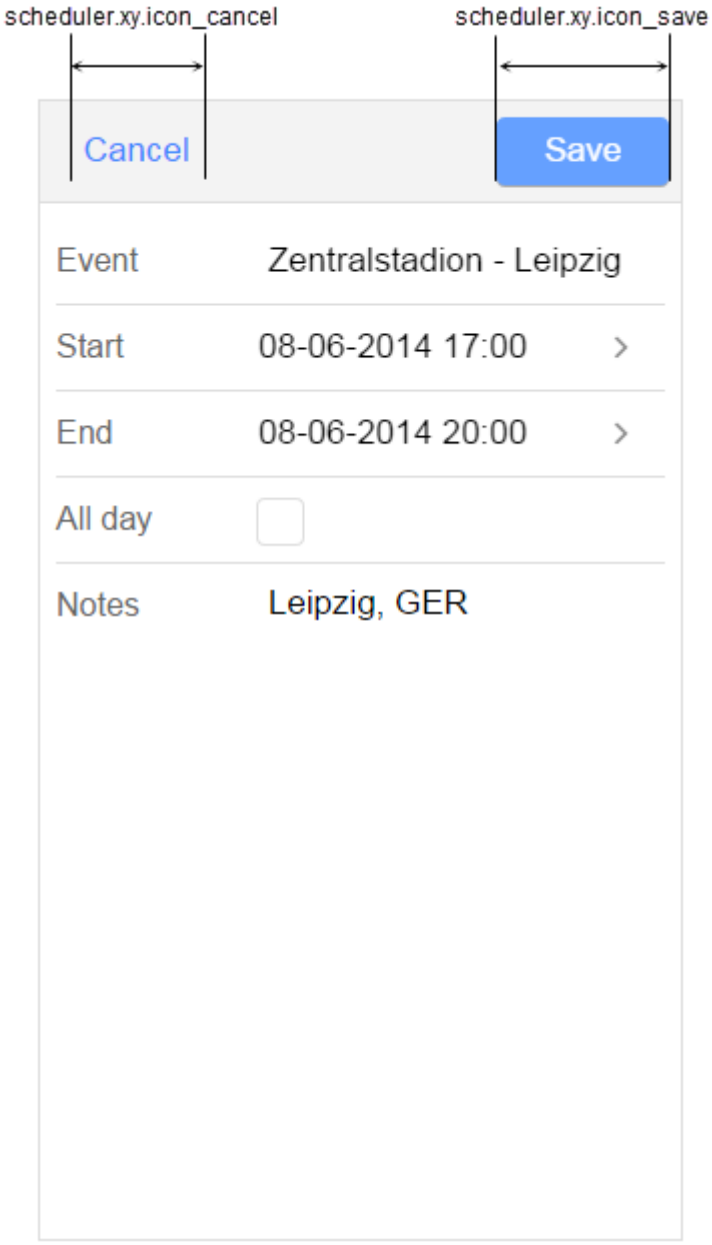
"Month" view



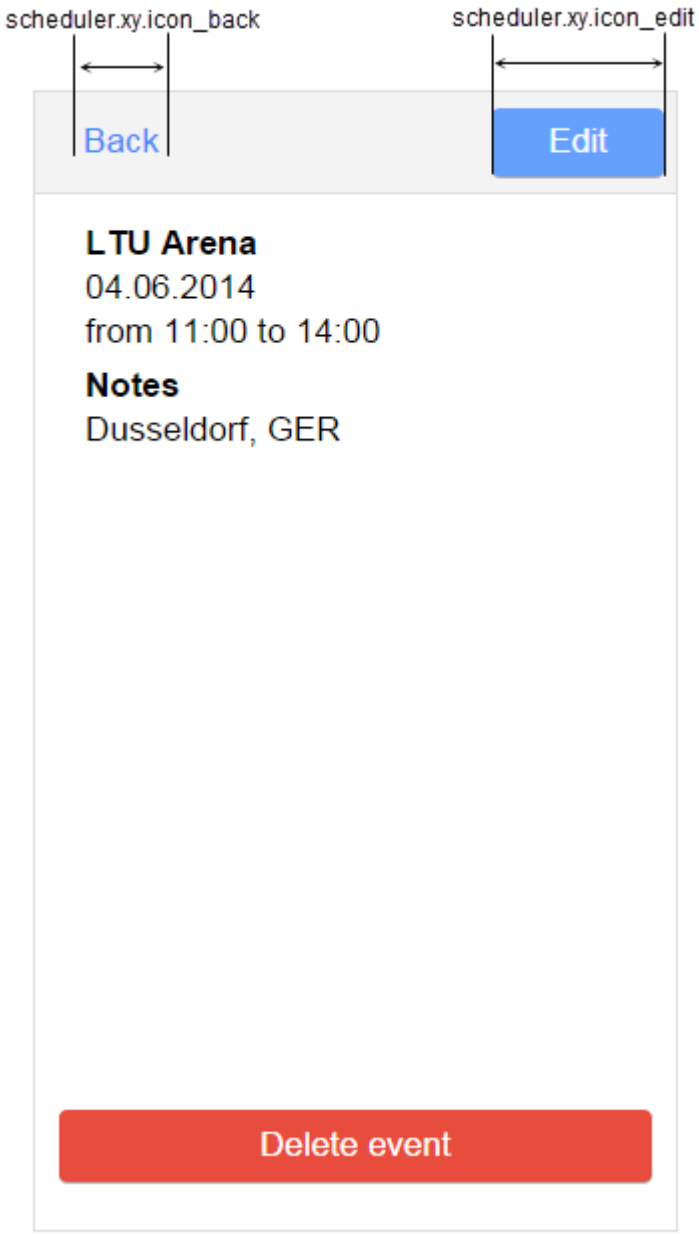
Start(end) date edit form


scheduler.xy.icon_cancel
scheduler.xy.icon_done
Cancel
Done
June 2014
Sun Mon Tue Wed Thu Fri Sat
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 1 2 3 4 5
6 7 8 9 10 11 12
17:00

"Edit" form



"Selected event" view


scheduler.xy.icon_back
scheduler.xy.icon_edit
Back
Edit
LTU Arena
04.06.2014
from 11:00 to 14:00
Notes
Dusseldorf, GER
Delete event

Date Formats

You've got the possiblity to configure date formats of the scheduler without effort.

To set the desired option, just write it as it's stated in this documentation.
Beware! Configuration options should go before the code line with scheduler initialization.

```
scheduler.config.day_header_date = "%d.%m.%Y";
//configuration options should go before the constructor
webix.ready(function(){
    webix.ui.fullScreen();
    webix.ui({
        view: "scheduler",
        id: "scheduler"
    });
    $$("scheduler").load("../scheduler/mobile.xml","scheduler");
});
```

RELATED SAMPLE: Date formatting

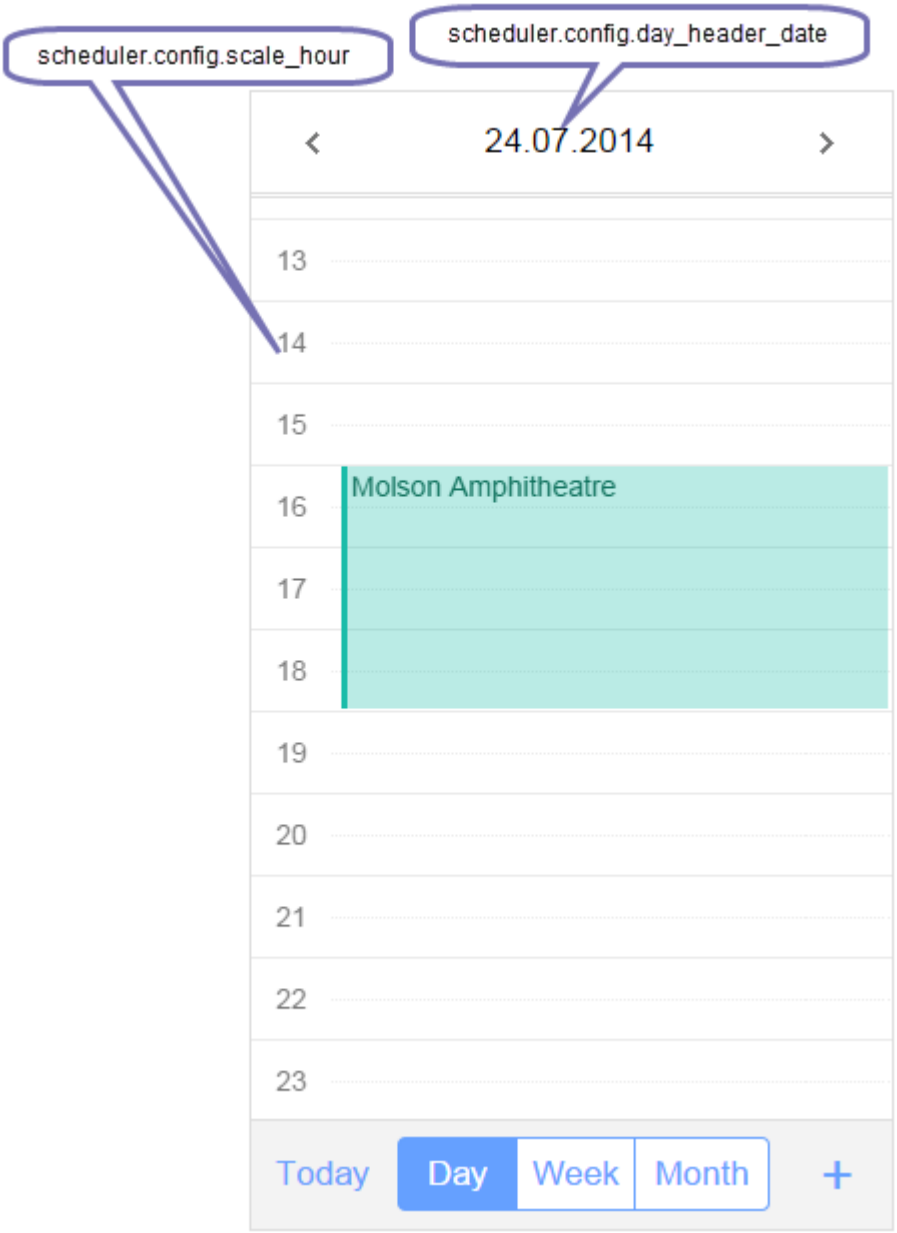
Configuration options

- **scheduler.config.all_day_date** - (string) the format of presenting all-day events in the 'Selected event' view (by default, "%l, %d %F %Y")
- **scheduler.config.calendar_date** - (string) the date format of the header in the 'Month' view (by default, "%F %Y")
- **scheduler.config.calendar_hour** - (string) the format of hours in the 'Start(end) date edit form' (by default, "%H:%i")
- **scheduler.config.day_header_date** - (string) the date format of the header in the 'Day' view (by default, "%d.%m.%Y")
- **scheduler.config.form_all_day** - (string) the format of dates of the 'start', 'end' fields in the 'Edit form' in case the 'All day' option is checked (by default, "%d-%m-%Y")
- **scheduler.config.form_date** - (string) the format of dates of the 'start', 'end' fields in the 'Edit form' in case the 'All day' option isn't checked (by default, "%d-%m-%Y %H:%i")
- **scheduler.config.hour_date** - (string) the format of the time in the 'Week' and 'Month' views (by default, "%H:%i")
- **scheduler.config.item_date** - (string) the default date format for events in the 'Selected event' view (by default,)
- **scheduler.config.parse_date** - (string) the date format in data files. Used by the parser to recognize dates (by default, "%Y-%m-%d %H:%i")
- **scheduler.config.scale_hour** - ("%H" or "%h") the format of the hours in the 'Day' view (by default,

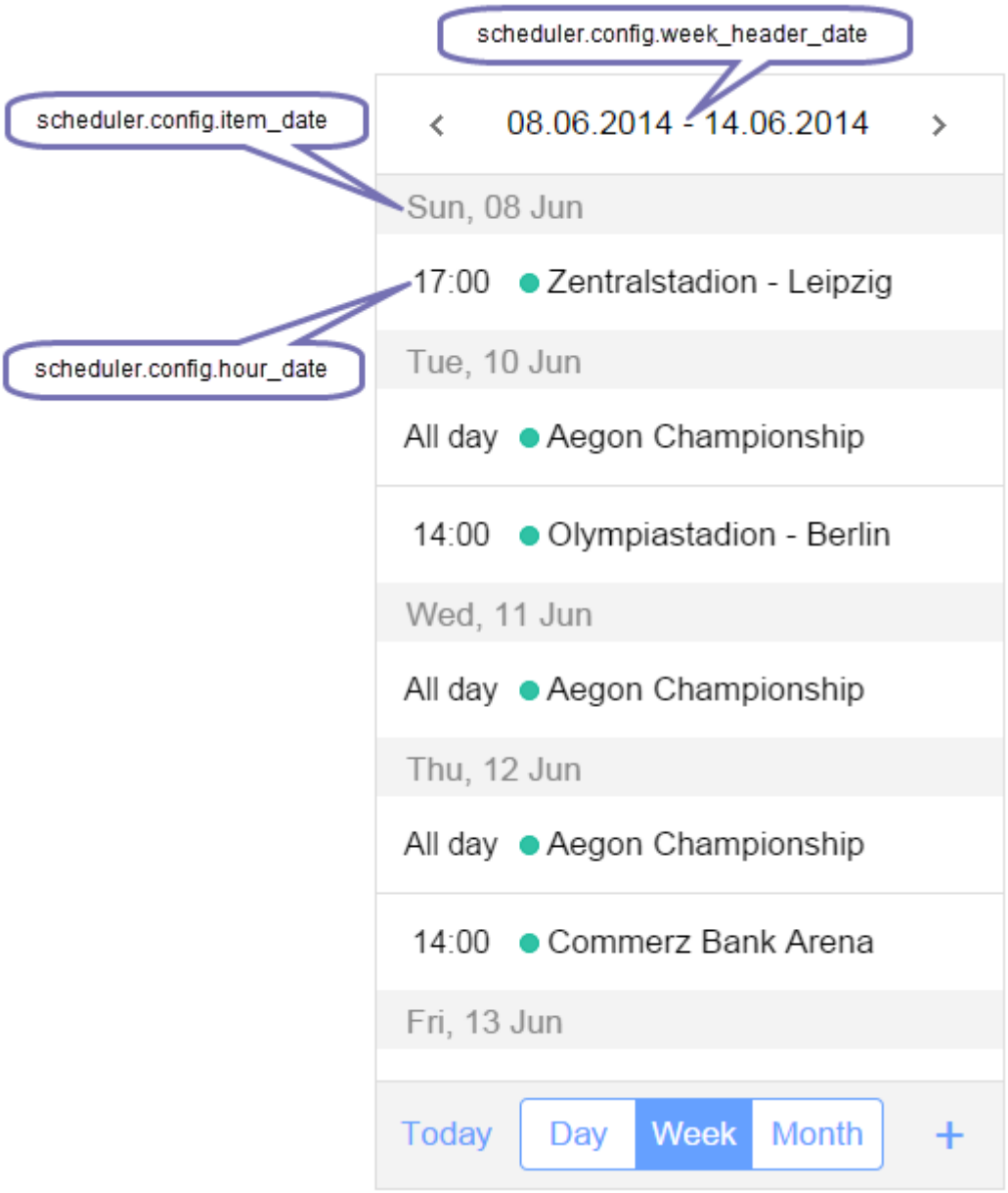
"%H")

- **scheduler.config.week_header_date** - (string) the date format of the header in the 'Week' view ("%d.%m.%Y")
- **scheduler.config.week_date** - (string) the date format of day titles in the 'Week' view ("%D, %d %M")

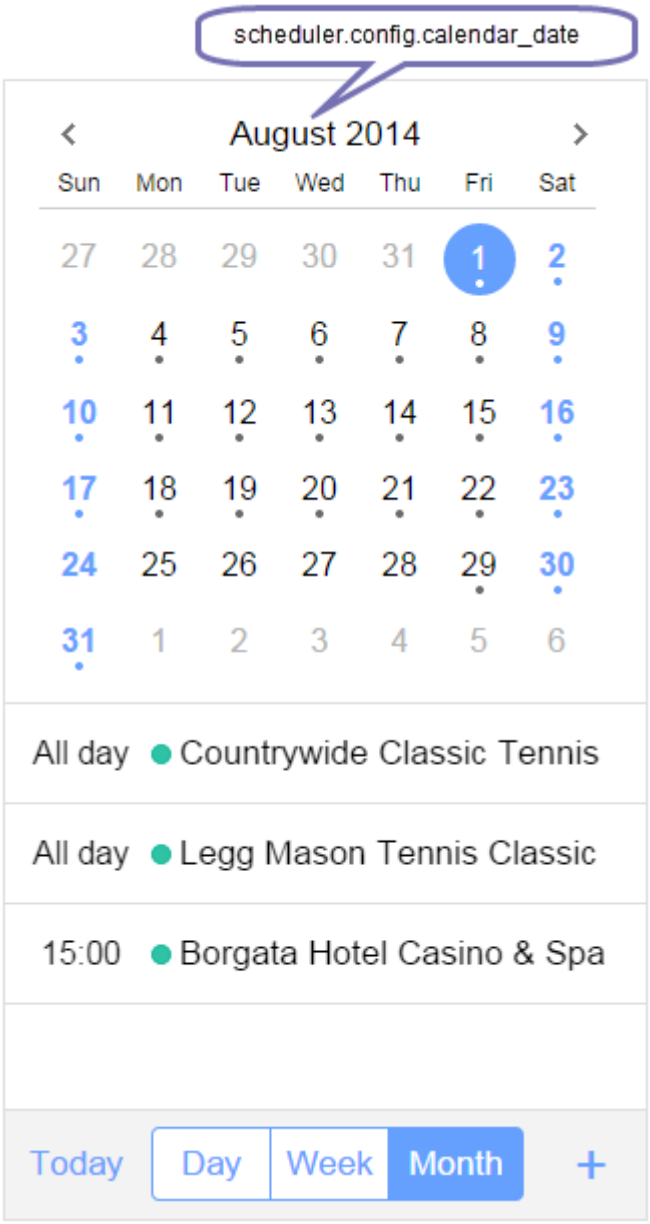
"Day" view



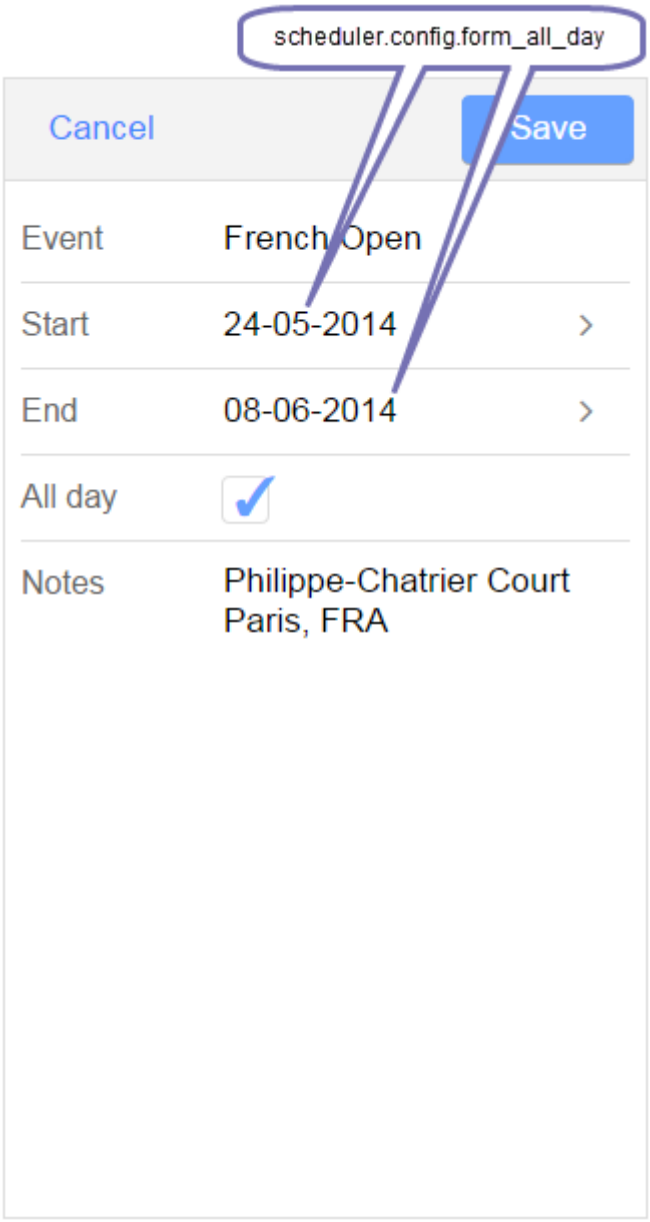
"Week" view

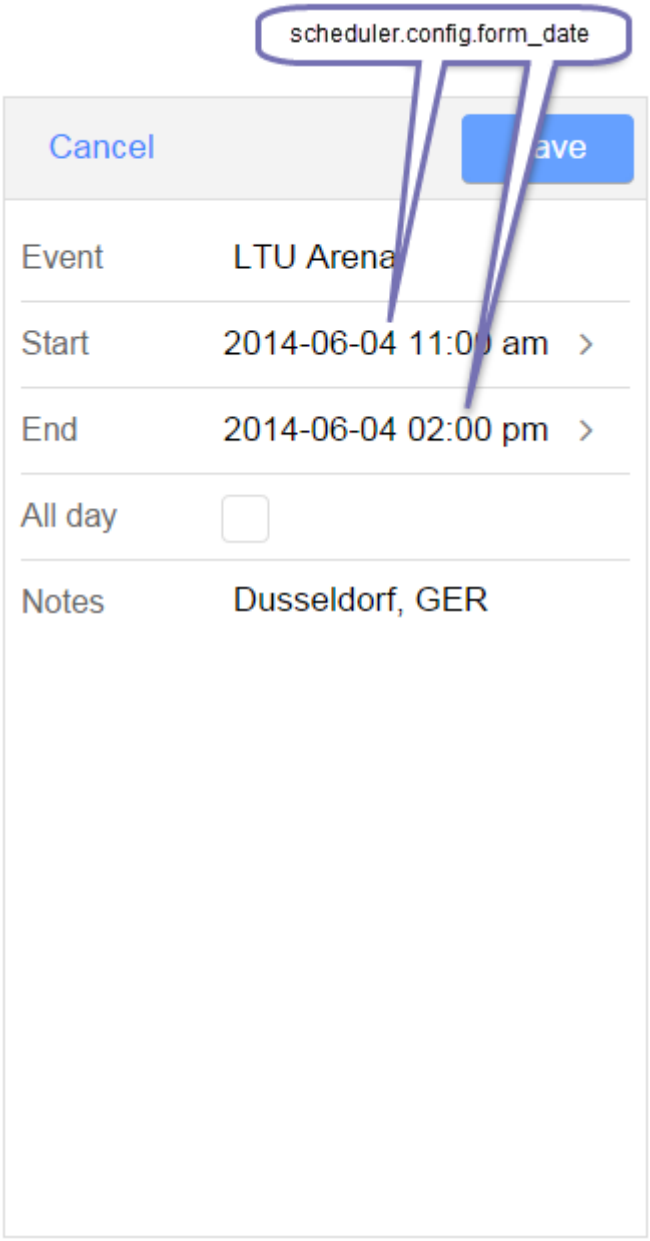

scheduler.config.week_header_date
scheduler.config.item_date
scheduler.config.hour_date
08.06.2014 - 14.06.2014
Sun, 08 Jun
17:00 Zentralstadion - Leipzig
Tue, 10 Jun
All day Aegon Championship
14:00 Olympiastadion - Berlin
Wed, 11 Jun
All day Aegon Championship
Thu, 12 Jun
All day Aegon Championship
14:00 Commerz Bank Arena
Fri, 13 Jun
Today
Day
Week
Month
+

"**Month**" view

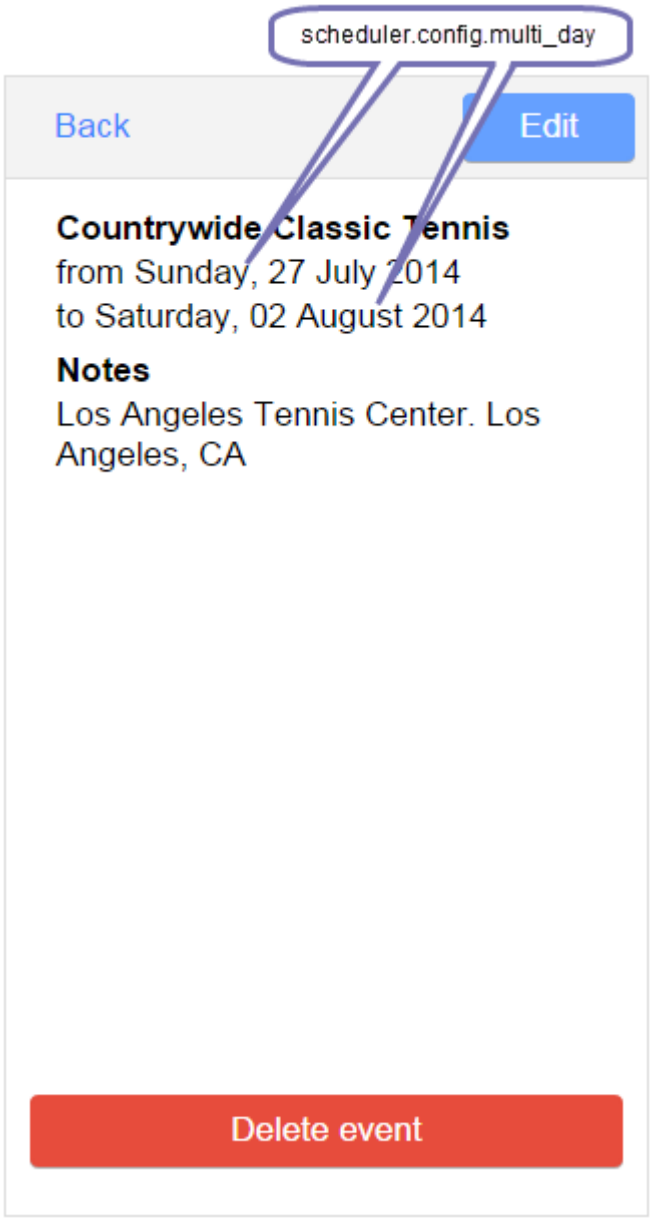

scheduler.config.calendar_date
August 2014
Sun Mon Tue Wed Thu Fri Sat
27 28 29 30 31 1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31 1 2 3 4 5 6
All day Countrywide Classic Tennis
All day Legg Mason Tennis Classic
15:00 Borgata Hotel Casino & Spa
Today
Day
Week
Month
+

Edit form

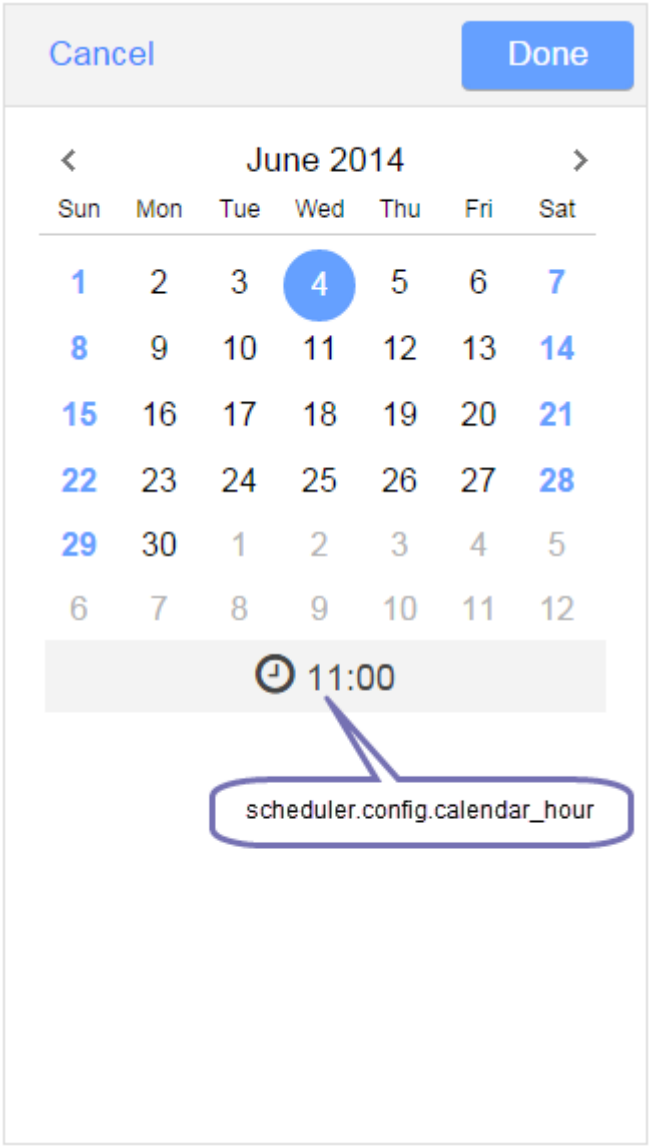

scheduler.config.form_all_day
Cancel
Save
Event
French Open
Start
24-05-2014
End
08-06-2014
All day
Notes
Philippe-Chatrier Court
Paris, FRA



Selected event view



Start(end) date edit form



Date format

Format string can contain any separator character and any element from the following list:

- **%y** - the year as a two-digit number (00 to 99);
- **%Y** - the year as a four-digit number (1900–9999);
- **%m** - the month as a number without a leading zero (1 to 12);
- **%n** - the month as a number with a leading zero (01 to 12);
- **%M** - the month as an abbreviation (Jan to Dec);
- **%F** - the month as a full name (January to December);
- **%W** - the ISO-8601 week number of the year. Weeks start on Monday;
- **%d** - the day as a number with a leading zero (01 to 31);
- **%j** - the day as a number without a leading zero (1 to 31);
- **%D** - the day as an abbreviation (Sun to Sat);
- **%l** - the day as a full name (Sunday to Saturday);

- **%h** - the hour based on the 12-hour clock with a leading zero (01 to 12);
- **%H** - the hour based on the 24-hour clock with a leading zero (00 to 23);
- **%g** - the hour based on the 12-hour clock without a leading zero (1 to 12);
- **%G** - the hour based on the 24-hour clock without a leading zero (0 to 23);
- **%i** - the minute as a number with a leading zero (00 to 59);
- **%s** - the second as a number without a leading zero (00 to 59);
- **%a** - displays **am** (for times from midnight until noon) and **pm** (for times from noon until midnight);
- **%A** - displays **AM** (for times from midnight until noon) and **PM** (for times from noon until midnight).

For example, if you want to present 1st June 2011 as 01/06/2011, you should specify ”%d/%m/%Y”.

Styling Scheduler

It's possible to customize any element of Webix Scheduler.
All you need to do is to redefine the needed attributes of the related css class in the <style> block of your page. The untouched attributes will take default values.

```
<style>
     .webix_bottombar{
         background:#6699FF !important;
      }

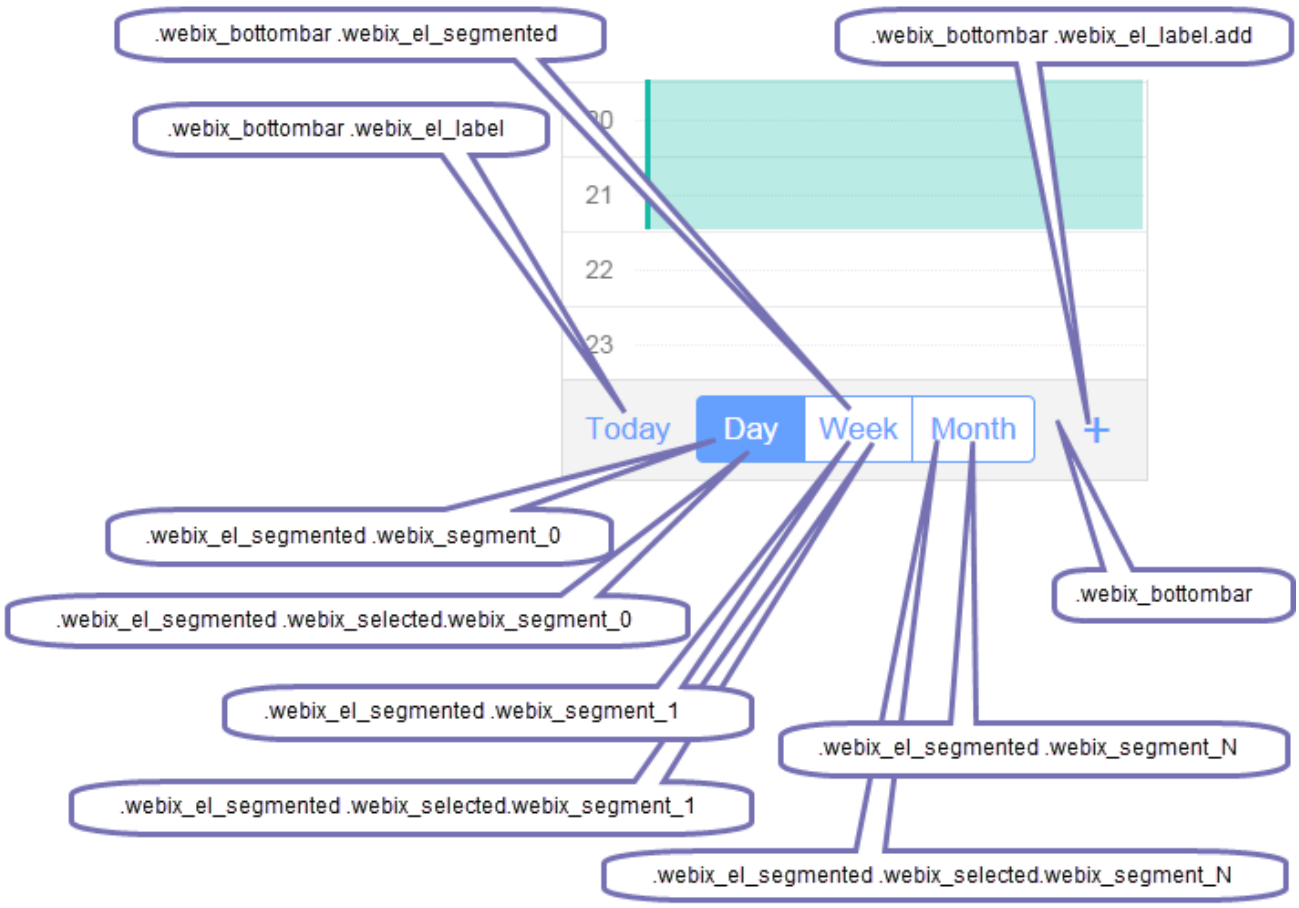
</style>
```

In the above example we redefined only the background color. All the other attributes defined in the **'.webix_bottombar'** class took the default values.

CSS classes and the related elements

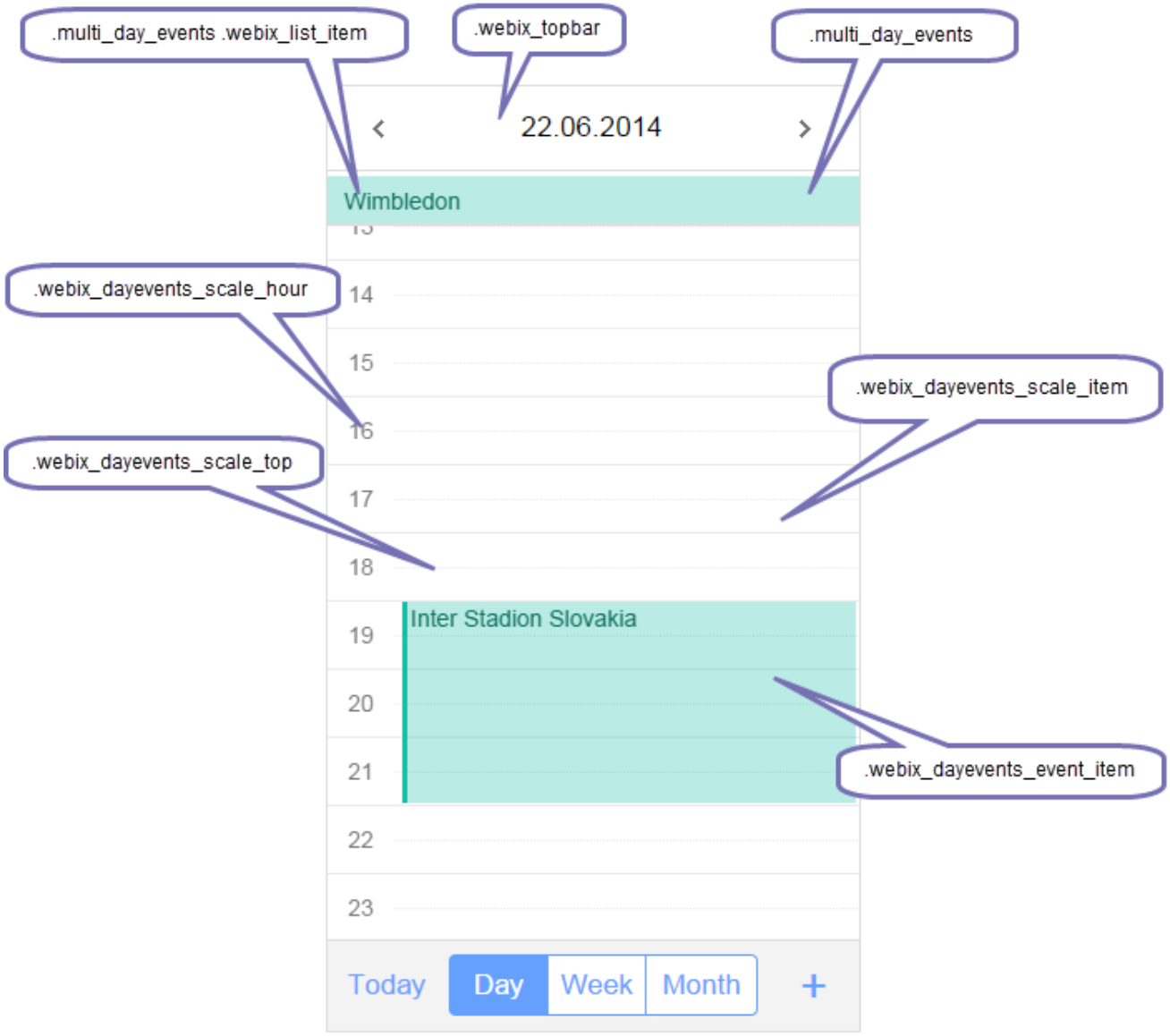
Main Toolbar

- **.webix_bottombar** - the toolbar
- **.webix_bottombar .webix_el_label** - the 'Today' button
- **.webix_bottombar .webix_el_label.add** - the '+' (add) button
- **.webix_bottombar .webix_el_segmented** - the tabbar
- **.webix_el_segmented .webix_segment_0** - the first tab
- **.webix_el_segmented .webix_selected.webix_segment_0** - the first tab in the selected state
- **.webix_el_segmented .webix_segment_1** - the second tab
- **.webix_el_segmented .webix_selected.webix_segment_1** - the second tab in the selected state
- **.webix_el_segmented .webix_segment_N** - the last tab
- **.webix_el_segmented .webix_selected.segment_N** - the last tab in the selected state



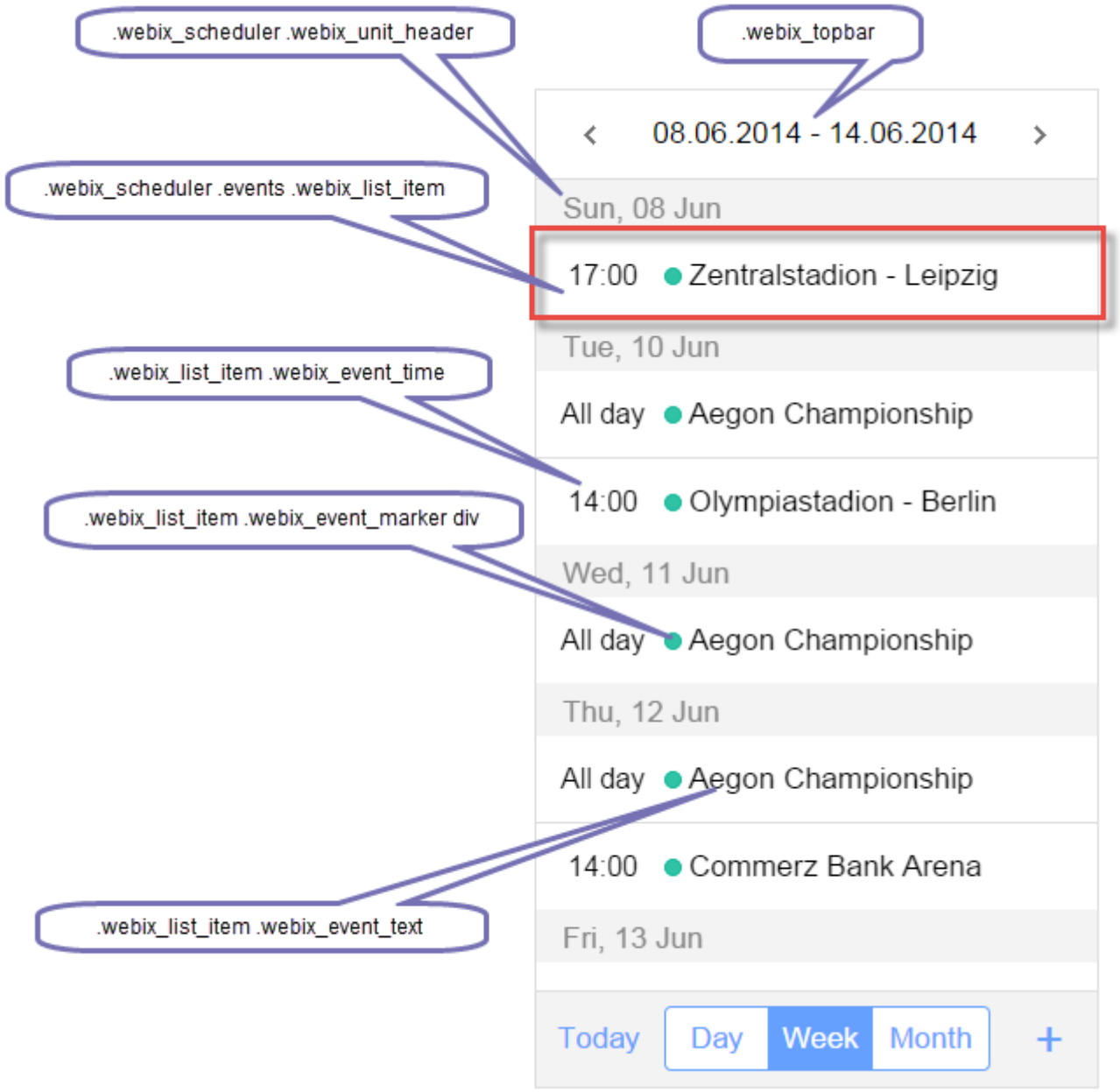
'Day' view

- **.webix_dayevents_event_item** - the container of an event
- **.webix_topbar** - the top toolbar
- **.webix_dayevents_scale_item** - the container of a single scale item
- **.webix_dayevents_scale_top** - the half an hour separator
- **.webix_dayevents_scale_hour** - the container inside of the scale item that divides an hour into halves
- **.multi_day_events** - the container for multi-day events
- **.multi_day_events .webix_list_item** - a multi-day event item



'Week view'

- **.webix_topbar** - the top toolbar
- **.webix_scheduler .events .webix_list_item** - the container for an event's elements
- **.webix_scheduler .webix_unit_header** - the event's header with the date
- **.webix_list_item .webix_event_time** - the container with the time of an event
- **.webix_list_item .webix_event_marker div** -the round marker of an event
- **.webix_list_item .webix_event_text** - the container with the text of an event

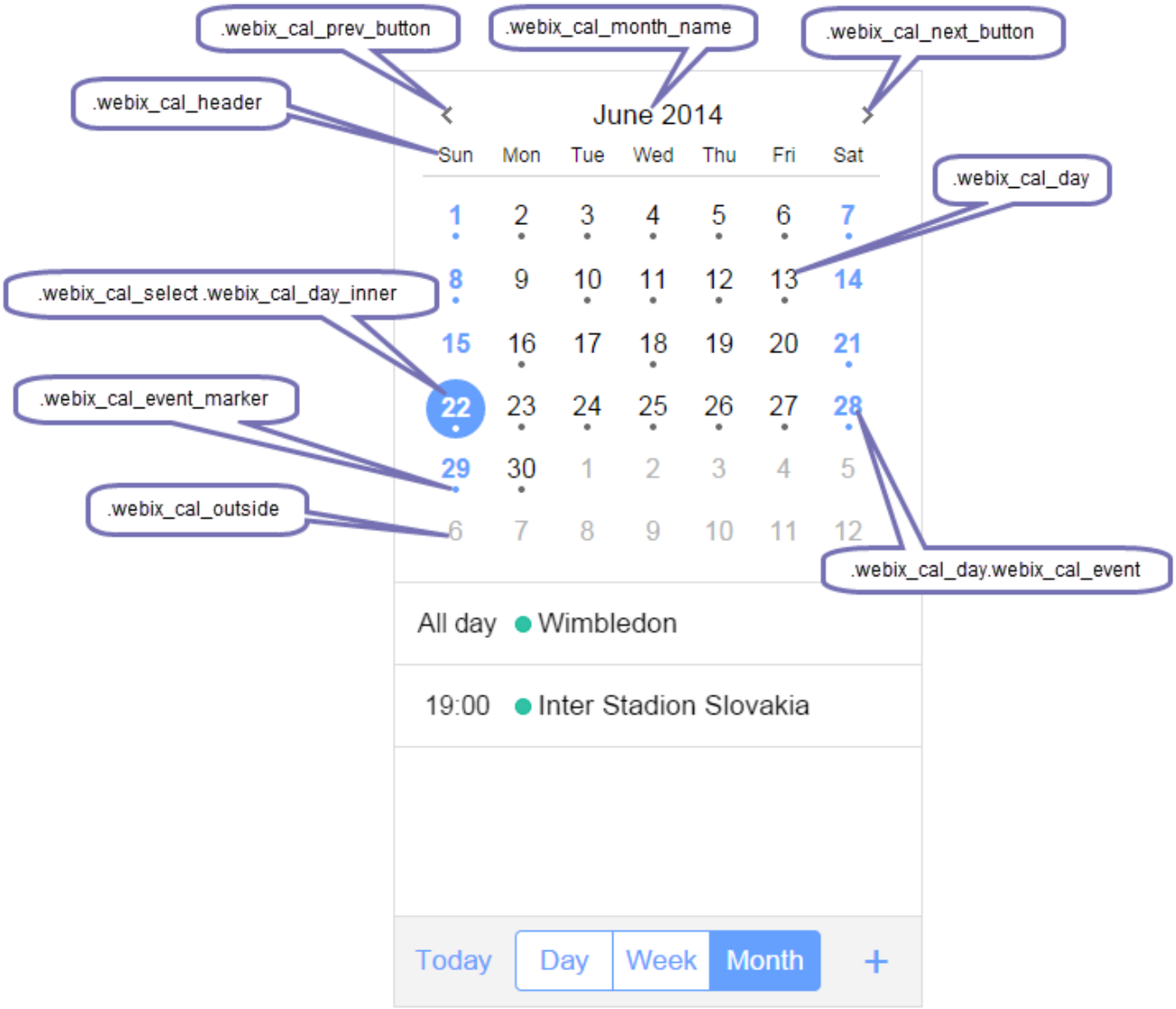


'Month' view

- **.webix_calendar** - the style of the calendar's container
- **.webix_cal_prev_button** - the 'previous' icon on the top toolbar
- **.webix_cal_next_button** - the 'next' icon on the top toolbar
- **.webix_cal_month_name** - the toolbar title
- **.webix_cal_header** - the container of the week header
- **.webix_cal_day** - the container for a day
- **.webix_cal_day.webix_cal_event** - the style of weekends
- **.webix_cal_outside** - the style for days outside the current month's date range
- **.webix_cal_select .webix_cal_day_inner** - the style of the selected day
- **.webix_cal_event_marker** - an event marker (for days with events)

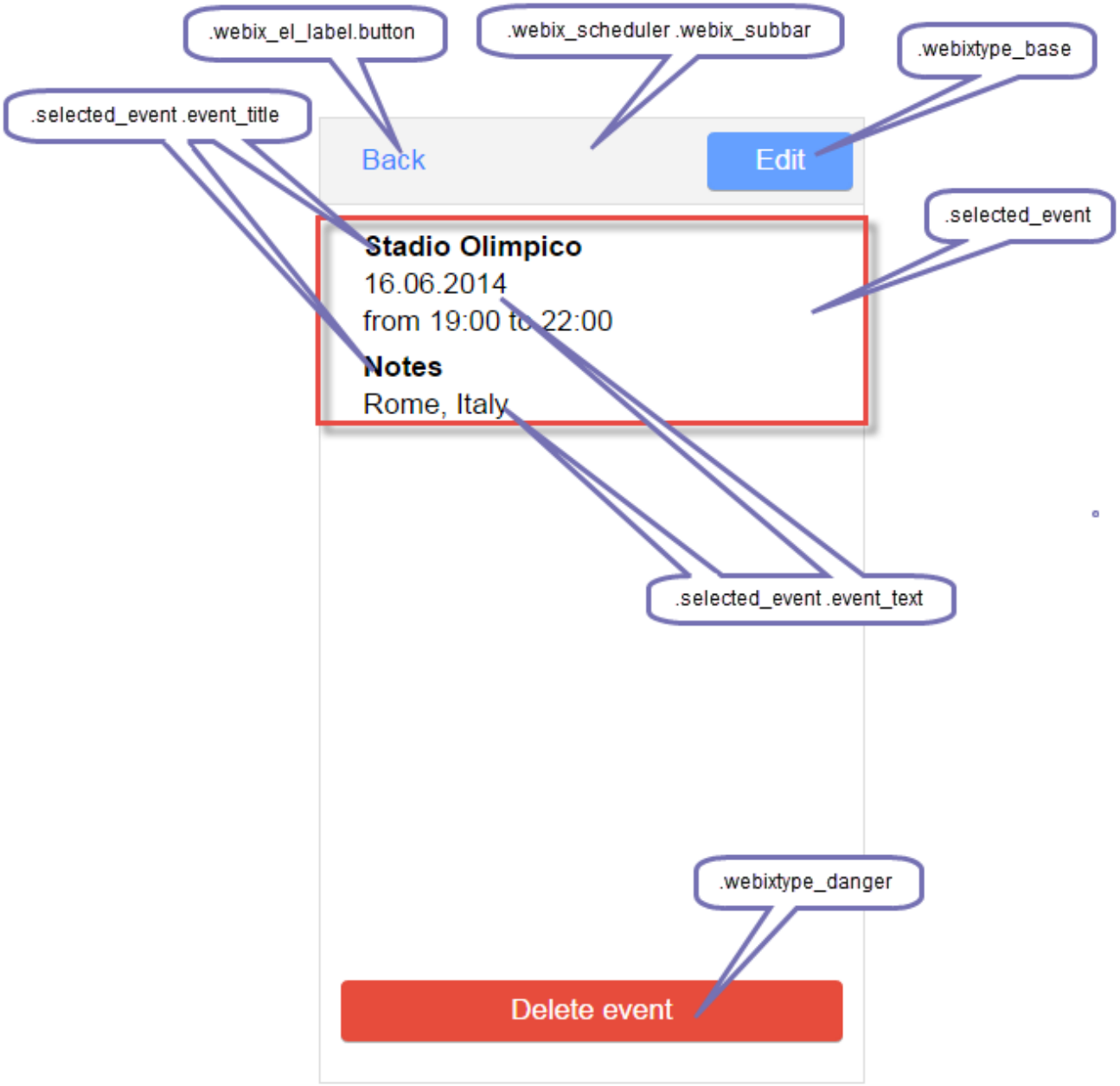
The styles of an event's items (see the image of Week View):

- **.day_events** - the style of the events' list container
- **.webix_scheduler .events .webix_list_item** - the container for an event's elements
- **.webix_list_item .webix_event_time** - the container with the time of an event
- **.webix_list_item .webix_event_marker div** -the round marker of an event
- **.webix_list_item .webix_event_text** - the container with the text of an event



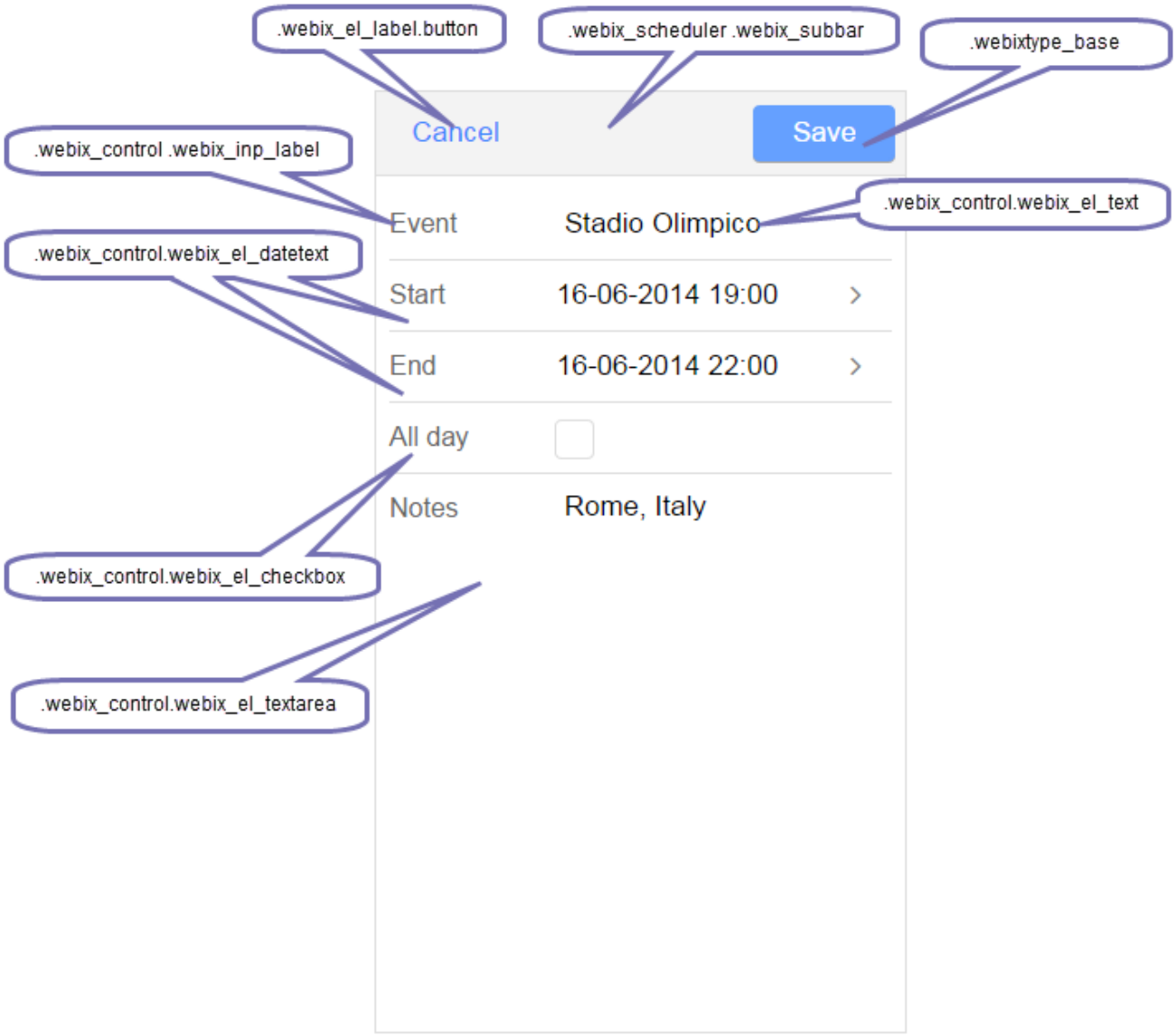
'Selected Event' View

- **.selected_event** - the container with an event
- **.selected_event .event_title** - the title of an event
- **.selected_event .event_text** - the description of an event
- **.webix_scheduler .webix_subbar** - the toolbar
- **.webixtype_base** - the 'Edit' button
- **.webix_el_label.button** - the 'Back' button
- **.webixtype_danger** - the 'Delete' button



Edit form

- **.webix_scheduler .webix_subbar** - the toolbar
- **.webixtype_base** - the 'Save' button
- **.webix_el_label.button** - the 'Cancel' button
- **.webix_control .webix_inp_label** - the style of an element's label (ex. “Event”,”Start”,”End”)
- **.webix_control.webix_el_text** - an event's title
- **.webix_control.webix_el_datetext** - the elements with "Start" and "End" dates
- **.webix_control.webix_invalid** - the style of a field with invalid data
- **.webix_control.webix_el_checkbox** - the "All day" checkbox element
- **.webix_control.webix_el_textarea** - the textarea style (the “Notes” field)



Confirm window

- **.webix_modal_box** - a window container
- **.webix_popup_title** - the window's title
- **.webix_popup_text** - the window's message
- **.webix_popup_button.confirm div** - the "Yes" button
- **.webix_popup_button div** - the "No" button



Event styling

It is possible to apply different styling to events in the scheduler. It's implemented with the help of templates.

The "event_class" template is applied to events in all 3 views: **Day**, **Week** and **Month**.

For example you can set different styles for event markers, depending on properties in event data:

```
<style>
.my_calendar .webix_event_marker div{
    background-color: rgba(102, 204, 255, 0.9);
    border-color: #66ccff;
}

.calendar2 .webix_event_marker div{
    background-color: rgba(, 204, 204, 0.9);
    border-color: #00cccc;
}

</style>
<script>
// template for class name of event container
    scheduler.templates.event_class =  function(obj, type){
        return (!obj.group?"my_calendar":"calendar"+obj.group);
};
</script>
```

RELATED SAMPLE: Event styling

There is a possibility to apply different colors to events using the **color** property in the event's data. This property is applied to the event's background in the **Day** view and event markers in the **Week** and **Month** views. However, you can use any other property for styling events.

```
$$("scheduler").parse([{id:"1", text:"Event 1",
color:"#66ccff",...},...],"json");
```

RELATED SAMPLE: Event colors

The default coloring rules can also be changed in the following templates:

- scheduler.templates.day_event_style - returns css text that will be applied to the event container in the 'Day' view
- scheduler.templates.multi_day_event_style - returns css text that will be applied to multi-day event in the 'Day' view
- scheduler.templates.event_marker - returns css text that will be applied to markers in the 'Week' and 'Month' views

Localizing Scheduler

Basic principles

The mobile version as well as the desktop one allows you to adapt the scheduler to a particular language and a region.

To localize the scheduler to some language you should know the following:
All locale's parameters for the mobile version are contained in the **scheduler.locale.labels** object and to redefine the parameters you should 'rewrite' this object.

For 'rewriting' you should use the technique below:

```
scheduler.locale.labels = labels;
```

where **labels** is any object variable that contains the redefined parameters.

```
var labels = {
    icon_today : "...",
    ...
}
```


In order to localize dates (week days and months) you should follow the instruction given in the article Date and Number Localization.

Localization params for scheduler

The table below presents all localization parameters available for the mobile version of the scheduler.

```
scheduler.locale.labels = {
    list_tab: "List",
    day_tab: "Day",
    month_tab: "Month",
    icon_today: "Today",
    icon_save: "Save",
    icon_done: "Done",
    icon_delete: "Delete event",
    icon_cancel: "Cancel",
    icon_edit: "Edit",
    icon_back: "Back",
    icon_close: "Close form",
    icon_yes: "Yes",
```

```
    icon_no: "No",
    confirm_closing: "Your changes will be lost, are your sure ?",
    confirm_deleting: "The event will be deleted permanently, are you
sure?",
    label_event: "Event",
    label_start: "Start",
    label_end: "End",
    label_details: "Notes",
    label_from: "from",
    label_to: "to",
    label_allday: "All day",
    label_time: "Time"
};
```

RELATED SAMPLE: Locale

Recurring events

Recurring events use the same technique, but take another locale object - **scheduler.locale.labels.recurring**.

Localization parameters for recurring events:

```
scheduler.locale.labels.recurring = {
    none: "Never",
    daily: "Daily",
    day: "day(s)",
    every: "Every",
    weekly: "Weekly",
    week: "week(s) on",
    each: "Each",
    on_day: "on the",
    monthly: "Monthly",
    month: "month(s)",
    month_day: "on what day of the month",
    week_day: "on what day of the week",
    yearly: "Yearly",
    year: "year(s) in",
    counters: ["the first","the second","the third", "the fourth", "the
fifth"],
    never: "Never",
    repeat: "Repeat",
    end_repeat: "End repeat",
    endless_repeat: "Endless repeat",
    end_repeat_label: "Repeat will end by"
};
```

Read-only Mode

The mobile scheduler can be used in 2 modes:

- **Edit** (by default)
- **Read-only**

To activate the **read-only** mode you should set the **scheduler.config.readonly** option to true. Beware! The option should go before the code line with scheduler initialization.

From now on, users can't modify events. The 'Add', 'Edit', 'Delete event' buttons, allowing to do it, will be hidden.

```
scheduler.config.readonly = true;
webix.ready(function(){
    webix.ui.fullScreen();
    webix.ui({
        view: "scheduler",
        id: "scheduler"
    });
    $$("scheduler").load("../php/events.php","scheduler");
});
```

RELATED SAMPLE: Read-only

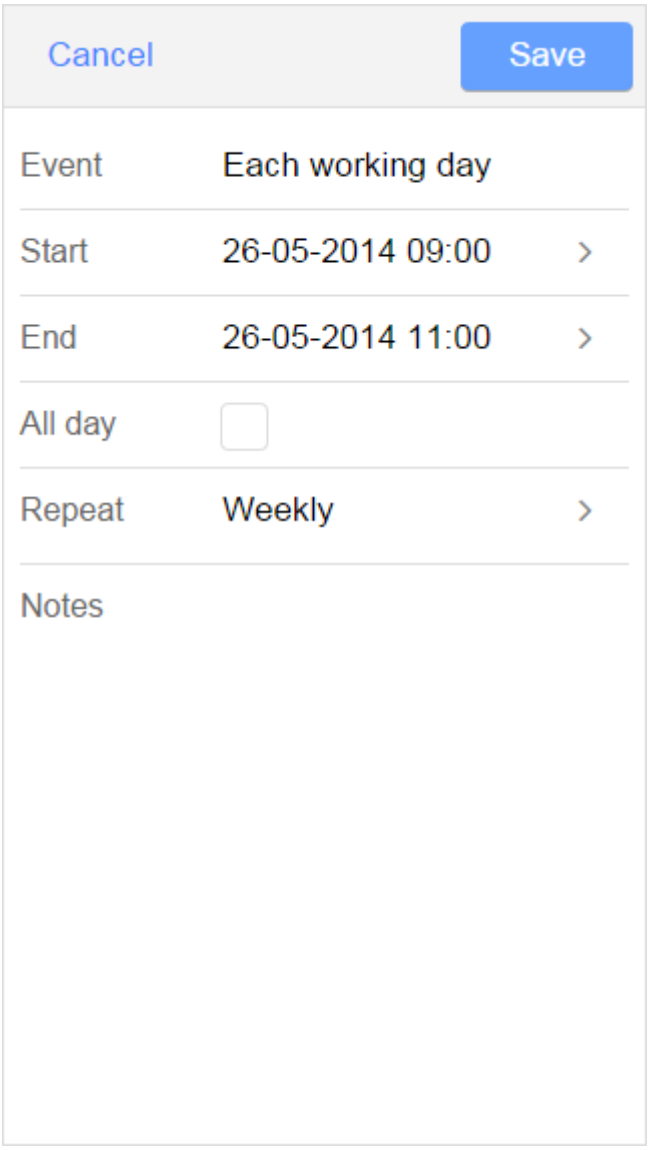
Recurring Events

Enabling functionality

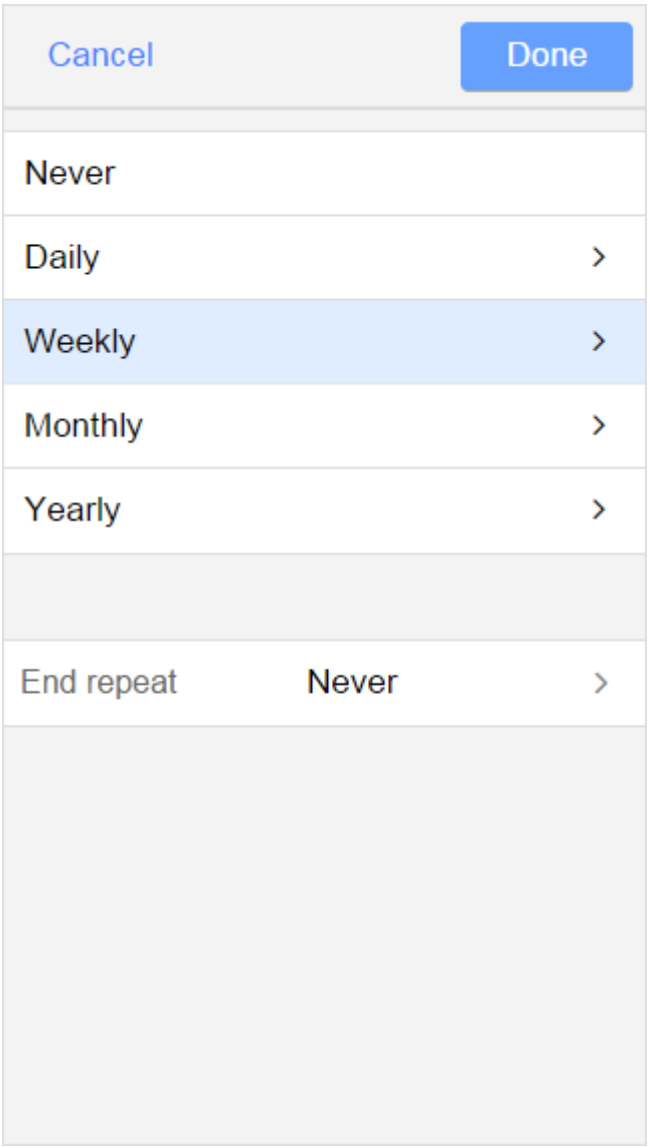
You can set a repetition period for events in the scheduler. For this purpose, you should enable the recurring events functionality, by means of using the following configuration property:

scheduler.config.recurring = true;

Once the recurring events are enabled, the "Edit form" starts looking like shown below:



To set a single event in this form, the user must choose the option **Never** in the **Repeat field**.



RELATED SAMPLE: Recurring view

Format details

A recurring event is stored in the database as a single record that contains all fields of regular events + 2 additional:

- **rec_type** - defines the logic of repetition;
- **event_length** - the actual time length of an event in seconds.

However, the fields **start_date** and **end_date** slightly change their meaning:

- **start_date** - the start date of the first event in a series in the format 'yyyy-mm-dd hh:mm:ss' (format details);
- **end_date** - the end date of the last event in a series in the format 'yyyy-mm-dd 00:00:00' (format details).

For example, a recurring event that starts on January 3, 2013 at 10:00, repeats every day and ends on January 13, 2013 at 12:00 will be presented in the db as follows:

```
id:1,
start_date:"2013-01-03 09:00:00",
end_date:"2013-01-13 00:00:00",
text:"some_text",
details:"",
rec_type:"day____",
event_length:"7200"
```

The client side gets data from the **rec_type** field as a string of the following format:

[type]_[count]_[day]_[count2]_[days]#[extra]

- **type** - the type of repetition: 'day','week','month','year';
- **count** - the interval between events in the “type” units;
- **day** and **count2** - define the day of a month (first Monday, third Friday, etc.);
- **days** - a comma-separated list of the affected week days;
- **extra** - the extra info that can be used to change presentation of recurring details.

Examples of the **rec_type** data:

- day_3___ - each three days;
- month_2___ - each two month;
- month_1_1_2_ - the second Monday of each month;
- week_2___1,5 - Monday and Friday of each second week;

The double or triple underline indicates that the related parameters of the string are omitted.

```
webix.ready(function(){
    webix.ui.fullScreen();
    webix.ui({ view:"scheduler", id:"scheduler"});

    var data = [
        {
            id:1,
            start_date:"2012-06-01 09:00:00",
            end_date:"2092-02-01 09:00:00",
            text:"going to the job",
            details:"",
            rec_type:"week_1___1,2,3,4,5",
            event_length:"7200"
        },
                ...
    ];

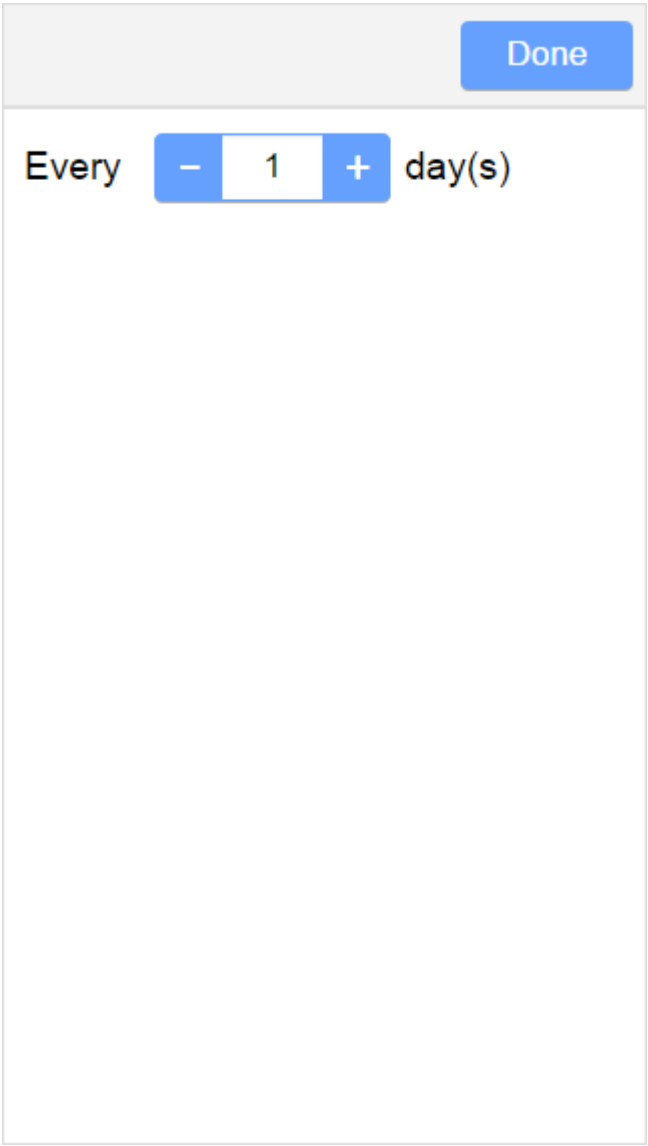
    $$("scheduler").parse(data);
```

```
});
```

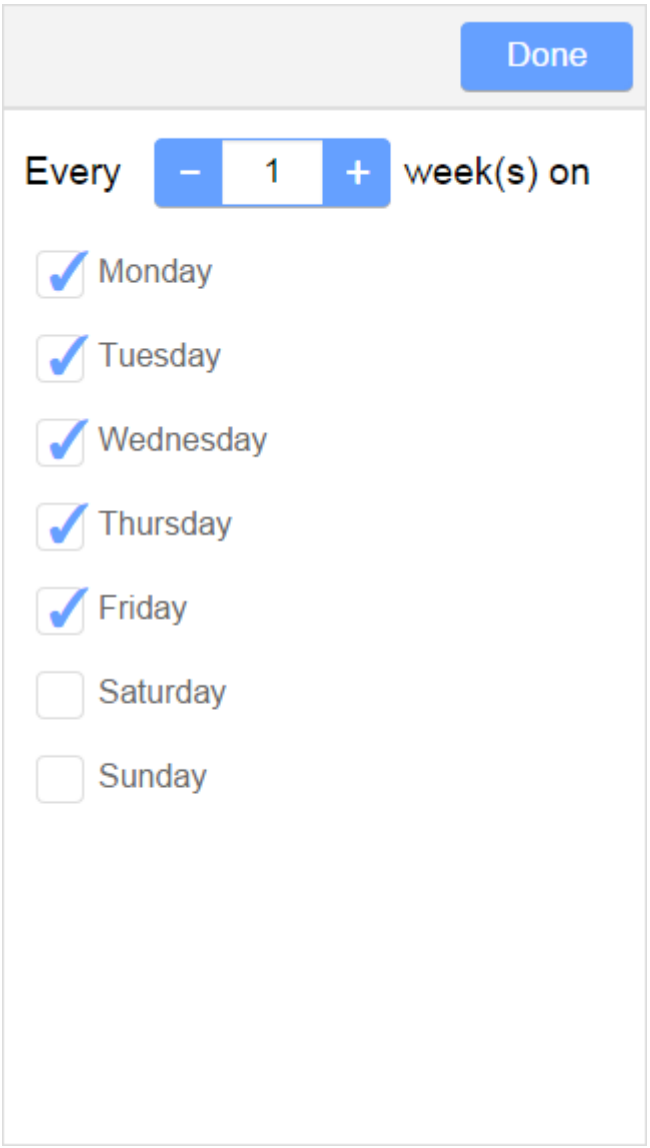
Repetition variations

As it was mentioned above, the **Repeat field** allows setting 4 different repetition steps:

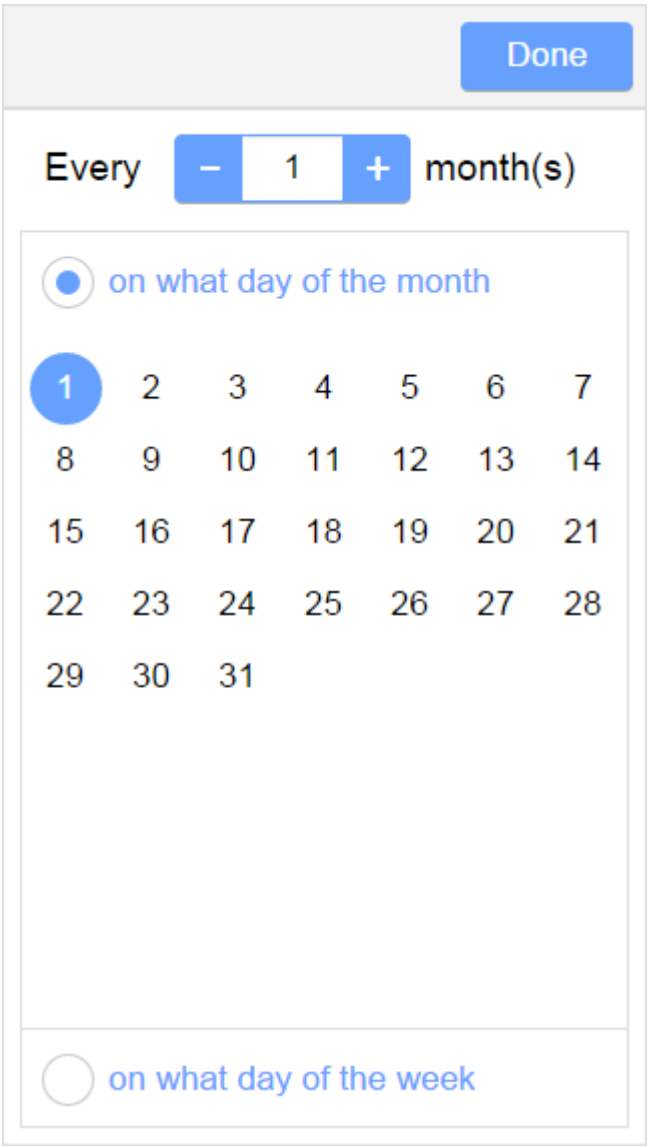
- **'daily'**



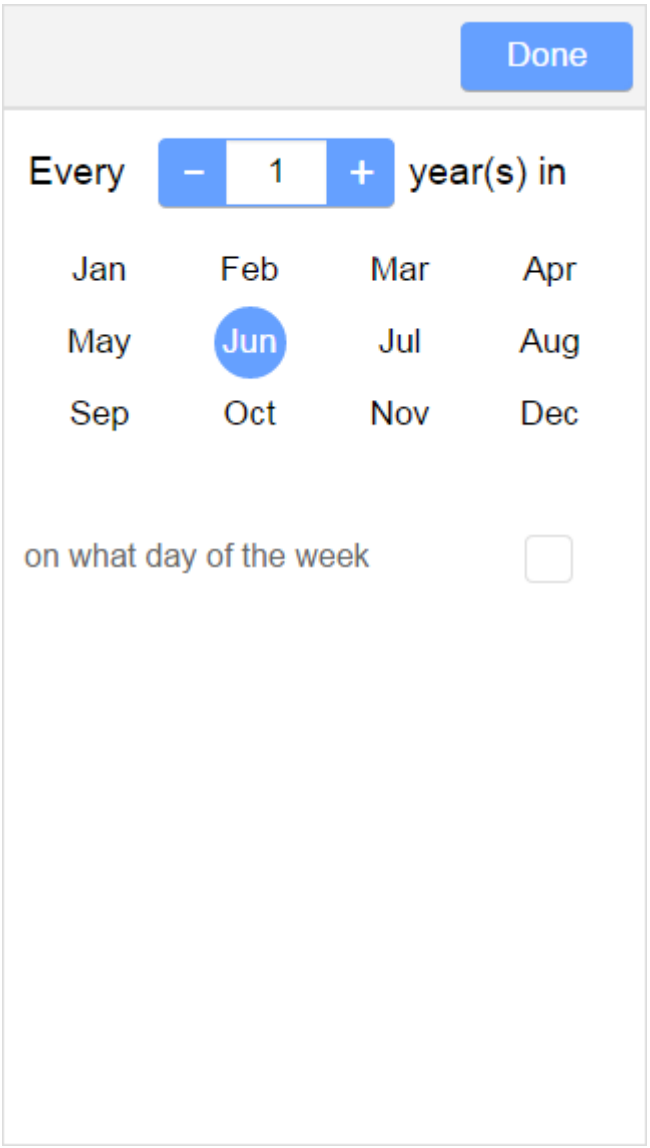
- **'weekly'**


Done
Every
1
week(s) on
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday

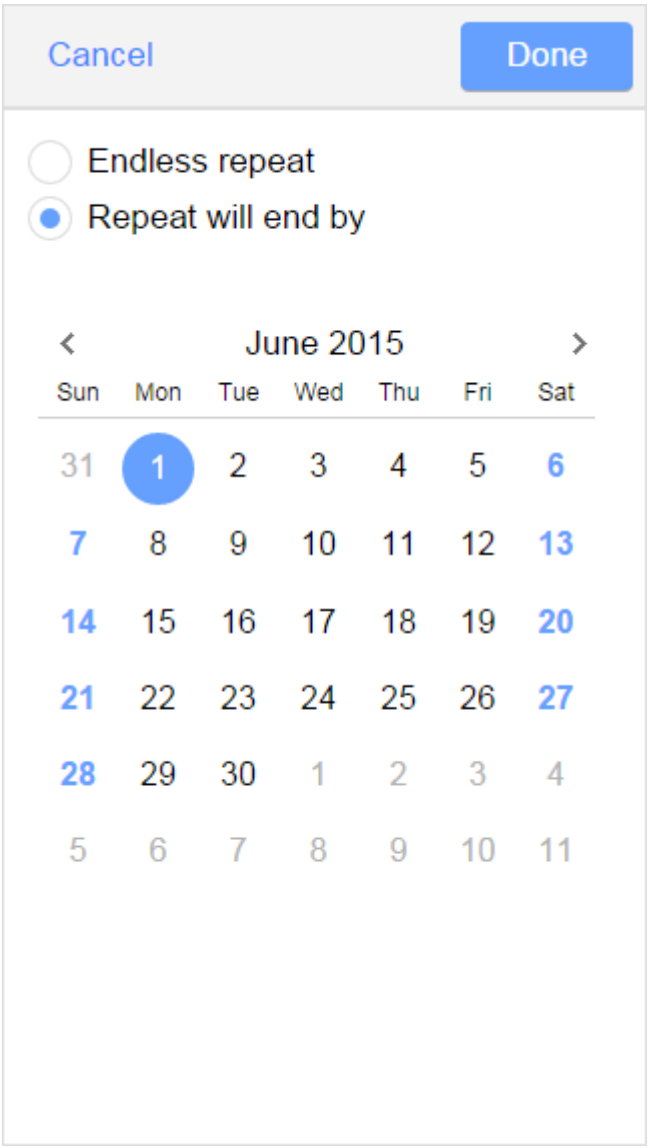
- **'monthly'**


Done
Every
1
month(s)
on what day of the month
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
on what day of the week

- **yearly'**



You can also set the date of **repetition's end**:



Customizing the 'Edit' form

The recurring 'Edit form' uses the same customization technique as the standard form does.

There's only a difference in the **'scheduler.config.form'** template. For recurring events it will look like:

```
scheduler.config.form = [
    {
        view: "text",
        label: scheduler.locale.labels.label_event,
        name: 'text',
        labelWidth: 90
    },
    {
        view: "datetext",
        label: scheduler.locale.labels.label_start,
        id: 'start_date',
        name: 'start_date',
        dateFormat: scheduler.config.form_date,
```

```
        labelWidth: 90
    },
    {
        view: "datetext",
        label: labels.label_end,
        id: 'end_date',
        name: 'end_date',
        dateFormat: scheduler.config.form_date,
        labelWidth: 90
    },
    {
        view: "checkbox",
        id: 'allDay',
        name: 'allDay',
        label: scheduler.locale.labels.label_allday,
        value: ,
        labelWidth: 100
    },
    {
        view: "rectext",
        label: scheduler.locale.labels.recurring.repeat,
        id: 'rec_type',
        name: 'rec_type',
        readonly: true,
        labelWidth: 90
    },
    {
        view: "textarea",
        label: scheduler.locale.labels.label_details,
        id: 'details',
        name: 'details',
        height: 110,
        labelWidth: 90
    },
    {
        view: "text",
        hidden: true,
        id: 'event_length',
        name: 'event_length'
    }
];
```

Dynamic Loading

By default, Scheduler loads all data at once.

It may cause some problems when you are using big event collections. In such situations you may use the dynamic loading mode and load data by parts. Date range depends on the chosen mode of loading.

To enable the dynamic loading, you should call the **setLoadMode** method before data loading:

```
scheduler.setLoadMode("month");

// load data
scheduler.load("some.php",)
```

As a parameter, the method takes the loading mode that defines the size of the data to load - **month** or **year**. For example, if you set the 'year' mode, the scheduler will request data just for the selected year and load remaining ones on demand.

The generated requests look like:

```
some.php?from=DATEHERE&to=DATEHERE
```

- where **DATEHERE** - a valid date value in the format defined by the configuration property **scheduler.config.dyn_load_date** ("%Y-%m-%d").

If you are using Data Connector at the server side, you don't need to do any additional server-side operations to parse the data:

```
scheduler.setLoadMode("month");
// if you are using Scheduler Connector,
// you need to set the second parameter in "load" method
scheduler.load("some.php","scheduler")
```

RELATED SAMPLE: Dynamic loading (connector)

Advanced

Changing Scheduler's Structure

The mobile scheduler has a multiview-based architecture. For details see the related article of documentation - Building App Architecture.

Please remember that you can customize just toolbars of the scheduler.

So in this article we will concentrate on toolbars. Below you can find all the available toolbars and their default definitions.

For detailed information on toolbar see the appropriate article of Webix documentation - Toolbar.

Multiview

The main **multiview** of the scheduler contains 3 views:

- **Selected event view**;
- **Edit form**;
- **Start(end) date edit form**;
- Sub-multiview (supplemented by the bottom toolbar):
  - **Day**;
  - **Week**;
  - **Month**.

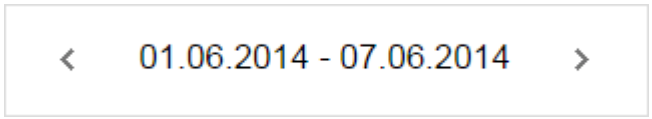
'Day' view



In the “Day” view, the toolbar contains navigation buttons and the label displaying the selected date.

```
scheduler.config.day_toolbar = [
    {view:'label', id:"prev", align:"left",
      label:"<div class='webix_cal_prev_button'><div></div></div>"},
    {view:'label', id:"date", align:"center", width:200},
    {view:'label', id:"next", align:"right",
      label:"<div class='webix_cal_next_button'><div></div></div>"}
];
```

'Week' view




```
scheduler.config.week_toolbar = [
    {view:'label',name: 'prevWeek', id:"prevWeek", width: 40,
align:"left",
      label:"<div class='webix_cal_prev_button'></div>"},
    {view:'label',id:"weekTitle",align:"center"},
    {view:'label',id:"nextWeek", align:"right",
      label:"<div class='webix_cal_next_button'></div>",width: 40}
];
```

'Month' view

In the “Month” view the toolbar is a part of calendar integrated to the view (not an individual component). So if you want to customize this toolbar, please, refer to the related Calendar documentation.

'Selected Event' view



Here, the toolbar contains 2 buttons: “Back” and “Edit” (the buttons are hidden in the read-only mode).

```
scheduler.config.selected_toolbar = [
{view:'label',width:scheduler.xy.icon_back,css:"cancel",name:"back",id:"ba
ck",
      align:"center", label:scheduler.locale.labels.icon_back},
    {view:'button', inputWidth:scheduler.xy.icon_edit, name:"edit",
id:"edit",
      align:"right", label:scheduler.locale.labels.icon_edit}
];
```

Start(end) date edit form



The toolbar contains 2 buttons: “Cancel” and “Done” and the label that displays which date is edited: **start** or **end** (the view is hidden in the read-only mode).

```
scheduler.config.date_toolbar = [
    {view:'label', width:scheduler.xy.icon_cancel,
name:"cancel_date",id:"cancel_date",
      css:"cancel",
align:"center",label:scheduler.locale.labels.icon_cancel},
```

```
    {view:'label',id:"datetype",align:"center"},
    {view:'button', width: scheduler.xy.icon_done, name:"done",
id:"done",align:"right",
      label:scheduler.locale.labels.icon_done}
];
```

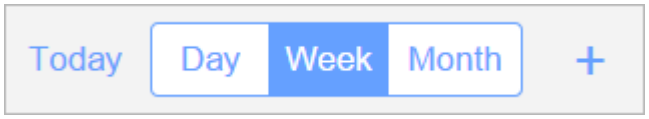
Edit form



The toolbar of the "Edit" form has 2 buttons: “Cancel” and “Save”.

```
scheduler.config.form_toolbar = [
    {view:'label', width:scheduler.xy.icon_cancel, name:"cancel",
id:"cancel",css:"cancel",align:"center",label:scheduler.locale.labels.icon
_cancel},
    {view:'button', inputWidth:scheduler.xy.icon_save, name:"save",
      id:"save",align:"right",label:scheduler.locale.labels.icon_save}
];
```

Main bottom toolbar



The main scheduler toolbar contains 3 controls:

- “Today” button
- a tabbar with the “Day”, "Week" and “Month” tabs
- “Add” button

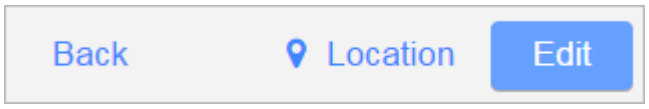
```
scheduler.config.bottom_toolbar = [
    {view:"label",id:"today", name:"today",
label:scheduler.locale.labels.icon_today,
      inputWidth:scheduler.xy.icon_today,
align:"center",width:scheduler.xy.icon_today+6},
    {view:"segmented", id:"buttons", value:"week", align:"center",
      multiview:true, options:[
        {id:"day", value:scheduler.locale.labels.day_tab,
width:scheduler.xy.day_tab},
        {id:"week", value:scheduler.locale.labels.week_tab,
width:scheduler.xy.week_tab},
        {id:"month",value:scheduler.locale.labels.month_tab,
width:scheduler.xy.month_tab}
     ]},
    {view:"label", css:"add",name:"add",id:"add",
```

```
align:"center",label:" + ",
      width:45},
    {view:"label", label:"",width:45, batch:"readonly"}
];
```

Adding new elements to the toolbar and event handling

If you want to add some custom button to one of the mentioned toolbars, you need to redefine the structure of that toolbar. Clicks on the default buttons are handled automatically. But for new buttons you should define the “click” event handlers.

Let's assume you want to add a new button "Location" to the **Selected Event** view.



The definition of your new toolbar can be implemented as follows:

```
scheduler.config.selected_toolbar = [
    {view:'label', width:70, css:"cancel",name:"back",id:"back", label:
"Back"},
    {view:'label', inputWidth:120, css:"cancel", id:"location",
align:"right",
      label:"<span class='webix_icon fa-map-marker'></span>Location",
click:"showLocation"},
    {view:'button', inputWidth: 70, width:90, name:"edit", id:"edit",
label: "Edit"}
];
```

To show the event location on a button click, you need to define some handler function, for example:

```
function showLocation(){
    var eventId = $$("scheduler").getCursor();
    var location = $$("scheduler").getItem(eventId).location;
    webix.alert(location);
}
```

In the function you could be confused by the method getCursor. This method is a part of 'binding' functionality that's used by the scheduler. You can read about the mentioned functionality in the related Webix documentation - Data Binding and Syncing.

RELATED SAMPLE: Custom form

Accessing elements of the scheduler

The scheduler includes different components: lists, toolbars, a form, etc. To access any of them you can use the **$$(schedulerId).$$(componentId)** construction.

The ids of the toolbars' buttons can be found above, in the toolbars' descriptions. And here are the ids of components from the multiview of the scheduler:

- 'event' - the 'Selected Event' view
  - 'eventBar' - the top toolbar
  - 'eventTemplate' - the template with the data of an event
- 'form' - the 'Edit form' view
  - 'editBar' - the toolbar
  - 'editForm' - the form with the details of an event
- 'tabViews' - the sub-multiview containing **Day**, **Week** and **Month** views
  - 'bottomBar' - the bottom toolbar
  - 'day' - the 'Day' view
    - 'dayBar' - the top toolbar
    - 'dayList' - the list with events
  - 'week' - the 'Week' view
    - 'weekBar' - the top toolbar
    - 'weekEventsList' - the list with events
  - 'month' - the 'Month' view
    - 'calendar' - the calendar
    - 'calendarDayEvents' - the list with events of the selected day

For example, if you want to show the 'Month' view initially, you need to call the show() method for the 'month' view and select the 'month' button of the tabbar control on the bottom toolbar:

```
$$('scheduler').$$('month').show();
$$('scheduler').$$('buttons').setValue('month');
```

- where 'month' and 'buttons' are the ids of the appropriate components of the scheduler.

For details on the mentioned components you can read the article UI Components.

A custom view

It's possible to add a custom view to the main multiview of the scheduler. All you need to do is to add view configuration to the **scheduler.config.views** array. Beware, view configuration should go before the code line with scheduler initialization.

For example, if you'd like to add 'list' view, it can be done as follows:

```
scheduler.config.views.push({
    view:"list",
    id:"mylist",
    ...
```

```
});
```

Handling Events

Attaching Event Handler

The user can add any user-defined handler to any of the available events. To do this, the user can use the attachEvent method with the following parameters:

- evName - name of the event;
- evHandler - user-defined event handler.

```
webix.attachEvent(evName, evHandler);
```

Several handlers can be attached to one and the same event, and all of them will be executed. The names of the events are case-insensitive.

Detaching Event Handler

There is a simple way of removing an event-handler:

```
webix.detachEvent(id); // unique id of the event handler
```

Add, Delete and Update Event Handling

To handle CRUD operations you can set onAfterAdd, onAfterDelete and onStoreUpdated event handlers for Scheduler's datastore. You can get datastore by Scheduler's "data" property.

onAfterAdd

fires after an event has been added into scheduler

```
$$("scheduler").data.attachEvent("onAfterAdd",function(itemId){
    var eventObj = this.getItem(itemId);
    // your code here
});
```

- itemId - {string} the id of the newly added event

onAfterDelete

fires after an event has been deleted

```
$$("scheduler").data.attachEvent("onAfterDelete",function(itemId){
    // your code here
```

```
    return true;
});
```

- itemId - {string} the id of the deleted event

onStoreUpdated

fires when the data store has been updated

```
$$("scheduler").data.attachEvent("onStoreUpdated",function(itemId,item,ope
ration){
    if(operation == "update"){
       // your code here
    }
    return true;
});
```

- itemId - {string} the id of the event object
- item - {object} the event object
- operation - {string} type of the operation

Handling event choosing

onBeforeEventShow

To process the behavior of the component on choosing an event, you can use the onBeforeEventShow event.

This event is fired by Scheduler when an event is chosen in one of the calendar lists. If this event returns false, a form with event details won't be shown:

```
$$("scheduler").attachEvent("onBeforeEventShow",function(eventId){
   // your code
   return true;
});
```

General How-Tos

How to add an event?

Scheduler data are loaded via parse or load methods. If you need to add a single event you can use the add method and pass event object in it:

```
$$("scheduler").add(
   {
     id:3,
     start_date:"2015-06-01 09:00:00",
     end_date:"2015-06-01 10:00:00",
     text:"Meeting"
   }
);
```

How to remove an event?

Use the remove method to remove an event by its id:

```
$$("scheduler").remove(eventid);
```

How to change event properties?

To change event properties you need to do the following:

- get the event object
- change properties of the event object
- redraw the event

```
var event = $$("scheduler").getItem(eventId);
event.text = "Meeting";
$$("scheduler").refresh(eventId);
```

How to hide a button?

You can use the hide method to hide a certain view in Scheduler. Here you may find details about Scheduler structure.

For example, if you want to hide the "add" button, you can do the following:

```
$$("scheduler").$$("add").hide();
```

How to handle event choosing?

When an event is chosen in one of the lists, Scheduler fires the "onBeforeEventShow" event. If this event returns *false*, a form with event details won't be shown:

```
$$("scheduler").attachEvent("onBeforeEventShow",function(eventId){
   // your code
   return true;
});
```

Tips for dhtmlxScheduler's users

1) Webix Scheduler as well as dhtmlxScheduler allows defining date formats. The details are given in the article Date Formats.

2) Scheduler supports multi-day and recurring events, too.

3) Webix Scheduler also allows applying templates to its views. Read more in the article Specifying Templates.

4) Elements' sizes are customizable, too. This feature is described in the Sizing Elements article.

5) If you are going to use dhtmlxConnector to load and save data, you should do the following:

- define "scheduler" as "DataDriver" in the load method:

```
...
$$("scheduler").load("data/events.php","scheduler");
...
```

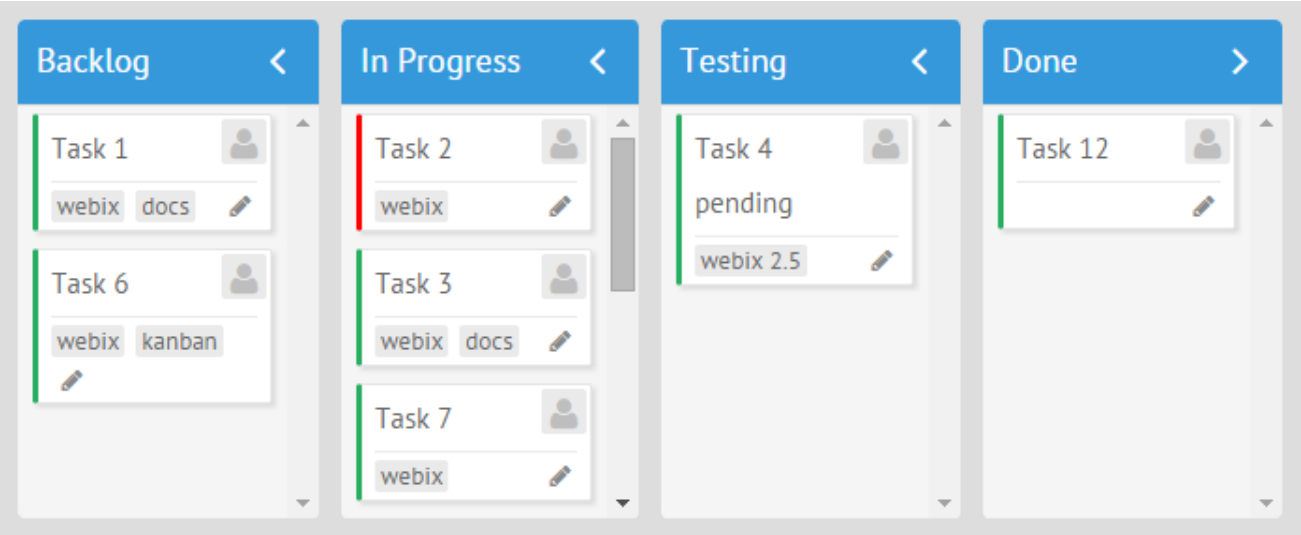
- set the **save** property of the scheduler to apply connector for data saving:

```
...
webix.ui({
    view: "scheduler",
    id: "scheduler",
    save: "connector->data/events.php"
});
...
```

Kanban Board

Webix Kanban board is a tool that is intended for viewing work in progress and controlling the flow of work. With this widget, your team will be aware of what each of the members should be working on.

Webix Kanban board is rather flexible and can have any number of columns and rows. It's also easily customizable via HTML Templates. Fully written in JavaScript and CSS this component can be integrated in any web application or web page.



Configuring Kanban

- Creating Kanban Board on a page
- Loading and Saving Data
- Data Visualization
- Customizing Kanban's structure

Discusses the main points of Kanban Board initialization and customization.

Working with Kanban

- Adding and Deleting Items
- Working with Data
- Editing Items
- Handling Events
- API Reference
- General How-Tos

Describes possible operations with Kanban Board and its elements.

Creating Kanban Board on a page

In order to start using Kanban Board you need to initialize it on the page. It requires 2 simple steps:

Include the necessary JS and CSS files into HTML file.

Call the object constructor.

Included files

There are 4 files that you need to include:

```
<!-- js files -->
<script src="codebase/webix.js" type="text/javascript"></script>
<script src="codebase/kanban/kanban.js" type="text/javascript"></script>
<!-- css files -->
<link rel="stylesheet" href="codebase/webix.css" type="text/css"
charset="utf-8">
<link rel="stylesheet" href="codebase/kanban/kanban.css" type="text/css"
charset="utf-8">
```

Initialization

Use the following code to initialize Webix Kanban Board:

```
//webix.ready() function ensures that the code will be executed when the
page is loaded
webix.ready(function(){
    //object constructor
    webix.ui({
        view:"kanban",
        type:"space",
        //the structure of columns on the board
        cols:[
            { header:"Backlog",
                body:{ view:"kanbanlist", status:"new" }},
            { header:"In Progress",
                body:{ view:"kanbanlist", status:"work" }},
            { header:"Testing",
                body:{ view:"kanbanlist", status:"test" }},
            { header:"Done",
                body:{ view:"kanbanlist", status:"done" }}
        ],
        //url to the file with data collection
        url: "tasks.php"
    });
});
```

RELATED SAMPLE: Basic Init of Kanban Board

Loading and Saving Data

Data can be loaded from different sources. However, Kanban stores data items as json objects:

```
{
    id: 1,
    status: "new",
    text: "Kanban board documentation",
    tags: "kanban,webix",
    $index: ,
    ...
}
```

Each data item possesses a unique **id** property and a set of other properties:

- status - {string} the item's status, defines the list in which an item will be shown (obligatory)
- text - {string} the item's (task's) title
- tags - {string} a comma-separated list of tags
- $index - {numeric} the task's position in the list that the item relates to

Kanban Board supports all data formats of the Webix library.

There are 2 main ways you can specify data source for Kanban Board:

To define the data set in the object constructor.

To use the url property.

Both ways lead to one and the same result and don't have any specificity.

Loading from inline dataset

If you want to specify the data set directly on the page, you can use the data property.

```
webix.ready(function(){
    webix.ui({
        view:"kanban", type:"space",
        cols:[
            { header:"Backlog",
                body:{ view:"kanbanlist", status:"new" }},
            ...
        ],
        //data collection                                    data:
[                                                  { id:1,
text:"Task 1", status: "new"},                                { id:2,
```

```
});
});
```

Loading from a data file

To load data from a file, you can use the url property. The url refers to the JSON object that will be loaded into the board after its initialization.

```
webix.ready(function(){
    webix.ui({
        view:"kanban",
        type:"space",
        cols:[
            { header:"Backlog",
                body:{ view:"kanbanlist", status:"new" }},
            ...
        ],
        //url to the file with data collection                    url:
"tasks.php"                                    });
});
```

Saving data

There are a lot of ways to save data in Webix. You can find the detailed description in the Data Saving article.

For example, in the case of using DataProcessor you can set the property:

```
webix.ready(function(){
    webix.ui({
        view:"kanban", type:"space",
        cols:[
            { header:"Backlog",
                body:{ view:"kanbanlist", status:"new" }},
            ...
        ],
        url: "tasks.php",                                           save:
"../tasks_save.php"                                 });
});
```

RELATED SAMPLE: Load and Save

More information about Kanban data loading you can find in DataStore docs.

Data Visualization

You can represent various HTML-elements in the Kanban Board.

Data items of Kanban Board are sorted by their status and displayed in the corresponding lists.

Each KanbanList allows configuring dimensions and templates of its items. It is implemented in the type of KanbanList:

```
// define a type for KanbanList views with a set of templates
webix.type(webix.ui.kanbanlist,{
    name: "cards",   // type name      icons:[{id: "edit", icon:
"edit"}],
    templateBody: function(obj){
        return (obj.img?"<img src='imgs/"+obj.img+"'/>":"")+obj.text;
    }
});

// create a Kanban board
//and apply "cards" type for lists
webix.ui({
    view:"kanban",
    cols:[
        {
           header:"Backlog",
           body:{
              view:"kanbanlist",
              status:"new",
              type: "cards"  // set type            }
        },
        ...
    ]
});
```

RELATED SAMPLE: Templates

You can set the following properties in the type:

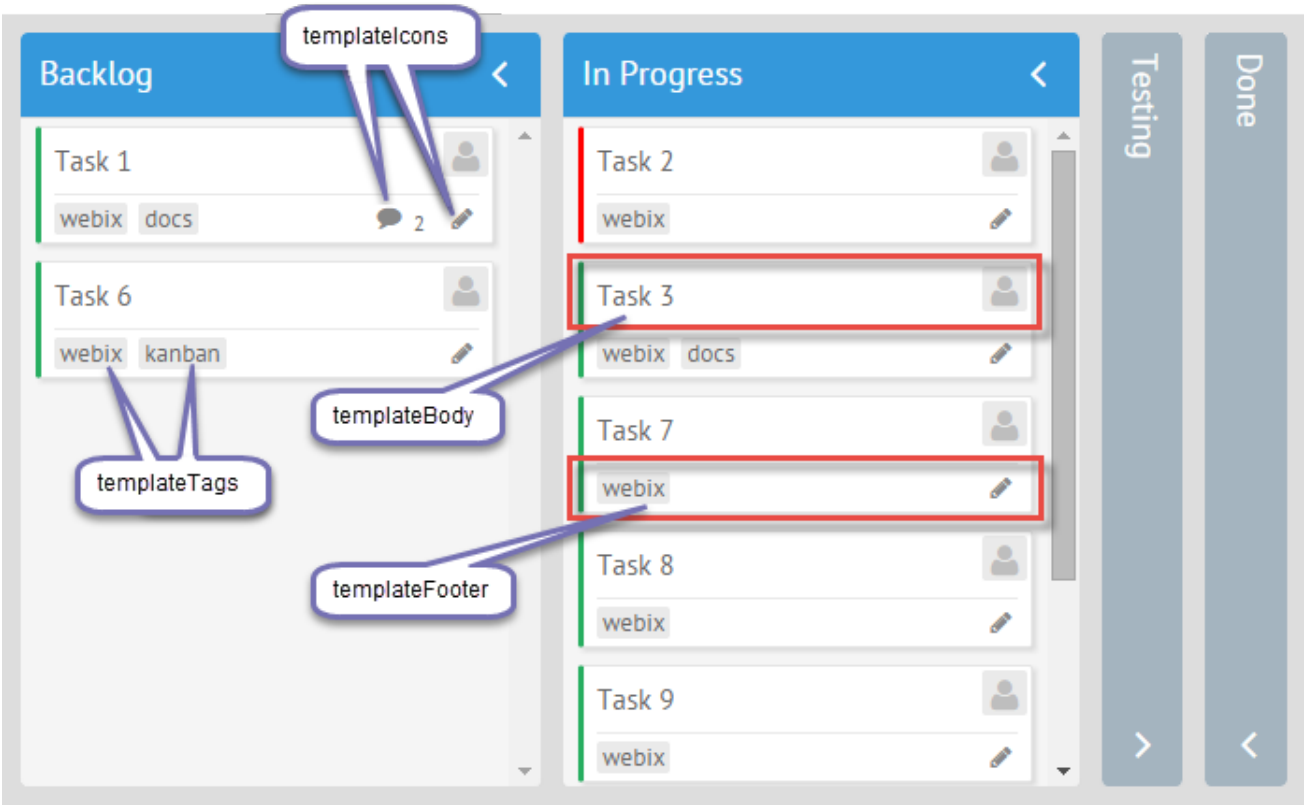
- 'width' - the width of a list item ('auto' or fixed number in pixels),
- 'height' - the height of a list item ('auto' or fixed number in pixels),
- 'template' - data template that defines inner html for each item. The default definition of this property applies many other templates: icons, text, avatar,
- 'templateBody' - template for main content of items. It is applied in the main 'template' and displays an

item's text by default,

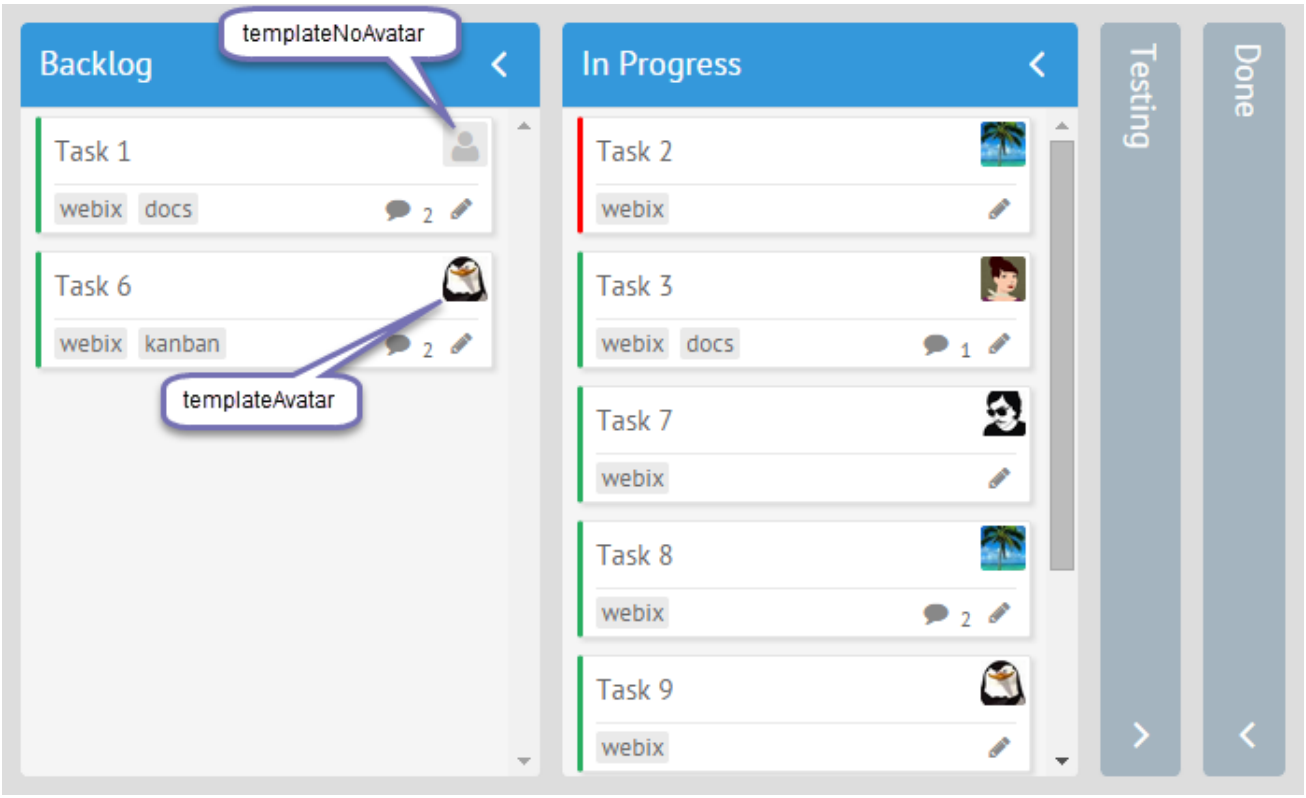
- 'templateFooter' - template for content below the item's body (it is applied in the main 'template' and displays an item's tags and icons),
- 'icons' - the array of icons. The icons are taken from the Font Awesome collection
- 'templateIcons' - template for icons representation (is applied in the 'templateFooter')
- 'templateTags' - template for tags representation (is applied in the 'templateFooter')
- 'tagDelimiter' - string that defines tags delimiter in datasource (',' by default)
- 'templateAvatar' - template for an item's avatar (an assignee's photo or other data)
- 'templateNoAvatar' - this template is applied if the 'templateAvatar' doesn't set html content for the element that shows an assignee's information.

A template can be specified either in the format of string or in the format of function. Below you'll find the description of all available templates.

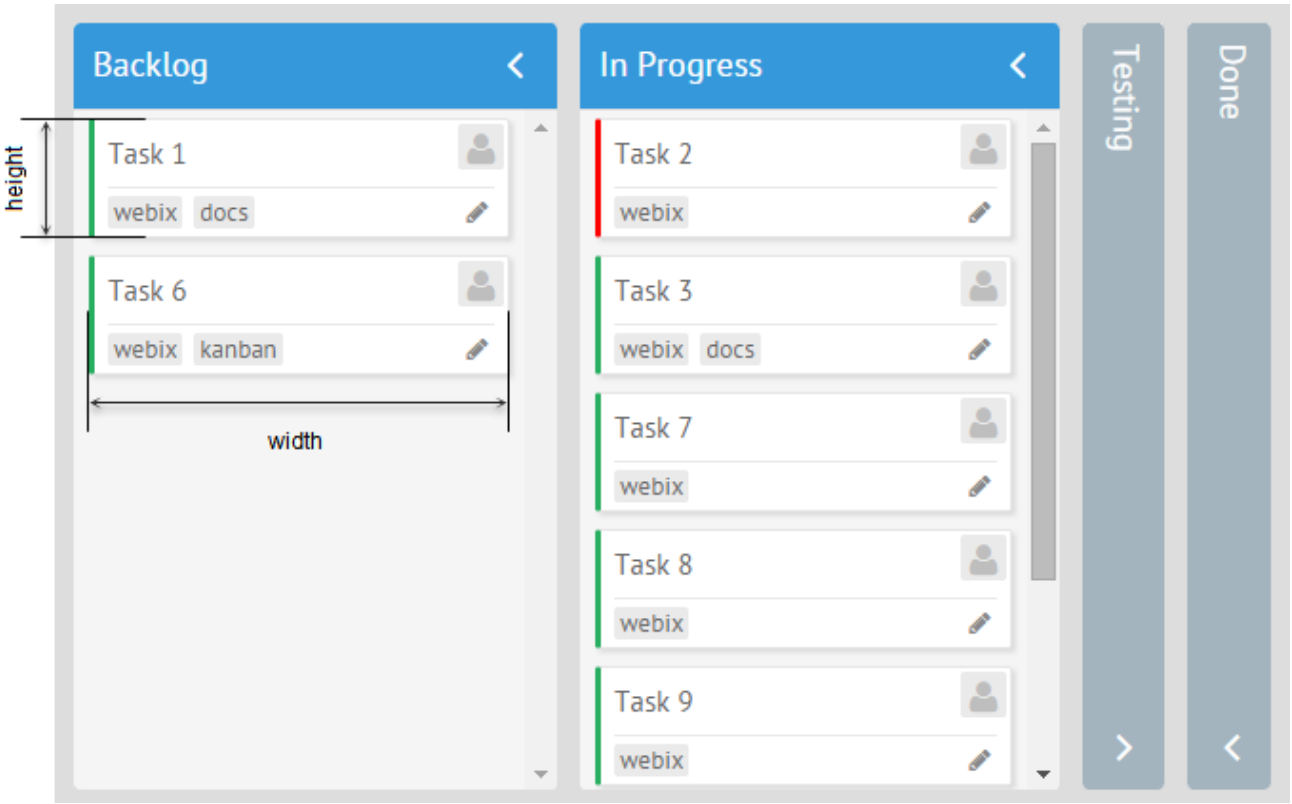
Main templates



Avatar templates



Width and height of an item



width

Specifies the width of an item in the list.

```
{
   width:"auto"
```

```
}
```

"auto" is the default width value. However, you can set a fixed value and show more than one item in list rows:

```
webix.type(webix.ui.kanbanlist,{
   name: "cards",
   width: 120
   ...
});
```

You can read more here.

height

Specifies the height of an item in the list.

```
{
   height:"auto"
}
```

the 'height' property as well as 'width' can be defined as a fixed value.

template

Defines inner html for each item. The default definition of this property applies many other templates: icons, text, avatar.

```
template: function(obj, t){
   var avatar, content;
   avatar = t.templateAvatar(obj,t)||t.templateNoAvatar(obj,t);
   avatar = '<div class="webix_kanban_user_avatar">'+avatar+'</div>';
   content = '<div
class="webix_kanban_body">'+t.templateBody(obj,t)+'</div>';
   content += '<div class="webix_kanban_footer">'+t.templateFooter(obj,t)+
'</div>';
   var style = 'style="border-left-color:'+obj.color+'"';
   return avatar+'<div class="webix_kanban_list_content"
'+style+'>'+content+'</div>';
}
```


Pay attention that you shouldn't redefine the main **template** in order not to lose data. Otherwise, you'll have to specify data collection by yourself.

templateBody

Defines the main content of items, displays an item's text by default

```
webix.type(webix.ui.kanbanlist,{
    name: "cards",
    // displays images and text property in item content
    templateBody: function(obj,common){
        var html = "";
        if(obj.image)
            html += "<img class='image'
src='../common/imgs/"+obj.image+"'/>";
        html += "<div>"+obj.text+"</div>";
        return html;
    },
    ...
});
```

templateFooter

Defines the content below the item's body, displays an item's tags and icons

```
{
    templateFooter: function(obj,common){
        var tags = common.templateTags(obj,common);
        return (tags?tags:" ")+ common.templateIcons(obj,common);
    },
    ...
}
```

icons

Specifies an array of icons for Kanban Board

You need to specify icons array in the type for Kanban Lists. In the definition of each icon you can set the following properties:

- id - icon id,
- icon - name of a font icon in Font Awesome collection,
- tooltip - title for an icon element,
- show - function that takes an item object as a param and returns true/false value -to show/hide an icon for a certain item
- template - text that will be displayed next to the icon
- click - onClick event handler for the icon

```
webix.type(webix.ui.kanbanlist,{
    name: "cards",
    icons:[
```

```
        {
            id: "comments",
            tooltip: "Comments",
            icon:"comment-o",
            show: function(obj){
                return !!obj.comments
            },
            template:"#comments.length#",
            click: function(e,id){
                var item = this.getItem(id);
                ...
            }
        },
        {
            id: "edit",
            icon:"edit",
            tooltip: "Edit Task",
            click: function(e,id){
                var item = $$("myBoard").getItem(id);
                ...
            }
        }
    ]
});

webix.ui({
    view:"kanban",
    ...
    cols:[
        {
            header:"Backlog",
            body:{
                view:"kanbanlist",
                status:"new",
                type: "cards"
            }
        },
        ....
    ]
});
```

templateIcons

Defines icons representation

```
templateIcons: function(obj,common){
    var icons = [], icon = null, id, html, text;
    for (var i = ; i < common.icons.length; i++){
        icon = common.icons[i];
        if(!icon.show || icon.show(obj)){
```

```
            id = icon.id||icon.icon;
            html = '<span webix_icon_id="'+id+'"
class="webix_kanban_icon">';
            html += '<span class="fa-'+icon.icon+' webix_icon"></span>';
            if(icon.template){
                text = webix.template(icon.template)(obj);
                html += '<span
class="webix_kanban_icon_text">'+text+'</span>';
            }
            html += '</span>';
            icons.push(html);
        }
    }
    return '<div  class="webix_kanban_icons">' +icons.join(" ")+'</div>';
},
...
```

templateTags

Defines tags representation

```
templateTags: function(obj,common){
    var html = '';
    if(obj.tags){
        var tags = obj.tags.split(common.tagDemiliter);
        for (var i = ; i < tags.length; i++){
            html += '<span class="webix_kanban_tag">'+tags[i]+'</span>';
        }
    }
    return '<div class="webix_kanban_tags">'+(html||" ")+'</div>';
}
```

templateAvatar

Specifies an item's avatar

```
webix.type(webix.ui.kanbanlist,{
    name: "cards",
    // avatar template
    templateAvatar: function(obj){
        var html = '';
        if(obj.personId){
            html = '<img class="avatar"
src="../common/imgs/'+obj.personId+'.jpg" />';
        }
        return html;
    },
    ...
});
```

templateNoAvatar

Applied if the 'templateAvatar' doesn't set html content for the element that shows an assignee's information

```
{
    templateNoAvatar: "<span class='webix_icon fa-user'></span>"
}
```

RELATED SAMPLE: Templates

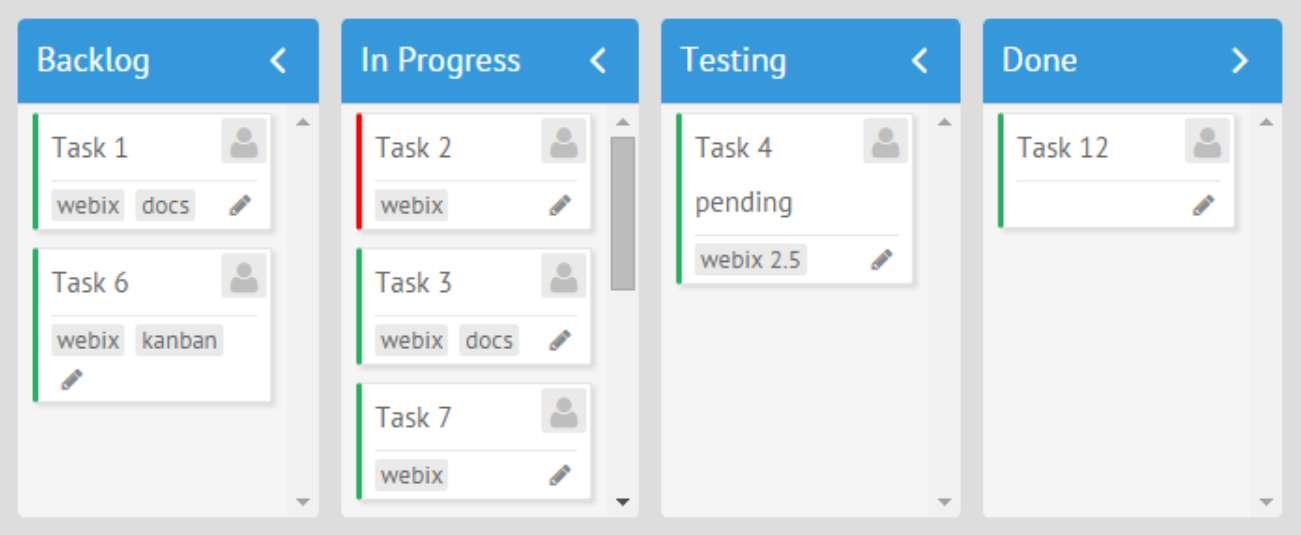
Customizing Kanban's structure

Kanban Board has a layout structure. It means that it consists of rows and columns that can be combined in different ways.

A column has a header with its name and a body which can contain either a KanbanList view or other rows- or cols-layouts (to create a more complex structure).

In order to specify rows on the board the property **rows** is used and the **cols** property is used to define columns.

A simple Kanban Board may look as:



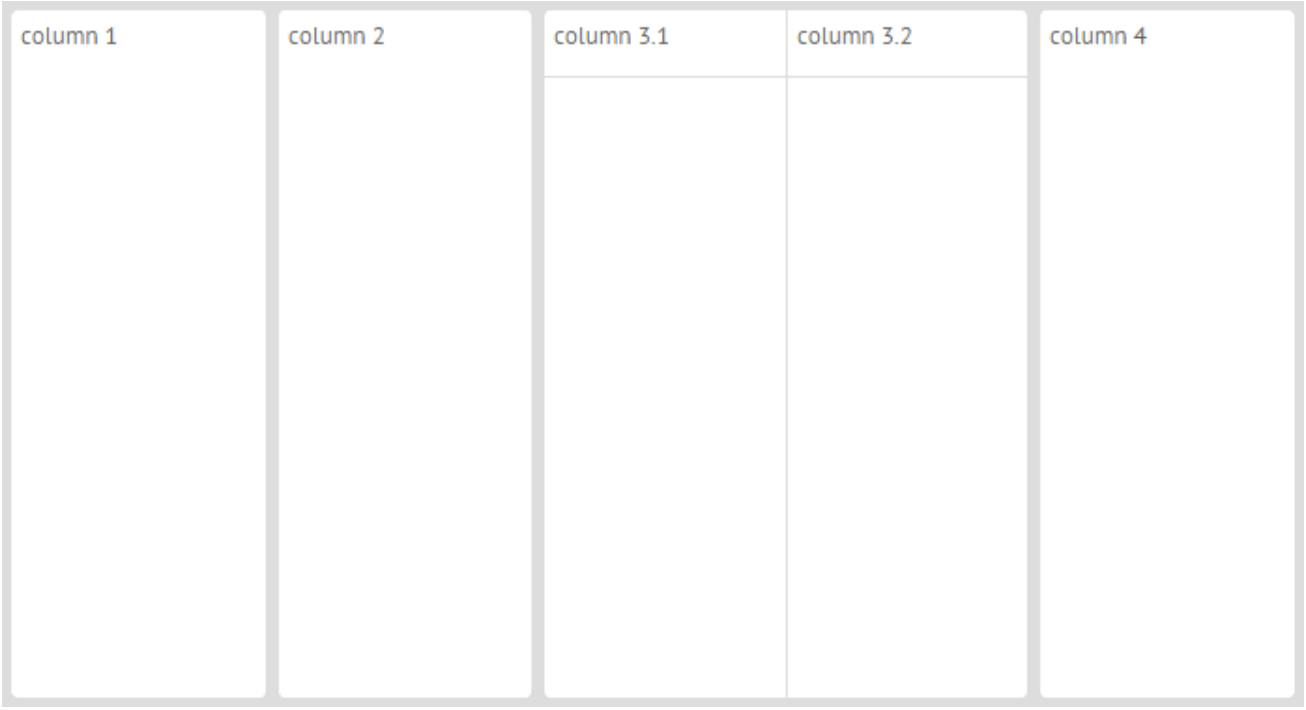
In case your working process consists of many stages, you may need to make a more complex Kanban board to reflect the full flow of work.

Splitting columns

Let's imagine that you want to make a Kanban Board consisting of 4 columns, besides:

- the 1st, 2nd and 4th columns contain KanbanLists
- the 3rd column has a cols-layout inside (2 columns)
- each of the columns has 2 rows:
  - the upper rows contain subheaders
  - the lower rows have lists inside

The scheme of the board will look as:



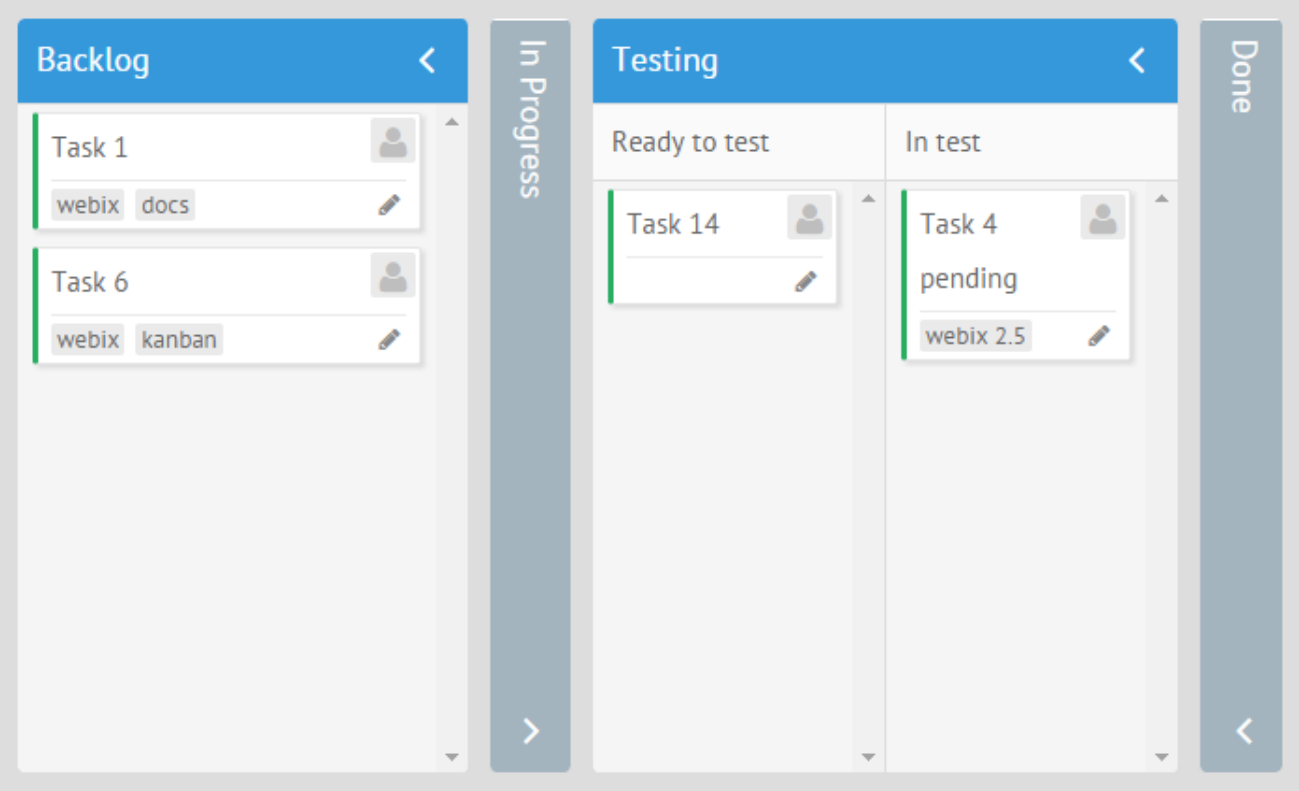
To implement this scheme we'll use the code below:

```
webix.ready(function(){
    webix.ui({
        view:"kanban", type:"space",
        cols:[
            {
                header:"Backlog",
                body:{ view:"kanbanlist", status:"new" }
            },
            {
                header:"In Progress",
                body:{ view:"kanbanlist", status:"work" }
            },
            {
                header:"Testing", gravity: 1.3,
                body:{

                    cols:[
                            {
                                rows:[
                                        { template:"Ready to test",
height:40},
                                        { view:"kanbanlist",
status:"ready"}
                                ]
                            },
                            {
                                rows:[
                                        { template:"In test", height:40},
```

```
                                        { view:"kanbanlist",
status:"test"}
                                ]
                            }
                        ]
                }},
            {
                header:"Done",
                body:{ view:"kanbanlist", status:"done" }
            }
        ],
            data:base_task_set
    });
});
```

The result will be the following:



RELATED SAMPLE: Layout

Creating a complex structure

For example, you need to create a Kanban board's structure consisting of 3 columns, besides:

- the 1st and the 3rd columns contain lists
- the 2nd column has a rows-layout:
    - the upper row has a list inside

- the lower row contains three columns with lists

The picture below illustrates the idea:



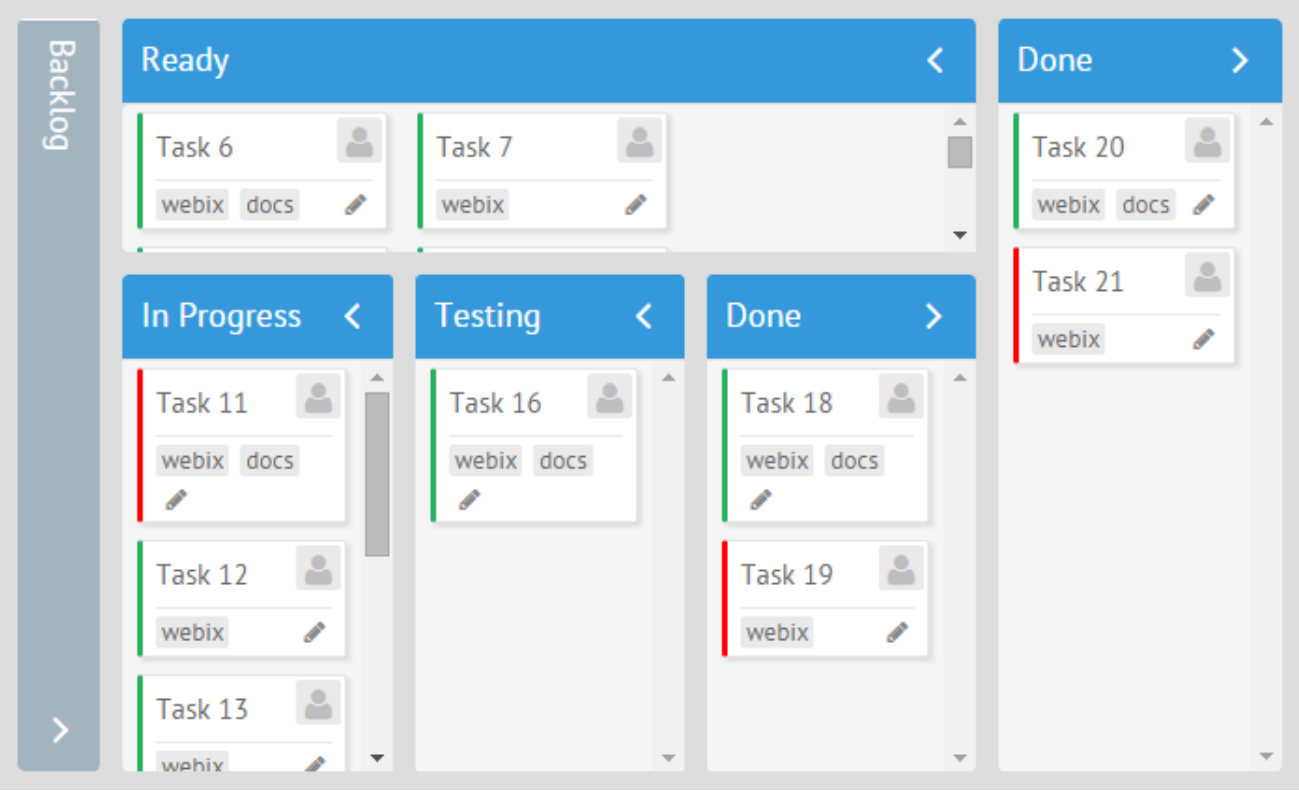
The code snippet below creates the Kanban Board described in the above-mentioned scheme:

```
webix.ready(function(){
    webix.ui({
        view:"kanban",
        type:"space",

            cols:[
                {
                    header:"Backlog",
                    body:{ view:"kanbanlist", status:"new"}
                },
                {
                    header:"Ready", gravity: 3,
                    body:{
                        type: "wide",
                        rows:[
                            {
                                view:"kanbanlist",
                                status:"ready",
                                xCount: -1,
                                type:{ width: 150 },
                                gravity: 0.3
                            },
                            {
                                cols:[
                                    {
                                        header:"In Progress",
```

```
                                        body:{ view:"kanbanlist",
status:"work"}
                                    },
                                    {
                                        header:"Testing",
                                        body:{ view:"kanbanlist",
status:"test"  }
                                    },
                                    {
                                        header:"Done",
                                        body:{ view:"kanbanlist",
status:"done"  }
                                    }
                                ]
                            }
                        ]
                    }
                },
                { header:"Done",
                    body:{ view:"kanbanlist", status:"complete" }}
            ],
            data:task_set
    });
});
```

You can see the result of the code's work in the following picture:



As you can see, tasks in the upper row of the 2nd column are placed next to each other.

The following section of the article gives the details on creating such a configuration.

RELATED SAMPLE: Rows and Columns

Changing structure of Kanban lists

By default, all kanban lists have items that are aligned one by one in a column.

However, in some cases we need to show more than one item per row (like in the above example).

Such a configuration can be created by means of setting a fixed width for list items and resetting the **xCount** property (-1 value).

```
webix.ready(function(){
    webix.ui({
        view:"kanban",
        type:"space",
            {
                header:"Ready", gravity: 3,
                body:{
                    type: "wide",
                    rows:[
                        {
                            view:"kanbanlist",
                            status:"ready",
                            xCount: -1,
type:{ width: 150 }                                 }
                    ]
                }
             },
            data:task_set
    });
});
```

Adding and Deleting Items

Adding and deleting can be implemented in the form of events that fire on user actions. You can set handlers on button clicks that will call the corresponding methods: add or remove.

Adding new items

Basic syntax will look as follows:

```
// get random id
var itemId = webix.uid();

// add an item to Kanban board
$$("myBoard").add({
    id: itemId,
    text: "Kanban docs",
    status: "new",
    ...
});
```

An added item should contain the 'status' property in order to be added in one of canban lists.

RELATED SAMPLE: Adding New Items

You can add an empty item and then open the edit form for it, giving a user the possibility to set necessary values.

Deleting items

To remove an item you should apply the remove() method to it:

```
// delete an item
$$("myBoard").remove(itemId);
```

An example of deleting a selected item is given below:

```
var itemId = $$("myBoard").getSelectedId();
if(itemId)
   $$("myBoard").remove(itemId);
else
   webix.message("No item is selected!");
```

All items can be removed from Kanban board via the clearAll mehtod:

```
$$("mylist").clearAll();
```

More information about adding and removing items you can find here.

RELATED SAMPLE: Adding New Items

Working with Data

Kanban board is based on DataStore, non-hierarchical (inline) store. That's why it supports all the DataStore API, including methods, events and properties.

In this chapter you'll find the description of main Kanban features inherited from DataStore that allow managing data. Specific Kanban API is listed in the articles API Reference and Handling Events.

Getting the item object

To get the data item object, you should use the getItem method:

```
webix.ui({
    view:"kanban",
    id: "myBoard",
    ...
    data:[
        {id:"task1", text:"Drag-n-drop support", status: "new", $order: },
        {id:"task2", text:"Uploader for the form", status: "work", $order:
},
        ...
    ]
});

//'task' variable will contain an object of the related kanban record
var task = $$("myBoard").getItem("task1");

//you can access data members directly
var title = task.text;//-> "Drag-n-drop support"
```

The getItem() method returns the object of the data item that possesses a unique **id** and a set of other properties:

- status - {string} the item's status, defines the list in which the item will be shown (**obligatory**)
- text - {string} the item's(task's) title
- tags - {string} a comma-separated list of tags (optional)
- $index - {numeric} the task's position in the list that the item relates to (optional)

Checking the existence of an item

To check whether a specific item exists in the data set, you should use the exists method:

```
var myBoard = webix.ui({
    view:"kanban",
    ...
```

```
    data:[
        {id:"task1", text:"Drag-n-drop support", status: "new", $order: },
        {id:"task2", text:"Uploader for the form", status: "work", $order:
}
    ]
});

myBoard.exists("task1");// -> true
myBoard.exists("task2222");// -> false
```

Changing data properties of an item

To change the current value of some item's property, you should use the following technique:

```
var task = myBoard.getItem("task1");
task.text = "Ticket 225";

myBoard.refresh("task1");
//or
myBoard.updateItem("task1", task);
```

Methods refresh and updateItem lead to one and the same result and don't have any specificity.

Iterating data items

To iterate through the collection of data items, you should use the each method of the DataStore class.

```
var myBoard = webix.ui({
    view:"kanban",
    ...
    data:[
        {id:"task1", text:"Drag-n-drop support", status: "new", $order: },
        {id:"task2", text:"Uploader for the form", status: "work", $order:
},
        ...
    ]
});

var titles =[];
myBoard.data.each(function(obj){ titles.push(obj.text); });
//-> titles=["Drag-n-drop support", "Uploader for the form"]
```

The code is equal to a standard 'for'-loop:

```
for(i = ; i<data.length; i++ ){
    titles.push(obj.text);
}
```

Filtering

If you need to show only a part of loaded data, depending on a certain rule, you can apply the filtering technique to Kanban board.

It is possible to filter data by a single property. Let's assume that we need to show only tasks assigned to a certain person:

```
var myBoard = webix.ui({
    view:"kanban",
    ...
    data:[
        {id:"task1", text:"Drag-n-drop support", status: "new", personId:
"1", ...},
        {id:"task2", text:"Uploader for the form", status: "work",
personId: "2", ...},
        ...
    ]
});

$$("myBoard").filter("#personId#","1");
```

Or you can set complex filtering rules. In this case you need to set a function that will return **true** or **false** value. If function returns **true** for an item, this item will be displayed:

```
var myBoard = webix.ui({
    view:"kanban",
    ...
    data:[
        {id:"1", text:"Task 1", status: "new", personId: "1", projectId:
"22", ...},
        {id:"2", text:"Task 2", status: "work", personId: "2", projectId:
"22", ...},
        ...
    ]
});

$$("myBoard").filter(function(task){
    return task.personId == "1" && task.projectId == "22";
});
```

RELATED SAMPLE: Filtering

Editing Items

Changing an item's properties

To change properties of an item, you should use the following technique:

```
// get item data by item id
var task = myBoard.getItem("task1");
// set new values
task.text = "Ticket 225";
task.status = "work";
task.color = "red";
...
// apply changes
myBoard.updateItem("task1", task);
```

You can read the details on the getItem and updateItem methods.

Editing via a Form

Webix Form and Board Binding

Form and Kanban Board can be bound to each other to ensure select-based **synchronous changing** of their **data**, which allows editing data on the Board:

- select a kanban's item the data of which you want to show in Form (use setCursor method);
- form will be filled with the item's data that you can edit;
- form saving will send changed data back to the component.

```
// binding Form to Kanban data
$$("myForm").bind($$("myBoard"));

// select an item for editing
function editItem(itemId){
   $$("myBoard").setCursor(itemId);
}

// data saving
function saveData(){
   $$("myForm").save();
}
```

RELATED SAMPLE: Load and Save

You can study data binding separately.

Handling Events

Attaching Event Handler

The user can add any user-defined handler to any of the available events. To do this, the user can use the attachEvent method with the following parameters:

- evName - name of the event;
- evHandler - user-defined event handler.

```
webix.attachEvent(evName, evHandler);
```

Several handlers can be attached to one and the same event, and all of them will be executed. The names of the events are case-insensitive.

Detaching Event Handler

There is a simple way of removing an event-handler:

```
webix.detachEvent(id); // unique id of the event handler
```

List of supported events

Below you can find the full list of events available for Kanban API:

- onListItemClick - fires on an item click
- onListItemDblClick - occurs on an item double-click
- onListIconClick - fires on clicking icon button in the item footer
- onAvatarClick - occurs on an avatar click
- onListBeforeSelect - fires before item selection started
- onListAfterSelect - occurs after an item is selected
- onListBeforeContextMenu - fires on item right click before native context menu displayed
- onListAfterContextMenu - fires after the context menu was called in the item area
- onListBeforeDrag - fires before the mouse button is pressed and the cursor is moved over a draggable item
- onListBeforeDragIn - fires before a dragged item is moved over the droppable list
- onListBeforeDrop - fires before a dragged element is released over the droppable list
- onListAfterDrop - fires after drag-n-drop has been finished
- onBeforeStatusChange - occurs before an item is dropped to a list with different status

- onAfterStatusChange - fires after an item is dropped to a list with different status

onListItemClick

fires on an item click

```
$$("myBoard").attachEvent("onListIconClick",
function(itemId,ev,node,list)){
    // your code
};
```

params:

- itemId - {string} the id of the clicked item
- ev - {event object} a native event object
- node - {element} the target HTML element
- list - {object} the list object where click has happened

RELATED SAMPLE: Item Click

onListItemDblClick

fires when an item has been double-clicked

```
$$("myBoard").attachEvent("onListItemDblClick",
function(itemId,ev,node,list)){
    // your code
};
```

params:

- itemId - {string} the id of the clicked item
- ev - {event object} a native event object
- node - {element} the target HTML element
- list - {object} the list object where doulble-click has happened

RELATED SAMPLE: Item DoubleClick

onListIconClick

fires on clicking any icon in the list item

```
$$("myBoard").attachEvent("onListIconClick", function(iconId,
itemId,ev,node,list)){
    // your code
    return false;
};
```

params:

- iconId - {string} the id of the icon
- itemId - {string} the id of the clicked item
- ev - {event object} a native event object
- node - {element} the target HTML element
- list - {object} the list object where click has happened

returns:

{bool} if an event handler returns false, onListItemClick handler will not be called.

RELATED SAMPLE: Icon Click

onAvatarClick

fires on clicking an avatar in the item

```
$$("myBoard").attachEvent("onAvatarClick", function(itemId,ev,node,list){
    // your code
};
```

params:

- itemId - {string} the id of the clicked item
- ev - {event object} a native event object
- node - {element} the target HTML element
- list - {object} the list object where click has happened

returns:

{bool} if an event handler returns false, onListItemClick handler will not be called

RELATED SAMPLE: Avatar Click

onListBeforeSelect

fires before an item selection started

```
$$("myBoard").attachEvent("onListBeforeSelect",
function(itemId,selection,list){
    // your code
};
```

params:

- itemId - {string} the item id
- selection - {boolean} true - if to select, false - to unselect

- list - {object} the list that contains the item

returns:

{bool} if an event handler returns false, the item will not be selected

RELATED SAMPLE: Item Click

onListAfterSelect

fires after an item has been selected

```
$$("myBoard").attachEvent("onListAfterSelect", function(itemId,list){
    // your code
};
```

params:

- itemId - {string} the item id
- list - {object} the list that contains the item

RELATED SAMPLE: Item Click

onListBeforeContextMenu

fires on an item right click, before native context menu displayed

```
$$("myBoard").attachEvent("onListBeforeContextMenu",
function(itemId,ev,node,list){
    // your code

    // block native context menu
    webix.html.preventEvent(ev);
};
```

params:

- itemId - {string} the id of the clicked item
- ev - {event object} a native event object
- node - {element} the target HTML element
- list - {object} the list object where click has happened

RELATED SAMPLE: onContext event

onListAfterContextMenu

fires after the context menu was called in the item area

```
$$("myBoard").attachEvent("onListAfterContextMenu",
```

```
function(itemId,ev,node,list)){
    // your code

};
```

params:

- itemId - {string} the id of the clicked item
- ev - {event object} a native event object
- node - {element} the target HTML element
- list - {object} the list object where click was happened

onListBeforeDrag

fires before the mouse button is pressed and the cursor is moved over a draggable item

```
$$("myBoard").attachEvent("onListBeforeDrag", function(context,ev,list)){
    if(...){
        return false;
    }
    return true;
};
```

params:

- context - {object} drag-n-drop context object with a set of properties:
  - from - the source object,
  - to - the target object,
  - source - the ID of the dragged item(s),
  - target - the ID of the drop target, null for drop on empty space,
  - start - the ID from which DnD started
- ev - {event object} a native event object
- list - {object} the list object where the event has happened

returns:

{bool} returning false will prevent dragging of the element

RELATED SAMPLE: Drag-n-Drop

onListBeforeDragIn

fires before a dragged element is moved over the droppable list

```
$$("myBoard").attachEvent("onListBeforeDragIn",
function(context,ev,list)){
    if(...){
```

```
        return false;
    }
    return true;
};
```

params:

- context - {object} drag-n-drop context object with a set of properties:
    - from - the source object,
    - to - the target object,
    - source - the ID of the dragged item(s),
    - target - the ID of the drop target, null for drop on empty space,
    - start - the ID from which DnD started
- ev - {event object} a native event object
- list - {object} the list object where the event has happened

returns:

{bool} returning false will prevent dropping of the element

RELATED SAMPLE: Drag-n-Drop

onListBeforeDrop

fires before a dragged element is released over the droppable list

```
$$("myBoard").attachEvent("onListBeforeDrop", function(context,ev,list){
    if(...){
        return false;
    }
    return true;
};
```

params:

- context - {object} drag-n-drop context object with a set of properties:
    - from - the source object,
    - to - the target object,
    - source - the ID of the dragged item(s),
    - target - the ID of the drop target, null for drop on empty space,
    - start - the ID from which DnD started
- ev - {event object} a native event object
- list - {object} the list object where the event has happened

returns:

{bool} returning false will prevent further drag-and-drop processing

onListAfterDrop

fires after drag-n-drop has finished

```
$$("myBoard").attachEvent("onListAfterDrop", function(context,ev,list)){
    // your code
};
```

params:

- context - {object} drag-n-drop context object with a set of properties:
  - from - the source object,
  - to - the target object,
  - source - the ID of the dragged item(s),
  - target - the ID of the drop target, null for drop on empty space,
  - start - the ID from which DnD started
- ev - {event object} a native event object
- list - {object} the list object where the event has happened

RELATED SAMPLE: Drag-n-Drop

onBeforeStatusChange

fires before an item is going to be dropped into the list with different status

```
$$("myBoard").attachEvent("onBeforeStatusChange",
function(itemId,newStatus)){
    var status = this.getItem(itemId).status;
    if(...){
        return false;
    }
    return true;
};
```

params:

- itemId - {string} the item's id
- status - {string} a new item's status
- list - {object} the list object where the event has happened
- context - {object} drag-n-drop context object
- ev - {event object} a native event object

returns:

{bool} returning false will prevent further drag-and-drop processing

onAfterStatusChange

fires after an item has been dropped into the list with a different status

```
$$("myBoard").attachEvent("onListAfterDrop", function(itemId,newStatus){
    // your code
};
```

params:

- itemId - {string} the item id
- status - {string} a new item's status
- list - {object} the list object where the event has happened
- context - {object} drag-n-drop context object
- ev - {event object} a native event object

General How-Tos

This article is dedicated to the most obvious questions that may arise during the creation of Kanban Board.

How to add icons to Kanban Board?

You need to specify icons array in the type for Kanban Lists. The description of the used properties is given in the article Data Visualization.

```
webix.type(webix.ui.kanbanlist,{
    name: "cards",
    icons:[
        {
            id: "comments",
            tooltip: "Comments",
            icon:"comment-o",
            show: function(obj){
                return !!obj.comments
            },
            template:"#comments.length#",
            click: function(e,id){
                var item = this.getItem(id);
                ...
            }
        },
        {
            id: "edit",
            icon:"edit",
            tooltip: "Edit Task",
            click: function(e,id){
                var item = $$("myBoard").getItem(id);
                ...
            }
        }
    ]
});

webix.ui({
    view:"kanban",
    ...
    cols:[
        {
            header:"Backlog",
            body:{
                view:"kanbanlist",
                status:"new",
                type: "cards"
            }
```

```
        },
        ....
    ]
});
```

RELATED SAMPLE: Icons

How to define tags for tasks?

There are 2 variants of specifying tags:

1) by setting the **tags** property in the item data. For example, in your datasource it can be something as in:

```
webix.ui({
   view: "kanban",
   cols:[..],
   data: [{ id: "1", tags: "tag1,tag2,tag3", ...},...]
});
```

2) by setting tags dynamically

```
// get an item
var item = $$("myBoard").getItem("1"); // '1' is the item's id
// set the 'tags' property
item.tags = "tag1,tag2,tag3";
// update the item
$$("myBoard").updateItem("1");
```

If you need to use custom logic for tags rendering, you can redefine the **templateTags**.

How to add an avatar for a task's assignee?

You need to specify **templateAvatar** in the type for Kanban Lists.

For example you can show <img> elements:

```
webix.type(webix.ui.kanbanlist,{
    name: "cards",
    // avatar template
    templateAvatar: function(obj){
        if(obj.personId){
            return '<img class="avatar"
src="../common/imgs/'+obj.personId+'.jpg" />';
        }
        return "";
    },
    ...
```

```
});
```

RELATED SAMPLE: Member Avatar

RELATED SAMPLE: Avatar Click

How to customize the task's content?

The tasks' content is specified in the **templateBody**. By default, only an item's text is displayed in the body.

For example, if you need to show images in tasks you can redefine templateBody like this:

```
webix.type(webix.ui.kanbanlist,{
    name: "cards",
    // template for item body
    // show the item's image and text
    templateBody: function(obj){
        var html = "";
        if(obj.image)
            html += "<img class='image'
src='../common/imgs/"+obj.image+"'/>";
        html += "<div>"+obj.text+"</div>";
        return html;
    },
    ...
});
```

RELATED SAMPLE: Templates

How to change the color of the task's border?

The default color of the tasks' left border is specified in the css class **.webix_kanban_list_content**. Set the necessary color in the **border-left** property:

```
kanban.css

.webix_kanban_list_content {
  ...
  border-left: 3px solid #27ae60;
}
```

You can redefine this css rule and set a new color for all tasks in your Kanban board.

However, if you need to set a new color for a separate item, you need to define the **color** property in the item's data. So, there will be something like this in your datasource:

```
webix.ui({
   view: "kanban",
```

```
    id: "myBoard",
    ...,
    data: [...,{ id: "3", color: "red", ...},...]
});
```

The color of the left border can also be changed dynamically by means of updating the item:

```
// get an item
var item = $$("myBoard").getItem("3"); // '3' is the item's id
// set a new 'color' property
item.color = "red";
// update the item
$$("myBoard").updateItem("3");
```

How to change the status of an item?

The status of an item can be changed in two ways:

1) by dragging an item to a different column

2) by applying a new value to the item's 'status' property:

```
// get an item
var item = $$("myBoard").getItem("1");  // '1' is the item's id
// set the 'status' property
item.status = "work";
// update the item
$$("myBoard").updateItem("1");
```

How to forbid dragging items from/to a column?

In case you need to forbid moving tasks from/to particular columns, you should make use of Kanban Events.

For example, you may ban dragging to columns that aren't situated next to the source column. Thus, tasks will be dragged only between two neighbouring columns.

To implement this idea, the onListBeforeDragIn event can be used.

The handler function will take 3 parameters:

- dragContext - {object} drag-n-drop context object with a set of properties:
  - from - the source object,
  - to - the target object,
  - source - the ID of the dragged item(s),
  - target - the ID of the drop target, null for drop on empty space,
  - start - the ID from which DnD started

- e - {event object} a native event object
- list - {object} the list object where the event has happened

```
//specifying event handler
function onBeforeDragIn(dragContext,e,list){

    // item id
    var item =this.getItem(dragContext.start);

    // if we move an item from one list to another
    if(dragContext.from != dragContext.to){
        //the status of the source column
        var statusFrom = dragContext.from.config.status;
        //the status of the target column
        var statusTo = dragContext.to.config.status;
        //assigning indices to statuses
        var statusIndex = {"new":,"work":1, "test": 2, "done":3};
        //getting difference between the source and target columns'
indices
        //if it's more than 1, the item won't be moved
        var diff = Math.abs(statusIndex[statusFrom] -
statusIndex[statusTo]);
        if(diff>1){
            return false;
        }
    }
    return true;
}

webix.ready(function(){
    webix.ui({
        view:"kanban",
        id: "myBoard",
        on:{
            onListBeforeDragIn: onBeforeDragIn,
        },
     });
});
```

RELATED SAMPLE: Drag-n-Drop

How to set the background color to the item?

To apply a specific css class for an item you can set the **$css** property for it. This class will be added to className of item element:

```
webix.ui({
    view:"kanban",
```

```
    data: [
       { id:"1", status:"new", text:"Task 1", $css: "critical", ... },
       ...
    ]
});
```

Here is an example of setting a new background color for an item with the 'critical' $css property:

```
<style>
  .critical .webix_kanban_list_content{
    background-color: #fff2c1;
    border-color:  #e0d7b7;
    border-left-color:  #f5cf3d;
  }
</style>
```

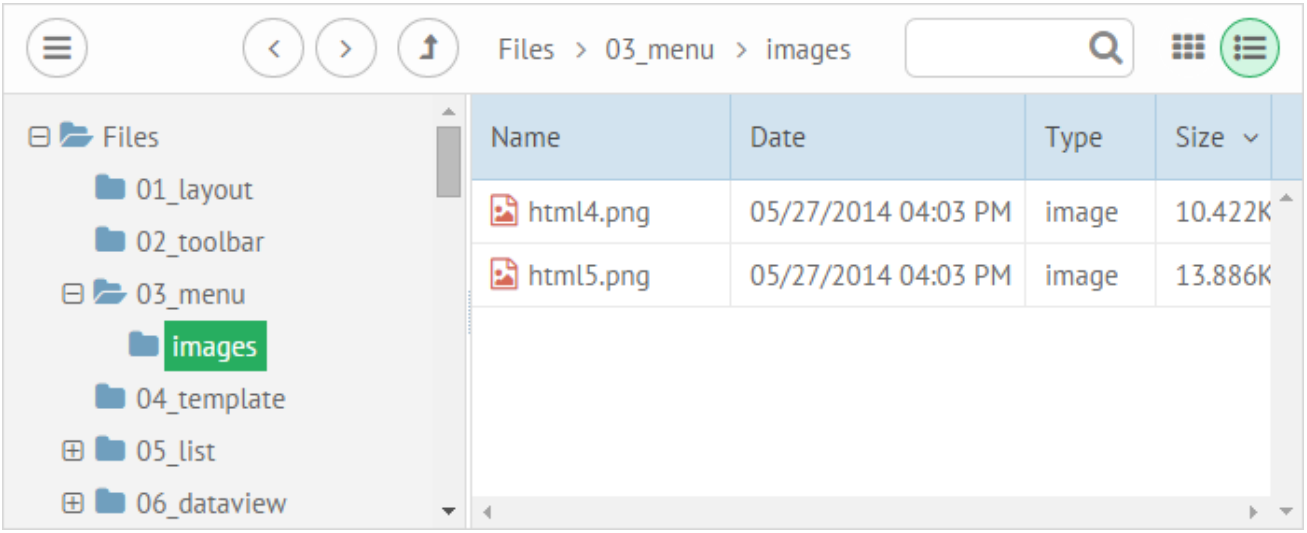
RELATED SAMPLE: Item styling

File Manager

Webix File Manager is intended for viewing folders and files and navigating through them in a handy way.

This tool lets you perform all habitual operations with files and folders (copy/paste, rename, delete), as well as drag-n-drop them. Moreover, it's possible to upload and download files and search for necessary files and folders.

Apart from files, File Manager allows working with any kind of hierarchical data and can be easily integrated into a web app.



Configuring File Manager

- Creating File Manager on a page
- Loading and Saving Data
- Data Visualization
- File Manager Structure
- Localizing File Manager

Discusses the main points of File Manager initialization and customization.

Working with File Manager

- Manipulating Items
- Handling Events
- API Reference
- General How-Tos

Describes possible operations with File Manager and its elements.

Creating File Manager on a page

There are two simple steps you need to complete to initialize File Manager on a page:

Include the necessary JS and CSS files into HTML file.

Call the object constructor.

Included files

You need to include 4 files on a page:

```
<!-- js files -->
<script src="codebase/webix.js" type="text/javascript"></script>
<script src="codebase/filemanager.js" type="text/javascript"></script>
<!-- css files -->
<link rel="stylesheet" href="codebase/webix.css" type="text/css"
charset="utf-8">
<link rel="stylesheet" href="codebase/filemanager.css" type="text/css"
charset="utf-8">
```

Initialization

The code below initializes File Manager and populates it with data:

```
//webix.ready() function ensures that the code will be executed when the
page is loaded
webix.ready(function(){
    //object constructor
    webix.ui({
        view:"filemanager",
        id:"files"
    });
    // loading data source
    $$("files").load("data/files.php");
});
```

RELATED SAMPLE: Basic init

Loading and Saving Data

Data can be loaded in different formats, including json (the default one) and xml. The data item should include at least 5 properties:

- id - {string} the unique id of the item
- value - {string} the name of a file or a folder
- type - {string} the type of a file
- date - {number} time of the file/folder modification in Unix format
- size - {number} file size in bytes

An example of json data item is given below:

```
{
    id:     "base1",
    value:  "Product1.jpg",
    type:   "image",
    date:   1420717913,
    size:   4500
},
```

You can use a data source in json, xml or other format supported by Webix TreeStore. Data can be generated via any server-side script. In the data source you can use information from a real file system or records from a database.

Requests to the server side

When some action is executed (upload, download, etc.) with files, File Manager sends a POST Ajax request to the corresponding server-side script specified in the "handlers" property of FileManager configuration:

```
webix.ready(function(){
    var fManager = new webix.ui({
        view:"filemanager",
        handlers:{
            "upload"    : "data/saving.php",
            "download"  : "data/saving.php",
            "copy"      : "data/saving.php",
            "move"      : "data/saving.php",
            "delete"    : "data/saving.php",
            "rename"    : "data/saving.php",
            "create"    : "data/saving.php"
        }
    });
    fManager.load("data/get.php");
});
```

The request contains several parameters:

- action - the name of the executed action
- source - the id of the file or the folder that the action has been executed with
- target - (optional) the id of the parent folder (for example, in the case of "copy" action) or the new name of the file ("rename" action)
- upload - (optional) a file that will be uploaded (for the "upload" action)

For example, if we copy a file with the id "myfile1" into the folder with the id "folder2", the request contains the following parameters:

```
{
    action: "copy",
    source: "myfile1",
    target: "folder2"
}
```

File Manager is a purely client-side widget, therefore you can implement any logic at the backend. You can find several demos that implement PHP data in the File Manager package.

RELATED SAMPLE: Actions Processing

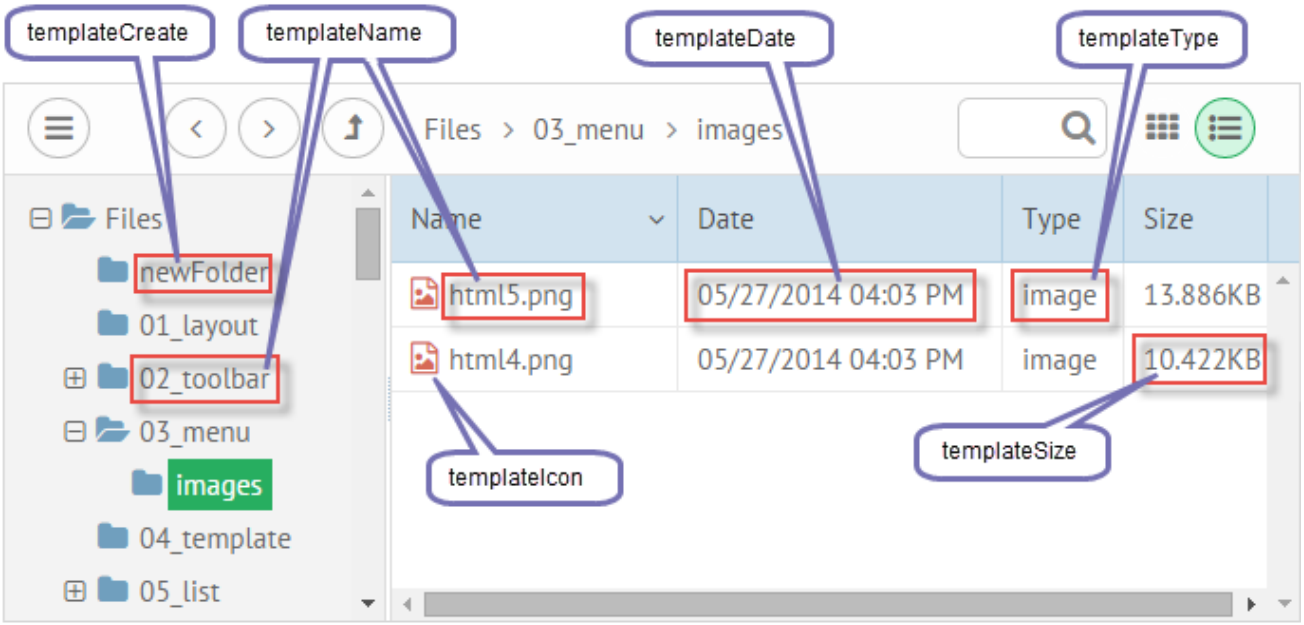
Data Visualization

Data items visualization in File Manager is rather flexible, so you can easily customize it by means of redefining the corresponding templates.

Thus, it's possible to specify the necessary date format, set desired icons or change an item's name presentation.

The following templates are available:

- templateName - the template applied for the folder/file's name
- templateSize - the template applied for the file size
- templateDate - the template applied for the format of folder/file's date of modification
- templateCreate - the template for a newly created folder
- templateIcon - the template for icons used in "Files" and "Table" view modes. The icons are customizable
- templateType - the template for the type of a file
- icons - an array of icons. The icons are taken from the Font Awesome collection



templateName

Specifies how the folder/file's name will be displayed in File Manager.

```
templateName: function(fileObj,common){
    return fileObj.value
}
```

templateSize

Specifies the file size.

```
templateSize: function(obj,common){
    var value = obj.size;
    var isInt = (parseInt(value,10) == value);

    // get size label
    var labels = webix.i18n.filemanager.sizeLabels; // ["B","KB",...]
    var sizeIndex = ;
    while(value/1024 >1){
        value = value/1024;
        sizeIndex++;
    }
    var label = labels[sizeIndex];

     // locale format
    var getFormatedValue = webix.Number.numToStr({
        decimalDelimiter:webix.i18n.decimalDelimiter,
        groupDelimiter:webix.i18n.groupDelimiter,
        decimalSize : isInt?:webix.i18n.groupSize
    });

    return getFormatedValue(value)+""+label;
},
```

templateDate

Specifies the format of folder/file's date of modification.

```
templateDate: function(obj){
    var date = obj.date; // 'date' property is Unix time
    if(typeof(date) != "object"){
        date = new Date(parseInt(obj.date,10)*1000);
    }
    return webix.i18n.fullDateFormatStr(date);
},
```

templateCreate

Specifies properties for a newly created folder.

```
templateCreate: function(){
    return {value: "newFolder", type: "folder", date: new Date()};
},
```

templateType

Specifies the type of a file.

```
templateType: function(obj){
   var types = webix.i18n.filemanager.types;
   return types&&types[obj.type]?types[obj.type]:obj.type;
}
```

templateIcon

A customizable template for icons in "Files" and "Table" view modes.

```
templateIcon: function(obj,common){
    return "<span class='webix_icon webix_fmanager_icon fa-"
        +(common.icons[obj.type]||common.icons["file"])+"'></span>";
},
```

RELATED SAMPLE: Icons

Icons

The "icons" collection allows specifying FontAwesome icons for the **templateIcon**. You can specify icons that correspond to the loaded file types. If the collection does not contain an icon for the loaded type, FileManager will set the "default" icon type.

By default, the following icons are set:

```
icons: {
            folder: "folder",
            doc: "file-word-o",
            excel: "file-excel-o",
            pdf: "file-pdf-o",
            pp: "file-powerpoint-o",
            text: "file-text-o",
            video: "file-video-o",
            image: "file-image-o",
            code: "file-code-o",
            audio: "file-audio-o",
            archive: "file-archive-o",
            file: "file-o"
        }
```

File Manager Structure

File Manager Layout

File Manager consists of several Webix views.

The main layout includes two rows: a row with a toolbar and the second row with a sublayout that contains Tree and mode views. Thus, the structure of File Manager is:

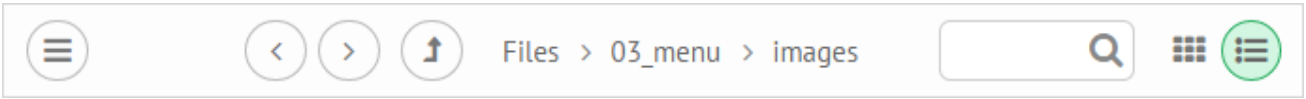
- Toolbar:
  - Menu button
  - Spacer
  - Back and Forward group of buttons
  - LevelUp button
  - Path view
  - Search control
  - Modes button
- bodyLayout with 3 columns:
  - Tree
  - Resizer view
  - 'Files' and 'Table' view modes

The layout of File Manager is rather flexible. It means that you can customize it according to your preferences.

For example, if you want to swap one toolbar element with another one or even to remove it, the functionality of File Manager will remain untouched.

You can find more information concerning the possibilities of customizing File Manager views in the article General How-Tos.

Toolbar



Toolbar is based on Webix Toolbar view.

```
{
    id:         "toolbar",
    css:        "webix_fmanager_toolbar",
    paddingX:   10,
    paddingY:   5,
    margin:     7,
    cols:[
```

```
        // buttons
    ]
}
```

You can use the **$$("toolbar")** method to refer to its object:

```
var toolbar = $$("fmanager").$$("toolbar");
```

Menu button



Menu button is based on Webix Button control. It is used to activate the Actions menu.

```
{
    id:     "menu",
    view:   "button",
    type:   "iconButton",
    css:    "webix_fmanager_back",
    icon:   "bars",
    width:  37
}
```

You can get the reference to its object in the following way:

```
var menu = $$("fmanager").$$("menu");
```

Spacer

The spacer is used to separate the Menu and Back buttons.

```
{
    id:     "menuSpacer",
    width:  65
}
```

To refer to its object use the **$$("menuSpacer")** method:

```
var spacer = $$("fmanager").$$("menuSpacer");
```

Back and Forward buttons



The Back and Forward buttons are based on Webix Button. They let move back and forward through the navigation history.

```

```

```
//the Back button
{
    id:     "back",
    view:   "button",
    type:   "iconButton",
    css:    "webix_fmanager_back",
    icon:   "angle-left", width: 37
}

//the Forward button
{
    id:     "forward",
    view:   "button",
    type:   "iconButton",
    css:    "webix_fmanager_forward",
    icon:   "angle-right",
    width:  37
}
```

To refer to the object of the buttons, use the **$$("back")** and **$$("forward")** buttons, respectively:

```
//for the Back button
var bback = $$("fmanager").$$("back");

//for the Forward button
var bforward = $$("fmanager").$$("forward");
```

LevelUp button



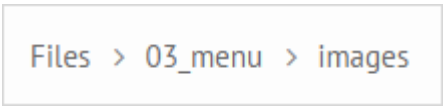
The Level Up button is based on Webix Button. It selects the parent folder of the selected item (file/folder):

```
{
    id:         "up",
    view:       "button",
    type:       "iconButton",
    css:        "webix_fmanager_up",
    icon:       "level-up",
    disable:    true,
    width:      37
}
```

You can get the reference to its object in the following way:

```
var levelUp = $$("fmanager").$$("up");
```

Path view



The Path view displays the path to the currently selected item (file/folder).

```
{
    id: "path"
    view: "path",
    borderless: true
}
```

To refer to its object use the **$$("path")** method:

```
var path = $$("fmanager").$$("path");
```

Search control



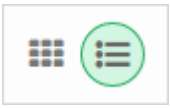
The Search element is based on Webix Search control and allows searching for files and folders, the names of which contain the entered letter combinations:

```
{
    id: "search",
    view: "search",
    gravity: 0.3
}
```

You can use the **$$("search")** method to refer to its object:

```
var search = $$("fmanager").$$("search");
```

Modes button



The Modes button presents a segmented button, based on Webix Segmented Button control. It lets switch between two modes of displaying folders and files: "Files" and "Table".

```
{
   id:      "modes",
   view:    "segmented",
   value:   settings.mode,
   options: [
      // options
   ]
}
```

The options are specified as an array:

```
[
    {
        id: "files",
        width: 32,
        value: "<span class=\"webix_fmanager_mode_option webix_icon fa-
th\"></span>"
    },
    {
        id: "table",
        width: 32,
        value: "<span class=\"webix_fmanager_mode_option webix_icon fa-
list-ul\"></span>"
    }
]
```

To refer to its object use the **$$("modes")** method:

```
var modes = $$("fmanager").$$("modes");
```

bodyLayout

bodyLayout includes 3 columns: Tree, Resizer view and a column with "Files" and "Tree" view modes.

Tree

The Tree is based on Webix Tree. It displays the hierarchy of folders and allows navigating through them.

```
{
    id: "tree",
    width: 230,
    view: "filetree",
    select: true,
    filterMode:{
        showSubItems:false,
        openParents:false
    },
    type: "FileTree",
    // mouse navigation
    navigation: true,
    scroll: true,
    // editing options
    editor:"text",
    editable: true,
    // disable editing on double-click
    editaction: false,
    // drag-n-drop
    drag: true,
    // tab 'key' navigation
```

```
    tabFocus: true,
    // context menu
    onContext:{}
}
```

Tree uses "FileTree" data type that configures folder icons for nodes:

```
webix.type(webix.ui.tree,{
    name: "FileTree",
    folder: function(obj, common){
        if (obj.icon)
            return "<div class='webix_icon icon fa-folder'></div>";
        if (obj.$count && obj.open)
            return "<div class='webix_icon icon fa-folder-open'></div>";
        return "<div class='webix_icon icon fa-folder'></div>";
    }
});
```

You can get the reference to tree object in the following way:

```
var tree = $$("fmanager").$$("tree");
```

Resizer view

Resizer is a draggable border between the Tree and the File views. It is based on Webix Resizer.

```
{
    view:"resizer",
    width:2
}
```

View modes

These are 2 modes of viewing files of the selected folder: "Files" and "Table". By default, the "Table" mode is active.

Files mode

"Files" mode is based on Webix Dataview.

```
{
    id: "files",
    view: "fileview",
    type: "FileView",
    select: "multiselect",
    // editing options
    editable:true,
    editor:"text",
    editValue:"value",
    // disable editing on double-click
```

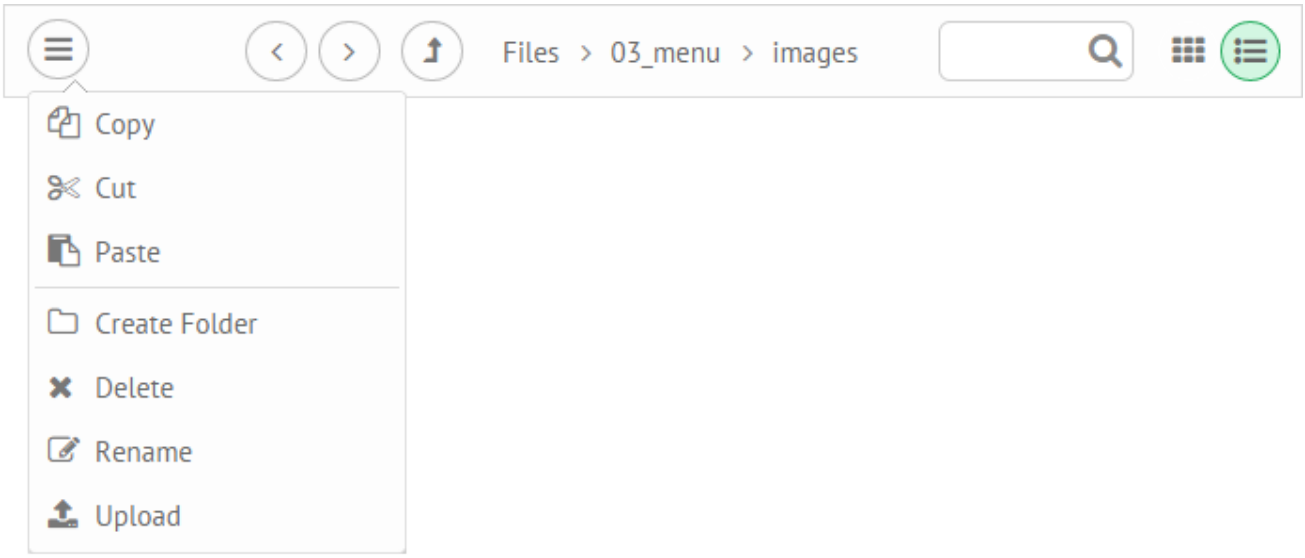
```
    editaction: false,
    // drag-n-drop
    drag: true,
    // mouse navigation
    navigation: true,
    // tab 'key' navigation
    tabFocus: true,
    // context menu
    onContext:{}
}
```

"Table" mode is based on Webix DataTable component.

```
{
    id: "table",
    view: "filetable",
    css: "webix_fmanager_list",
    columns: "columns",
    // editing options
    editable: true,
    // disable editing on double-click
    editaction: false,
    select: "multiselect",
    // drag-n-drop
    drag: true,
    // mouse navigation
    navigation: true,
    resizeColumn:true,
    // tab 'key' navigation
    tabFocus: true,
    // context menu
    onContext:{}
}
```

Actions menu

The Actions menu displays actions that can be applied to selected files or folders. It appears by clicking the Menu button or by clicking the right mouse button:



The Actions menu is based on Webix ContextMenu. That's why you can operate its items via TreeStore API.

To refer to the Actions menu, you can apply the **getMenu()** method to File Manager:

```
var actions = $$("fmanager").getMenu();
```

By default, actions menu contains the following data array:

```
[
    {
        id: "copy",
        method: "markCopy",
        icon: "copy",
        value: webix.i18n.filemanager.copy // "Copy"
    },
    {
        id: "cut",
        method: "markCut",
        icon: "cut",
        value: webix.i18n.filemanager.cut // "Cut"
    },
    {
        id: "paste",
        method: "pasteFile",
        icon: "paste",
        value: webix.i18n.filemanager.paste // "Paste"
    },
    {   $template:"Separator" },
    {
        id: "create",
        method: "createFolder",
        icon: "folder-o",
        value: webix.i18n.filemanager.create // "Create Folder"
    },
    {
        id: "deleteFile",
```

```
        method: "deleteFile",
        icon: "times",
        value: webix.i18n.filemanager.delete // "Delete"
    },
    {
        id: "edit",
        method: "editFile",
        icon: "edit",
        value: webix.i18n.filemanager.rename // "Rename"
    },
    {
        id: "upload",
        method: "uploadFile",
        icon: "upload",
        value: webix.i18n.filemanager.upload  // "Upload"
    }
]
```

The properties of actions are the following:

- id - {string} the id of an action
- method - {string} the name of the method that is used to implement the action
- icon - {string} an icon from the Font Awesome collection used for the action
- value - {string} the action title

You can reload data using the following approach:

```
var actions = $$("fmanager").getMenu();
actions.clearAll();
var newData = [{id: "copy", ...},...];
actions.parse(newData);
```

The add method allows adding a new action to menu:

```
actions.add({id: "myAction", icon: "file", value: "My Action"});
```

It is possible to set a click handler for the added action using onItemClick event handler:

```
actions.attachEvent("onItemClick", function(id){
   if(id == "myAction"){
       // some code here
   }
});
```

To remove an existing action, you can call the remove method for it:

```
actions.remove("upload");
```

Localizing File Manager

You can localize File Manager according to the peculiarities of a certain language.

The properties that can be localized are the following:

- names of actions ("Copy", "Paste", etc.)
- column titles
- date formats

Date formats

Date can be easily localized by setting one of the build-in locales.

All locales are stored in webix.i18n.locales object. By default, the library uses the "en-US" locale:

```
webix.i18n.locales["en-US"]={
  ...
};
```

Titles localization

File Manager titles are stored in the **webix.i18n.filemanager** object:

```
webix.i18n.filemanager = {
    name: "Name",
    size: "Size",
    type: "Type",
    date: "Date",
    copy: "Copy",
    cut: "Cut",
    paste: "Paste",
    upload: "Upload",
    remove: "Delete",
    create: "Create Folder",
    rename: "Rename",
    location: "Location",
    select: "Select Files",
    sizeLabels: ["B","KB","MB","GB"],
    saving: "Saving...",
    errorResponse: "Error: changes were not saved!",
    replaceConfirmation: "Would you like to replace existing files?",
    createConfirmation: "The folder already exists. Would you like to
replace it?",
    renameConfirmation: "The file already exists. Would you like to
replace it?",
```

```
    yes: "Yes",
    no: "No",
    types:{
        folder: "Folder",
        doc: "Document",
        excel: "Excel",
        pdf: "PDF",
        pp: "PowerPoint",
        text: "Text File",
        video: "Video File",
        image: "Image",
        code: "Code",
        audio: "Audio",
        archive: "Archive",
        file: "File"
    }
};
```

There are 2 ways of applying a custom locale:

1) redefine default titles according to your locale directly in the **webix.i18n.filemanager** object;

2) set a certain locale in the following way:

- include the chosen locale library into your page:

```
<script src="../../../codebase/i18n/ru.js"
type="text/javascript"></script>
```

- define 'filemanager' titles in the **webix.i18n.locales[localeName]** object:

```
webix.i18n.locales['ru-RU'].filemanager = {
    name: "Имя",
    size: "Размер",
    ...
};
```

- apply the created locale with the help of the setLocale method:

```
webix.i18n.setLocale('ru-RU');
```

RELATED SAMPLE: Localization

Manipulating Items

You can perform various actions with files and folders in File Manager. The actions are presented in the popup menu called "Actions". It is activated by right-clicking on a folder/file in the Tree or Files view or by clicking the Menu button on the toolbar.

However, you can also customize actions by means of redefining the corresponding methods.

Copying folders/files

To copy a folder/file, use the copyFile() method:

```
fmanager.copyFile(source,target);
```

Moving folders/files

To move a folder/file, use the moveFile() method:

```
fmanager.moveFile(source,target);
```

Deleting folders/files

To delete a folder/file, use the deleteFile() method:

```
fmanager.deleteFile(ids);
```

Renaming folders/files

To rename a folder/file, use the renameFile() method:

```
fmanager.renameFile("file1","newFile.txt");
```

Creating a new folder

To create a folder, use the createFolder() method. You need to pass an id of the folder where you want to create a new folder and templateCreate will apply other properties automatically:

```
fmanager.createFolder("folder1");
```

Downloading files

To download a file, you need to call the download() method and pass the file id to it:

```
fmanager.download("myfile");
```

Uploading files

To open a file upload dialog, use the uploadFile() method:

```
fmanager.uploadFile("myfolder");
```

Handling Events

Attaching Event Handler

The user can add any user-defined handler to any of the available events. To do this, the user can use the attachEvent method with the following parameters:

- evName - name of the event;
- evHandler - user-defined event handler.

```
webix.attachEvent(evName, evHandler);
```

Several handlers can be attached to one and the same event, and all of them will be executed. The names of the events are case-insensitive.

Detaching Event Handler

There is a simple way of removing an event-handler:

```
webix.detachEvent(id); // unique id of the event handler
```

List of supported events

The File Manager API includes the following groups of events:

1) Actions handling events

- onBeforeMarkCopy - fires before an item has been copied
- onBeforeCreateFolder - fires before a new folder has been created
- onBeforeMarkCut - fires before an item has been cut
- onBeforeDeleteFile - fires before an item is deleted
- onBeforeEditFile - fires when the Rename action is chosen in the popup menu
- onBeforePasteFile - fires before an item has been pasted
- onBeforeUploadFile - fires on the "upload" action click

2) Drag-n-drop Events

- onAfterDrop - fires after drag-n-drop has been finished
- onBeforeDrag - fires before drag-n-drop has started
- onBeforeDragIn - fires before a dragged item is moved over the droppable area
- onBeforeDrop - fires before a dragged item is released over the droppable area

3) Editing events

- onAfterEditStop - fires after the editor is closed
- onBeforeEditStop - fires before editing is finished

4) Other

- onErrorResponse - fires if an error has occured during some operation
- onBeforeRun - fires on file double-click or on enter click
- onFolderSelect - fires on selecting a folder in the Tree view
- onViewInit - fires during the initialization of an object with the mode configuration

onBeforeMarkCopy

fires before an item has been marked for copying

```
$$("fmanager").attachEvent("onBeforeMarkCopy", function(ids){
    // your code
    return true;
});
```

params:

- ids - {string} the ids of the copied items

returns:

- {bool}- returning false will prevent copying

RELATED SAMPLE: Copy, Cut, Paste Events

onBeforeCreateFolder

fires before a new folder has been created

```
$$("fmanager").attachEvent("onBeforeCreateFolder",function(id){
    // your code
    return true;
});
```

params:

- id - {string} the id of the created item

returns:

- {bool} - returning false will prevent creating

onBeforeMarkCut

fires before an item has been marked for cutting

```
$$("files").attachEvent("onBeforeMarkCut", function(ids){
    // your code
    return true;
});
```

params:

- ids - {string} the ids of the copied items

returns:

- {bool} - returning false will prevent cutting operation

RELATED SAMPLE: Copy, Cut, Paste Events

onBeforeDeleteFile

fires before an item is deleted

```
$$("fmanager").attachEvent("onBeforeDeleteFile",function(id){
    // your code
    return true;
});
```

params:

- id - {string} the id of the item to delete

returns:

- {bool} - returning false will prevent deleting

onBeforeEditFile

fires when the Rename action is chosen in the popup menu

```
$$("fmanager").attachEvent("onBeforeEditFile",function(id){
    // your code
    return true;
});
```

params:

- id - {string} the id of the edited item

returns:

- {bool} - if an event handler returns false, onBeforeEditFile handler will not be called.

RELATED SAMPLE: Editing Events

onBeforePasteFile

fires before an item has been pasted

```
$$("fmanager").attachEvent("onBeforePasteFile", function(ids){
    // your code
    return true;
});
```

params:

- ids - {string} the ids of the pasted items

returns:

- {bool} - returning false will prevent pasting

RELATED SAMPLE: Copy, Cut, Paste Events

onBeforeUploadFile

fires on "upload" action click

```
$$("fmanager").attachEvent("onBeforeUploadFile",function(targetId){
    // your code
    return true;
});
```

params:

- targetId - {string} the id of an folder where an new file will be uploaded

returns:

- {bool} - returning false will prevent opening the upload file dialog

onAfterDrop

fires after drag-n-drop has been finished

```
$$("fmanager").attachEvent("onAfterDrop",function(context,ev){
    // your code
    return true;
});
```

params:

- context - {object} drag-n-drop context object with a set of properties:
  - from - the source object,
  - to - the target object,
  - source - the ID of the dragged item(s),
  - target - the ID of the drop target, null for drop on empty space,
  - start - the ID from which DnD started
- ev - {event object} a native event object

RELATED SAMPLE: Drag-N-Drop

onBeforeDrag

fires before drag-n-drop has started

```
$$("fmanager").attachEvent("onBeforeDrag",function(context,ev){
    // your code
    return true;
});
```

params:

- context - {object} drag-n-drop context object with a set of properties:
  - from - the source object,
  - to - the target object,
  - source - the ID of the dragged item(s),
  - target - the ID of the drop target, null for drop on empty space,
  - start - the ID from which DnD started
- ev - {event object} a native event object

returns:

- {bool} - returning false will prevent dragging of the element

RELATED SAMPLE: Drag-N-Drop

onBeforeDragIn

fires before a dragged item is moved over the droppable area

```
$$("fmanager").attachEvent("onBeforeDragIn",function(context,ev){
    // your code
    return true;
});
```

params:

- context - {object} drag-n-drop context object with a set of properties:
  - from - the source object,
  - to - the target object,
  - source - the ID of the dragged item(s),
  - target - the ID of the drop target, null for drop on empty space,
  - start - the ID from which DnD started
- ev - {event object} a native event object

returns:

- {bool} - returning false will prevent dropping of the element

RELATED SAMPLE: Drag-N-Drop

onBeforeDrop

fires before a dragged item is released over the droppable area

```
$$("fmanager").attachEvent("onBeforeDrop",function(context,ev){
    // your code
    return true;
});
```

params:

- context - {object} drag-n-drop context object with a set of properties:
  - from - the source object,
  - to - the target object,
  - source - the ID of the dragged item(s),
  - target - the ID of the drop target, null for drop on empty space,
  - start - the ID from which DnD started
- ev - {event object} a native event object

returns:

- {bool} - returning false will prevent further drag-and-drop processing

RELATED SAMPLE: Drag-N-Drop

onAfterEditStop

fires after the edtor is closed

```
$$("fmanager").attachEvent("onAfterEditStop",function(id,state,editor,view
){
    // your code
    return true;
```

```
});
```

params:

- id - {string} the id of the edited item
- state - {object} contains two properties:
  - old - the old value
  - value - a new value
- editor - {object} editor object
- view - {string} the view where editing occurs

RELATED SAMPLE: Editing Events

onBeforeEditStop

fires before editing is finished

```
$$("fmanager").attachEvent("onBeforeEditStop",function(id,state,editor,vie
w){
    var newValue = state.value
    if(...){
        return false;
    }
    return true;
});
```

params:

- id - {string} the id of the edited item
- state - {object} contains two properties:
  - old - the old value
  - value - a new value
- editor - {object} editor object
- view - {string} the view where editing occurs

returns:

- {bool} - returning false will prevent editing

RELATED SAMPLE: Editing Events

onErrorResponse

fires if an error has occured during some operation

```
$$("files").attachEvent("onErrorResponse", function(requestData,response){
    var action = requestData.action;
```

```
    // your code
    return true;
});
```

params:

- requestData - {object} the data sent with the request
- response - {string} the server response

RELATED SAMPLE: onErrorResponse Event

onBeforeRun

fires before a selected file is downloaded

```
$$("fmanager").attachEvent("onBeforeRun",function(id){
    webix.confirm({
        text:"Do you want to download this file?",
        ok:"Yes",
        cancel:"No",
        callback:function(result){
            if(result)
                $$("files").download(id);
        }
    });
    return false;
});
```

params:

- id - {string} the id of the downloaded item

returns:

- {bool} - if an event handler returns false, onBeforeRun handler will not be called.

RELATED SAMPLE: onBeforeRun event

onFolderSelect

fires on selecting a folder in the Tree view

```
$$("fmanager").attachEvent("onFolderSelect",function(id){
    // your code
});
```

params:

- id - {string} the id of the selected folder

RELATED SAMPLE: Folder Selection

onViewInit

fires during the initialization of an object with the mode configuration

```
$$("fmanager").attachEvent("onViewInit": function(id, config){
    // disable multi-selection for "table" and "files" views
    if (id == "table" || id == "files"){
        config.select = true;
    }
});
```

params:

- id - {string} the view id
- config - {object} the object with view configuration

RELATED SAMPLE: Changing View Configuration

General How-Tos

How to make File Manager readonly?

To activate the readonly mode, you need to set the **readonly** property to **true** in File Manager configuration:

```
webix.ready(function(){
    webix.ui({
        view:"filemanager",
        readonly: true,
        url:"../common/data.php"
    });
});
```

From now on, users won't be able to manipulate items. The Menu button will be hidden.

RELATED SAMPLE: Read-only mode

How to hide a view?

It is possible to hide any view of File Manager by calling the hide method. For example, imagine that you need to remove the Menu button and the spacer between the Menu and Back buttons.

To implement this task, use the following technique:

```
webix.ready(function(){
    webix.ui({
        view:"filemanager",
        id:"files",
    });
    $$("files").load("../common/data.php");

     // hide the Menu button
     $$("files").$$("menu").hide();
     // hide the spacer between the Menu and Back buttons
     $$("files").$$("menuSpacer").hide();
});
```

RELATED SAMPLE: Hiding Buttons

How to add a custom action?

The add method allows adding a new action to menu:

```
var actions = $$("fmanager").getMenu();
actions.add({id: "myAction", icon: "file", value: "My Action"});
```

To process clicks on the new action you can set onItemClick event handler for actions view:

```
actions.attachEvent("onItemClick", function(id){
   if(id == "myAction"){
       // some code here
   }
});
```

How to add a new mode?

To create a custom mode of displaying files in the Files view, you need to define a new mode configuration and apply the onViewInit event.

For example, it's possible to create a new mode based on the List view as follows:

```
webix.ui({
    view:"filemanager",
    id:"files",
    // specify mode selected by default
    mode: "custom",
    // all available modes including a new mode
    modes:["files","table","custom"],
    // object with configuration for new views of File Manager
    structure:{
        // specify the view of the new mode
        "custom": {
            view: "list",
            template: function(obj,common){
                return common.templateIcon(obj,common)+obj.value;
            },
            select: "multiselect",
            editable:true,
            editaction: false,
            editor:"text",
            editValue:"value",
            drag: true,
            navigation: true,
            tabFocus: true,
            onContext:{}
        }
    },
    on:{
        "onViewInit": function(name, obj){
           if (name == "modes"){
               obj.width += 32;
               obj.options.push({
```

```
                    id: "custom",
                    width: 32,
                    value: "<span class='webix_fmanager_mode_option
webix_icon fa-file'/>"
                });
            }
        }
    }
});
```

RELATED SAMPLE: Adding New Mode

How to add a new column in "table" mode?

You can add a new column into the "table" view. For this purpose you should use the onViewInit event (described in the above example) to customize the view and insert column configuration into the "columns" array of the table.

Here is an example of inserting a new column after the "Date" column:

```
var fManager = new webix.ui({
    view: "filemanager",
    on:{
        "onViewInit": function(name, config){
            if( name == "table" ){
                // an array with columns configuration
                var columns = config.columns;
                // configuration of a new column
                var newColumn = {
                  id: "new",
                  header:"Extension",
                  fillspace: 1,
                  template: function(obj,common){
                      return obj.ext||""; // "ext" property of files
                  }
              };
              // insert a column
              webix.toArray(columns).insertAt(newColumn,3);
            }
        }
    }
});
```

RELATED SAMPLE: Adding New Column

How to customize existing views?

To illustrate this possibility, let's turn off multiple selection for the 'table' and 'files' views".

We can use the onViewInit event to customize configuration of exiting views:

```
webix.ready(function(){
    webix.ui({
        view:"filemanager",
        id:"files",
        on:{
            "onViewInit": function(name, config){
                // disable multi-selection for "table" and "files" views
                if (name == "table" || name == "files"){
                    config.select = true;
                }
            }
        }
    });

    $$("files").load("../common/data.php");
});
```

RELATED SAMPLE: Changing View Configuration

Guides

You are in the heart of the documentation that covers tricks, tips and techniques for working with the library. The chapter is divided into subject areas, each of which contains a set of task-oriented articles. Articles are of various difficulties and both a beginner and a master can find there a helpful information to build a powerful application.

Designing User Interface

- Building App Architecture
- Structuring a Big Application
- Initialization from Markup

Describes different ways you can follow to design and build the user interface for your application.

Loading and Storing Data

- Data Loading
- Defining Data Scheme
- Server-side Integration
- Data Binding and Syncing
- Data Updating
- Ajax operations

Teaches you how to load data into a component using different data sources, formats and techniques.

Configuring Components

- Changing Visibility
- Disabling/Enabling
- Adding/Removing Views
- Adding/Removing Multiview Cells
- Adding Headers
- Changing Properties
- Initial Sizing
- Resizing
- Managing Scrolling
- Localization

Says about various means you've got to configure a component initially and to dynamically change this configuration later: sizing/ resizing techniques, localization, changing configuration properties etc.

Managing Data in a Component

- Data Object Operations
- Selecting Items and Navigation
- Adding/Deleting Items
- Data Templates
- Working with Dates
- Filtering and Sorting
- Grouping Data Items
- Validation
- Editing Data Items
- Controls for Editing
- Paging
- Drag-n-Drop within Webix Views
- Working with the clipboard

Covers techniques for performing general and advanced operations under data, such as adding, deleting, sorting, filtering etc.

Styling and Animation

- Skins
- Component Styling
- Animation
- Icons with UI Components
- Setting Borders to the Components

Says how to style Webix application: styling techniques, css classes used in components, animation and visual effects.

Interacting with Users

- Creating Message Boxes
- Setting Progress Bar and Overlay Messages
- HTMLForm and Form Treatment
- Managing Controls in a Form
- Uploading Files
- Uploading Files with Form Component
- Event Handling
- Keyboard Events and App Navigation
- Adding Active Elements to Components

Describes various ways you can communicate with users: events, message boxes, forms and their controls

Web and Internet

- Webix Storage Interface
- Offline Support
- Saving Application State

Focuses on how to provide offline support for an app and save its state.

Extending components ◦ Component Copying and Extending ◦ Creating a Custom Component	If the wealth of provided functionality is not enough for you, the library provides flexible means for extending existing components.
Third Party Integration ◦ Text Editor Integration ◦ Scheduler and Gantt Integration ◦ Chart Integration ◦ Maps Integration ◦ Integration with DHTMLX Components ◦ JQuery Integration ◦ Backbone Integration ◦ Angular JS Integration ◦ How Integrate any Thirdparty Tool Using a Custom Component	Says which third-party libraries and how can be integrated with Webix components
Auxiliary Resources ◦ Top 10 Helpers You Should Know About ◦ Webix in Touch Environment ◦ Advanced Drag-and-Drop ◦ Data drivers ◦ Number formatting methods ◦ Date formatting methods	Extra goodies in the Webix library

Designing User Interface

- Building App Architecture
- Structuring a Big Application
- Initialization from Markup

Beginner

Building App Architecture

Basic principles

Before using library components you need to draw the scheme of their arrangement and define the way they would be positioned towards each other. It can be done with the help of four interface-building components, namely:

- Layout
- Multiview
- Accordion
- Scrollview
- Carousel

All the interface-building components inherit from **view**, a base class or, simply put, a container where you ought to place the component before working with it.

Layout allows dividing a page into **rows** and **columns** to organize sections to put other components. All of them will be visible within the same page at the same time.

Similar functionality you can find with a form and toolbar components that house **controls** (input fields, checkboxes, buttons, etc.) organized under the same pattern as layout (**rows** and **cols**). See Controls Guide to learn the variety of controls offered by this lib.

Mutliview, carousel, accordion and scrollview allow presenting different views in one and the same area when you are short of space. They differ in the ways of switching between these views:

- for **multiview cells** you need a dedicated button to switch between them;
- **accordion items** are expanded and collapsed on clicking their headers;
- **scrollview** allows scrolling through the whole content it houses;
- **carousel panes** change in turn by clicking on them or swiping (on touch devices).

All layout types are inited within the **webix.ui** constructor and can be nested into each other:

```
webix.ui({
    view:"layout", rows:[
     {...}, //any component
     {...}
    ]
```

```
});
```

Layout

If you just divide the page into **rows** or **columns** (cols) - you'll get standard **layout** to display the components side by side.

```
webix.ui({
    rows:[
     {...}, //any component
     {cols:[
        {...},
        {...}
    ]
});
```

More about Layout.

Form

Form helps divide the page into **rows** and **cols** to create a layout for controls. Unlike **layout**, this component should be initialized directly, using the view name.

```
webix.ui({
    view:"form",
    elements:[
     {...}, //any control
     {cols:[
        {...},
        {...}
    ]
});
```

If form controls are placed into the **elements** array - they will be arranged in **rows**.

More about Form

Accordion

Accordion consists of items each of which takes a separate row or column with a header and can be collapsed and expanded.

```
webix.ui({
    view:"accordion", rows:[
        {view:"accordionitem", header:"Item1", body:{ ...}},
        {header:"Item2", body:{...}}
```

```
    ]  //two ways of accordion item initialization are equal
});
```

If you place an array of objects with **body** and **header** into layout row/column - you'll automatically get **accordion** with **accordion items** without direct initialization.

```
webix.ui({
    rows:[
        {header:"Item1", body:{ ...}},
        {header:"Item2", body:{...}}
    ]
});
```

More about Accordion.

Scrollview

Scrollview is standard layout with a scrollbar. It can house any HTML content, any layout or component in its **body**:

```
Scrollview

webix.ui({
    view:"scrollview",
    body:{
        rows:[{...}, {...}]
    },
    scroll:"y" //scrolling direction
});
```

More about Scrollview.

Carousel

Carousel rows and **cols** are shown in turn, one by one, on clicking dedicated buttons or swiping on touch devices:

```
Carousel

webix.ui({
    view:"carousel",
    cols:[
        {...},
        {...}
    ]
});
```

More about Carousel

Multiview

Multiview is a layout with multiple hidden views that are shown in turn with one view being always visible.

```
webix.ui({
    view:"multiview", //optional line
    cells:[
        {...},
        {...}
    ]
});
```

Note that for mutliview you should provide the logic for switching between views. It can be a tabbar or a segmented button.

If you just define **cells** you'll automatically get **multiview**, if you add **tabs** to **cells** within one and the same constructor - it will turn into **tabview**. In this case, a switching button is not needed as tabs perform this function.

```
webix.ui({
    view:"tabview", //optional line
    tabs:[
        {id:"1", ...},
        {id:"2", ...}
    ],
    cells:[
        {id:"1", ...},
        {id:"2", ...}
    ]
});
```

More about Multiview and Tabview.

Isolating IDs inside Layouts

Basic rules of ID assignement can are described here.

The features is true to all the layout types described above. It allows for **using the same IDs** for components lying in different layout sections while preserving the ability to **access the needed component** without ambiguity.

Layout section should be equipped with an isolate property:

```
webix.ui({
   cols:[
```

```
      { id:"col1", isolate:true, rows:[
         { view:"list", id:"mylist" },
      ]},
      { id:"col2", isolate:true, rows:[
         {view:"list", id:"mylist" },
      ]}
   ]
});
```

The list view is used twice in the "app" but in different layout columns with **isolate** functionality switched on.

To address such lists separately, use yhe ID of its parent layout section:

```
var list1 = $$("col1").$$("mylist");
var list2 = $$("col2").$$("mylist");

//Otherwise, you'll get the last view
var list2 = $$("mylist");
```

What's Next

- To learn how to set various **border types** or create a borderless design, consult the dedicated article.
- To learn how to **set headers** for components in layout rows, cols, multiview cells, etc, study the corresponding article.
- Layouts can be **sized** according to common rules described in the Sizing Components article.
- Layouts can respond to changes of window size. Learn how to tune responsive behaviour.

View

View is a base class for all the ui-related components. It's a place where you put them before working. A pure view doesn't contain any information yet it may come with borders.

View performs a function of a placeholder to the layout components and layout itself. All of them are initialized according to the following scheme:

```
webix.ui({
    view:"dataview",
    ...
});
```

View ID

Views themselves can be put into other views thus creating a complex app structure with parent and child components. Any view features its unique **ID** that differentiates it from other views in the application:

```
Default ID

var layout = webix.ui({
    rows:[
        {view:"list", ...}, //gets "$list1" ID
        {view:"list", ...} //gets "$list2" ID
    ]
});
```

Then, you can get the objects of these views as:

```
//getting list 1 object
var list1 = layout.getChildViews()[];
..or
var list1 = $$("$list1");

//getting list 2 object
var list2 = layout.getChildViews()[1];
..or
var list2 = $$("$list2");
```

Still, it's much more convenient to define custom "speaking" IDs to the views.

```
Custom ID

webix.ui({
    id:"lists",
    rows:[
        {view:"list", id:"order-list"},
        {view:"list", id:"data-list"}
```

```
    ]
});
```

Then, you can get the objects of these views as:

```
//getting list 1 object
var list1 = $$("lists").getChildViews()[];
..or
var list1 = $$("$order-list");

//getting list 2 object
var list 2 = $$("lists").getChildViews()[1];
..or
var list2 = $$("$data-list");
```

Working within several views you can refer to the components with respect to their hierarchy or by using the view's unique ID.

Note that:

- Using auto-generated IDs is unsafe as they may change dynamically;
- The ID of any component can be found in its object as **view.config.id**;
- View IDs are reflected in HTML as a **view_id** attribute of a topmost div of a component;

Empty View

Views can as well be **empty**. An empty view is two braces **{}** you put btween meaningful components to leave space between them.

```
Spacer between buttons on toolbar

webix.ui({
    view:"toolbar", cols:[
        {view:"button", value:"but1" },
        {},
        {view:"button", value:"but2"}
    ]
})
```

Views in HTML

Views can be placed into **HTML containers** and take the dimensions of these containers (if any)

```
//body section
<div id="mylayout" style="width:600px;height:400px;"></div>

//script section
```

```
webix.ui({
    view:"layout",
    container:"mylayout",
    rows:[...]
    ...
});
```

Basic View Methods

1 . getParentView() & getTopParentView()

The method is called from any view and is used to get the parent object of this view.

```
webix.ui({
    view:"layout", id:"my_layout", rows:[
        {view:"toolbar", id:"my_Bar", cols:[
            {view:"button", id:"my_button", ..}
        ]},
        {view:"template", ...}
     ]
});

var t = $$("my_button").getParentView(); //  returns toolbar object
var l = $$("my_button").getTopParentView(); //  returns layout object
```

2 . getFormView()

The method returns an object of a **parent form** for the control it is called for. It's especially useful in case of complex forms:

```
webix.ui({
    view:"form", id:"my_form", elements:[
        {view:"text"},
        {cols:[
            {view:"combo", id:"my_combo", ..}
        ]},
        {view:"checkbox", ...}
     ]
});

$$("my_combo").getFormView(); //returns "my_form" object
```

3 . getChildViews()

This method, on the contrary, returns, an array of all the children of the calling component. Array members are numbered starting from 0.

For instance, if you have a toolbar within the layout and the toolbar itself comes with three buttons, the function will return the following:

```
var bar_id = $$("my_layout").getChildViews()[].config.id; //toolbar ID

var but_id = $$("my_toolbar").getChildViews()[1].config.id //ID of the 2nd
button
```

4 . index()

Like the above mentioned one, this method works with views containing other views that are stored in an array (the property may be called *cols, rows, elements, options* depending on the component).

Getting the view's index means getting its position in the array (zero-based numbering.) Method is called from the parent view and takes the ID of the necessary child view as parameter.

```
webix.ui({
    rows:[
        {id:"cell1", view:"list", ...},
        {id:"cell2", view:"form", ...},
        {id:"cell3", view:"dataview", ...}
    ]
});

$$("mylayout").index($$("cell1")); //-> returns 0 for the 1st array
element
```

5 . getNode()

The method helps us get an HTML container of the component.

```
<div id="my_box"></div>
...
webix.ui({
    view:"toolbar", id:"my_bar", ...
});

var box = $$("my_bar").getNode;
```

6 . getInputNode()

The method works with form controls defined by **input** tag in HTML, the simplest being text. It returns DOM element related to the input with all its properties.

```
{view:"form", elements: [
    { view:"button", id:"b1", value:"Login", type:"form" },
    {...}
]}
```

```
$$("b1").getInputNode();//-> <input type="button" style="height:27px;
width:149px" value="Login">
```

Dimensions here are library defaults for the button.

See the corresponding chapter of API Reference to learn about view's methods, parameters and events.

Responsive Layouts

Webix layout and accordion structures can respond to the changes of window size by hiding & showing or moving their rows (cols) depending on currently available space.

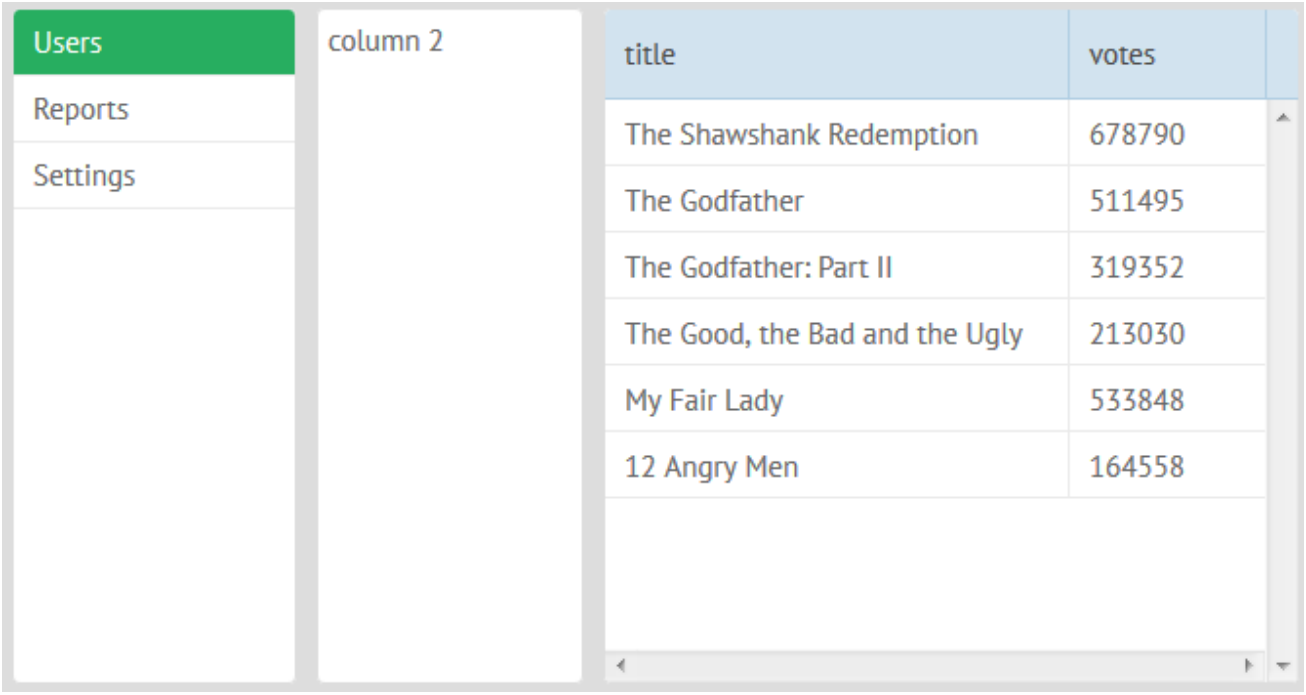
It occurs when:

- Layout or accordion are inited without an HTML container, and hence, feature no fixed sizing;
- The size of any component in such layout/accordion is bigger than the space available for its rendering. Fixed sizing is the right way to achieve it. Read more on initial component sizing.

Responsive behavior features 2 modes:

- Views are hidden and can be shown back;
- Views are moved to another place in layout.

Initial Layout



Responsive Hiding

To enable view hiding and showing on window resizing, set **responsive** property of a layout/accordion to *true*.

```
webix.ui({
    view:"layout", //optional line
    responsive:true,
    cols:[
         {view:"list", ...},
         {view:"template", ...}
         {view:"datatable", ...}
```

```
    ]
});
```

RELATED SAMPLE: Responsive Layout - Hidden cells

When space is not enough the last view from the left is hidden first.

After Window Resizing



Responsive Moving

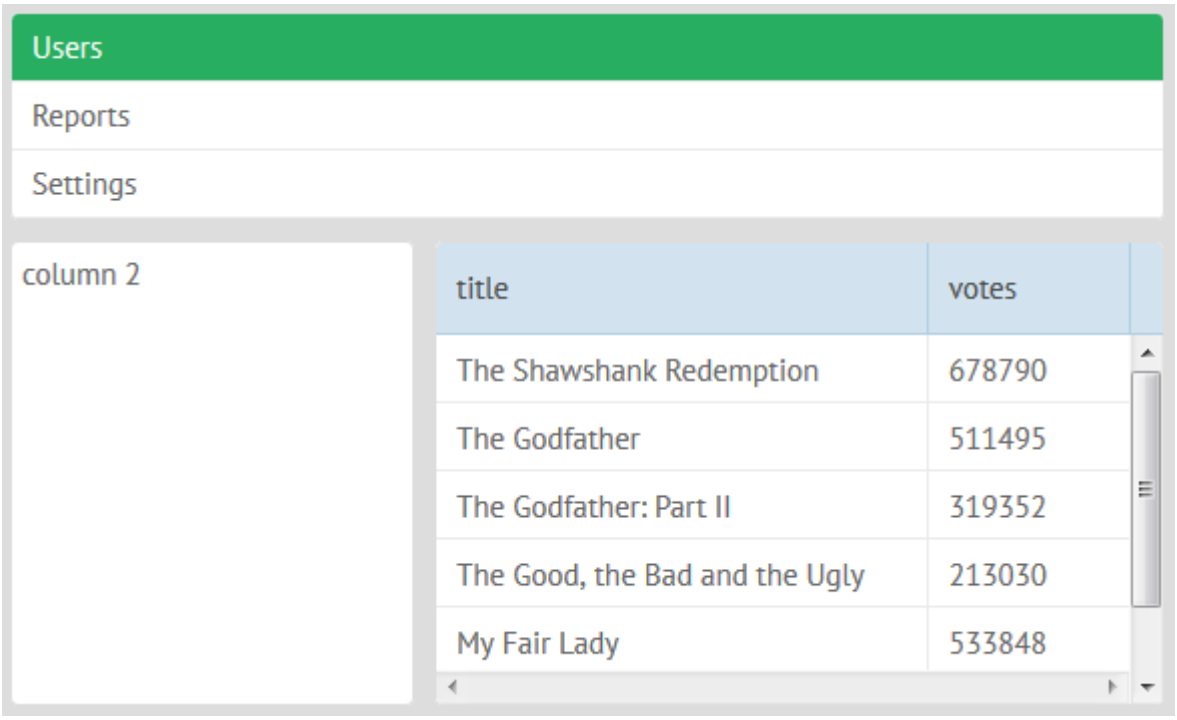
To enable component moving during window resizing, provide an ID of the target layout as a value of **responsive** attribute of a parent layout of the needed component:

```
webix.ui({
    id:"a1", rows:[
        { responsive:"a1", cols:[ //parent layout for template
            { view:"list", ...},
            { view:"template", ... }, //template will be moved to "a1"
layout
            { view:"datatable", ... }
        ]}
    ]
});
```

When space is not enough middle and right views are more likely to be moved.

RELATED SAMPLE: Responsive Layout

After Window Resizing



If accordion items are placed into layout during resizing, they will behave like layout rows and cols.

For layouts and accordions that lie inside an HTML container the functionality can be switched on by calling resize() method after the container has been resized:

```
webix.event("divId", "resize", function(e){
    $$("layout").resize();
});
```

Intermediate

Structuring a Big Application

The article tells about **logic-based app structuring** and shows how to divide the application code into semantic blocks, which enhances its readability and simplifies code amends including the addition of some new functionality.

For better undertanding, please study the following articles:

- Start Coding with the Webix Library
- Quick Start Guide, or How to Initialize a Component
- Building App Architecture - Layout and its Types

Here everything is about logic, structure and optimized coding. The **basic pronciples** are:

- store big code snippets in variables;
- put apart application design and inner functionality.

For samples, please look through the demo applications provided with this library.

Constructing Methods

The two main functions to construct the application are as follows:

- **webix.ui()** - the object constructor that creates a new instance of the specified view. Must be used anyway.
- **webix.ready()** - optional - a handler that is called just after the page has been completely parsed. The alternative to the **onload()** method annd **onDocumentReady** event.

```
In short, it look like this:

webix.ready(function(){
    webix.ui({
        view:"list", ... //any component including "layout"
    });
});
```

Inside the **webix.ui** constructor you define the layout of your app, divide it into rows and columns and place UI component into it. Details here

Putting Apart Logic and Design

Design here means layout type (layout, mutliview, accordion) as well as the UI components you've chosen for your application and the controls you insert into them.

Logic here means any functions you attach to the components and controls as well as those establishing links between components.

For a clear code it's recommended to store logic and design separately in variables (call them **logic** and **ui_scheme**), or in case of really big apps - in separate files.

Three-file application includes:

- **index.html** file that contains:
  - standard **HTML markup** with head and body sections;
  - **library scripts** included into the document head;
  - **webix.ready** function with **webix.ui** constructor in the script part of the document body;
  - links to **ui** and **logic** javascript files (look below);
  - links to CSS files and those storing inline JSON and XML data;
- **webix.js** file that contains all the components included into the **ui_scheme** variable(any name can be used);
- **logic.js** file with all the functions you need to build a fully interactive app. All of them are stored in the **logic** variable (again, the variable name is to your choice).

You can store the whole application in one and the same index file, but the division of the script into logic and design sections is a must. Just put components in the **ui_scheme** variable and functions into the **logic** variable.

webix.ready Function - "index.html" File Contents

HTML for such file looks like this:

```
<!DOCTYPE html>
    <head>
        <!--must-have files-->
        <link rel="stylesheet" href="../webix.css" type="text/css"
media="screen" charset="utf-8">
        <script src="../webix.js" type="text/javascript"
charset="utf-8"></script>

        <!--additional files-->
        <script type="text/javascript" src='./logic.js'></script>
```

```
        <script type="text/javascript" src='./webix.js'></script>

        <link rel="stylesheet" type="text/css" href="./styles.css">
    </head>

    <body>
        <script>
         ... place for you app
</script>
</body>
</html>
```

The app is placed into document body embraced by script tags. Everything lies within the **webix.ready** function:

```
webix.ready(function(){
    webix.ui(ui_scheme);
    logic.init(); //reference to logic section
 });
```

- If there any **dates** in your app - define common formatting methods for them right now.
- If you have components that are **beyound the application layout** (e.g. a popup window called on a button clicking) - put it into another *webix.ui* constructor right now.

```
webix.ready(function(){
    webix.ui(ui_scheme);
    webix.ui(ui_window);
    logic.init(); //reference to logic section
    webix.i18n.parseFormatDate = webix.Date.strToDate("%m/%d/%Y");
 });
```

Inside the webix.ui constructor layout as well as components are placed. Here, we've put a variable there, to be exact - the ui_scheme and ui_window variables that store these components.

Design Section - "webix.js" File Contents

The variables from above are declared separately, **outside the webix.ui constructor**, either in the same file within the *webix.ready* fucntion, or (better) in a separate **webix.js** file.

They should also be as short as possible:

```
var ui_scheme = {
    rows:[grid, footer],
    maxWidth:800
};
```

Grid and *footer* and variables declared separately and placed into the two-row layout. This is rather a simple application.

It's recommended that components with options (tabbar, toolbar), forms, non-visible dataCollections should be included into the main app in the form of variables declared beforehand.

Any component or several connected components can be stored in the variable that is later on included into the main layout variable(here:*ui_scheme*) that is placed into the webix.ui constructor.

Logic Section - "logic.js" File Contents

Logic block is stored in a **logic variable** declared either in a separate **logic.js** file (better), or somewhere outside the *webix.ui* constructor but within the *webix.ready* function.

Logic block is divided into the functions called during component initialization and the definition of all the fucntion used for the application.

The **init()** function includes:

- all the events attached to the components (except for those attached to them within their bodies). See Event Handling for better understanding.
- links between components - binding and syncing.

```
var logic = {
    init: function(){
        //ties between views
        $$("form1").bind($$("datatable1"));
        $$("datatable1").sync($$("list1"));

        //biult-in functions
        $$("list1").attachEvent("onAfterSelect", function(id){
            $$("datatable1").select(id);
        });

        //custom functions
        $$("datatable1").attachEvent("onItemClick", logic.delete_row);
    }
}
```

For custom functions attached either within component/control body or within init block (see above) use the following reference:

```
logic:function_name //since they are placed into the "logic" var
```

Outside the init block you should describe all the custom functions attached to application components either in the init block or within their bodies:

```
delete_row: fucntion(){
    $$("datatable1").remove(id);
}
```

Other Possibilities

Sometimes the variable that stores main layout structure (ui_scheme) is included into the init function of the logic block and the whole script in the index file contains just a line:

```
ui.html

logic.init();.
```

```
logic.js

var logic = {
    init:function(){
        webix.ui(ui_scheme);
        //some additional logic defined outside the init block (if needed)
        logic.init_filtering();
    }
}
```

It goes without saying that a long list of custom **styles** is taken out into a separate file that is included into an index file with a help of a **link** in its head section (together with library styles).

Images for the application as well as custom icons (exept for built-in ones) are stored in a separate folder, as a rule, in the same directory as the index file or somewhere nearby.

Advanced

Initialization from Markup

Library components are JSON objects in essence. Each of them is constructed with a set of **key:value** pairs that describe its properties.

```
webix.ui({
    view:"toolbar",
    width: 300,
    elements:[
        {view:"button", value:"OK", inputWidth:100},
        {view:"button", value:"No", inputWidth:100}
        ]
});
```

The toolbar from above is made of three members (*view, width, elements*), the last one containing an array of two members (*two buttons*).

At the same time, components can be initialized not only by literal notation, but also within HTML and XML markup.

Bonuses of markup initialization

- you needn't define template to enable data rendering).Still, you should use property names in hash signs;
- datasource can be defined in frames of unordered lists or HTML tables.

License restrictions

- **XML** initialization is available in **Webix Pro** edition only;
- **HTML** initialization can be used with **any** Webix edition.

Initializing from Markup

XML and HTML markup initialization can totally replace JSON configuration for people who are more used to working with markup languages. It has simpler syntax so you are less likely to get lost among brackets and commas. Such functionality can be useful when you code the page with the help of server scripts since it's easier to get XML (HTML) data from them than JSON.

In general, the markup can be taken from:

- any **node** of this document including **body**;
- **separate file** with HTML or XML markup;
- response of **server script**;

HTML and XML markup get **peculiar tags** to serve the needs of Webix object initing (described later).

Webix HTMLform component is more likely to initialized from HTML and XML, yet any component is subject to this technique.

Working with Webix Markup Class

To init Webix components from HTML or XML, you should work with **webix.markup** class that offers a pattern for markup processing.

The main methods here are **init**(node, target) and **parse**(node) where arguments mean:

- **node** - which data to use for initialization. By default data is taken from **document.body**.
- **target** - where should the newly created component be placed. By default it's placed into the HTML node defines by **node** (document.body, if it's not defined as well).

Initing and parsing serve different purposes:

- The **init()** method inits Webix object in HTML layout.
- The **parse()** method inits the component in Webix layout.

The simplest initialization script looks as follows:

```
// data is taken from document body and component is inited there
webix.ready(function(){
    webix.markup.init();
});
```

External-file or **serverside data** is loaded to the page by **Ajax**. Here you should specify path to the file(script) and set initialization pattern in a callback.

In the code below, data comes from external XML file, the **node** is incoming data while the **target** is body of the current document.

```
webix.ajax("data/config.xml?!", function(text, data){
    var config = webix.markup.parse(text, "xml");
    webix.ui(config);
});
```

RELATED SAMPLE: Load UI by ajax

You can use markup initialization by **parsing markup** to the neccessary part of the Webix script thus creating a Webix object.

```
webix.ui({
    rows:[ {}, {id:"placeholder", template:"html->dummy" }
    ]
});

function load_grid() {
    webix.ajax("data/gridconfig.xml", function(text, data){
        webix.ui(
            webix.markup.parse(text, "xml"), //source
            $$("placeholder")) //target
    })
};
```

Target is not defined within the **parse()** function yet it is specified in the **webix.ui()** constructor as *$$('placeholder')*.

Markup Peculiarities

The rules of HTML and XML initialization differ, so you should consult dedicated articles to learn them.

- Initialization from HTML;
- Initialization from XML - available **Webix Pro** edition only.

Related Articles

- HTMLform

Initialization from HTML Markup

See also **general rules of markup initialization** that are described in separate article.

Initialization from HTML is a nice feature for those who work in HTML mainly. By the way, a similar technology is used within JQuery Mobile framework.

Basic principles

1. DIV blocks as components

The page is divided into **div** blocks that house Webix components defined by their **data-view** attribute:

- *data-view="component_name"* to initialize most of Webix components;
- *data-view="cols"* or *data-view="rows"* to initialize Webix layout;

```
<div data-view="rows"> <!--layout-->
    <div data-view="template" data-height="35"> My header</div> <!--
template-->
</div>
```

2. "Data" attributes as component properties

DIV "data" attributes set other properties of the component:

```
<div data-view="text" data-name="email" data-label="Email"></div>
```

In other words, you use the same properties as in JSON configuration, but prefix them with "data-".

3. Inner HTML matters

Inner HTML of a DIV block defines visible text:

- text of a button, template, text, textarea, etc.:

```
<div data-view="button" data-width="150">Button</div>
<div data-view="text" data-label="Phone">123456789</div>
<div data-view="template" data-height="35"> My header</div>
```

- text of a window head:

```
<div data-view="window" data-id="win1">
```

```
    <div data-view="head">Test header</div>
    ...
</div>
```

RELATED SAMPLE: HTML initialization : Window

- data template for data management components:

```
<div data-view="list" data-url="data/data.json" data-select="true">
    Package: #Package# (#Version#)
</div>
```

RELATED SAMPLE: HTML initialization

4. Specific tags and attributes

- **data-stack** - defines array or sub items such as rows or cols for layout and the like, and headerlayout (accordion) components:

```
<div data-view="layout" data-stack="cols">
    <div data-view="template">Cell 1.1</div>
    <div data-view="template">Cell 1.2</div>
</div>
```

RELATED SAMPLE: HTML initialization : Nested layouts

```
<div data-view="headerlayout" data-stack="cols">
    <div data-view="body" data-header="Section A">
        <div data-view="list" data-url="data/data.json"> #id#: #Package#
</div>
    </div>
    <div data-view="body" data-header="Section B">
        <div data-view="list" data-url="data/data.json"> #id#: #Package#
</div>
    </div>
</div>
```

RELATED SAMPLE: HTML initialization : Accordion

By default, the sub items will be treated as **rows** for layouts and form and **cols** for toolbar.

```
<div data-view="toolbar"> <!--data-stack="cols"-->
    <div data-view="button" data-width="150">Button</div>
    <div data-view="label" data-label="Press the button to run App"></div>
</div>
```

RELATED SAMPLE: HTML initialization

5. Standard HTML

Standard HTML is used for:

- form and its elements:

```
<form>
    <input type="text" name="title" value="" placeholder="Book title"
/><br/>
    <textarea name="annotation" id="annotation" rows="4">some book
annotation is here</textarea><br/>
</form>
```

Such code can be easily turned into Webix Htmlform component:

```
<form data-view="htmlform" data-id="formView">..
```

RELATED SAMPLE: HTML initialization : HTML form

- inline data within data presenting components:

```
<div data-view="list" data-width="320">
    #id#: #value#
    <ul data-view="data">
        <li id="9998">Item 1</li>
        <li id="9999" class='myitem'>Item 2</li>
    </ul>
</div>
```

Note that inline **datasource** is marked by *data-view* attribute with *data* value.

RELATED SAMPLE: HTML initialization

Datatable

Datatable and treetable columns are built from child DIVs of the main component container:

```
<div data-view="datatable" data-autoheight="true" data-width="400">
    <div data-view="column" data-width="150" data-id="firstname">
        First Name
    </div>
    <div data-view="column" data-width="150" data-id="lastname">
        Last Name
    </div>
    ...
</div>
```

- "data" attributes are used to set column properties;

- inner HTML of column DIVs is turned into their headers.

RELATED SAMPLE: HTML initialization : DataTable

Pager

When initialized from markup Webix pager should be placed into a separate DIV block while the master component should refer to it by its ID:

```
<div data-view="datatable" data-pager="mypager">
...
</div>
<div data-view="pager" data-id="mypager" group="10" size="3"></div>
```

Note that pager properties are not prefixed with "data-".

RELATED SAMPLE: HTML initialization : DataTable with Pager

Tabview and Accordion

Webix tabview and accordion are made of separate blocks, each of which featuring **header** with title and **body** with content inside.

In markup you need to place each block into a separate DIV with the attributes:

- **data-view** with **"body"** value;
- **data-header** with header text.

Tabview

```
<div data-view="tabview">
    <div data-view="body" data-header="List 1">
        <div data-view="list" data-url="data/data.json"> #id#: #Package#
</div>
    </div>
    <div data-view="body" data-header="Text 2">
        <div data-view="template">Just some text here</div>
    </div>
</div>
```

RELATED SAMPLE: HTML initialization : Tabbar

Accordion

```
<div data-view="accordion">
    <div data-view="body" data-header="Section A">
        <div data-view="list" data-width="320" data-url="data/data.json">
#id#: #Package# </div>
```

```
    </div>
    <div data-view="body" data-header="Section B">
        <div data-view="list" data-width="320" data-url="data/data.json">
#id#: #Package# </div>
    </div>
</div>
```

RELATED SAMPLE: HTML initialization : Accordion

Chart

Chart axes are defined by **config** tags in HTML markup:

```
<div data-view="chart" data-type="stackedArea" data-color="#58dccd" data-
value="#sales#"
  data-alpha="0.7" data-url="data/chart.json">
    <config name="xAxis" template="#year#"></config>
    <config name="yAxis"></config>
</div>
```

Axis names are defined by **name** attributes while other attributes are not prefixed by "data-" and are the same as in JSON configuration.

chart Series

If **series** are needed for the chart they are initialized via a separate **config** object:

```
<div data-view="chart" data-type="stackedArea" alpha="0.7" eventRadius="5"
data-url="data/chart.json">
    <config name="xAxis" template="#year#"></config>
    <config name="yAxis"></config>
    <config stack='series'>
        <config  value="#sales#" color="#58dccd">
            <config name="tooltip" template="#bottom#"></config>
        </config>
        <config  value="#sales2#" color="#dc58cd">
            <config name="tooltip" template="#bottom#"></config>
        </config>
        ...
    </config>
</div>
```

The attributes of series are not prefixed by "data-" and are the same as in JSON configuration.

RELATED SAMPLE: HTML initialization

Windows and Menus

Window peculiarities

When initializing Webix window from markup, you need to remember the following things:

- window should be set on the same level as other layout components;
- window is comprised of two parts - **head** and **body** that should be defined via separate DIV blocks.

```
<div data-view="rows">
    <!--layout config>-->
</div>
...
<div data-view="window" data-id="win1">
    <div data-view="head">Test header</div>
    <div data-view="body">
        <div data-view="template">Cell 1.2</div>
    </div>
</div>
```

- in addition, window show method should be called as it won't appear by itself:

```
webix.markup.init();
$$("win1").show();
```

RELATED SAMPLE: HTML initialization : Window

Menu and submenu peculiarities

If menu is initialized from markup it's vital to remember that its submenus should be as well inited the same way as windows:

```
<div data-view="submenu" data-id="submenu1">
    <ul data-view="data">
       <li data-id="1.1" data-submenu="submenu3">Item 1.1</li>
       <li data-id="1.2">Item 1.2</li>
    </ul>
</div>
```

And then they should be included into the main menu (or other submenus) with the help of their **submenu** property (submenu ID should be specified):

```
<div data-view="menu" data-height="50">
    <ul data-view="data">
        <div data-view="menu">
            <ul data-view="data">
                <li data-id="1" data-submenu="submenu1">Item 1</li>
                <li data-id="2" data-submenu="submenu2">Item 2</li>
            </ul>
```

```
        </div>
        <li data-id="3">Item 3</li>
    </ul>
</div>
```

Any level of nesting is possible.

RELATED SAMPLE: HTML initialization::Menu

Available only in **PRO Edition**

Initialization from XML Markup

See also general rules of markup initialization that are described in a separate article.

Initialization from XML is simpler in terms of syntax while at the same time tags feature greater semantic value than HTML, yet XML is not that popular due to its "strict" nature.

To intergate this functionality with IE8, set the initial page tag as **< html xmlns:x >** instead of < html >.

Below the peculiarities of initialization for different Webix components are listed.

Layout with different components

```
XML Toolbar, List and Form

<x:ui>
        <x:rows>
            <x:template>My header</x:template>
            <x:toolbar>
                <x:button width="150">Button</x:button>
                <x:label>Press the button to run App</x:label>
            </x:toolbar>
            <x:cols>
                <x:list url="data/data.json" select="true">
                    <x:template>Package: #Package#
(#Version#)</x:template>
                </x:list>
                <x:htmlform>
                    <input type="submit" />
                </x:htmlform>
                <x:list width="320">
                    <x:template>#id#: #value#</x:template>
                   <x:data>
                        <li id="9998">Item 1</li>
                        <li>Item 2</li>
                   </x:data>
                </x:list>
            </x:cols>
            <x:template>My footer</x:template>
        </x:rows>
    </x:ui>
```

RELATED SAMPLE: XML initialization

Comments:

- semantially rich tags coincide with component names. The pattern such for tags is as follows: **< x:name >**;
- **x:** namespace for tags that form Webix component including Webix **ui** constructor(**< x:ui >**);
- component properties come as tag attributes;
- standard HTML tags perform their function in the way they do it in HTML (< li >);
- Form in layout comes in standard HTML tags while **form** not placed in layout requires specific tags as well (see later).

Accordion

Accordion features an array of **accordionitems**, each of which has a **header** and **body** where any Webix component can be placed.

JS initialization

```
{
    view:"accordion", rows:[
        {header:"Section A", body:{view:"list", url:"data/data.json",
template:"#id#: #Package#"} },
        {...}
    ]

}
```

XML initialization

Remember that anything you init from XML, should be among **< x:ui >** tags!

```
<x:accordion>
    <x:body header="Section A">
        <x:list width="320" url="data/data.json"> #id#: #Package#
</x:list>
    </x:body>
</x:accordion>
```

Or:

```
<x:headerlayout stack="cols">
    <x:body header="Section A">
        <x:list url="data/data.json"> #id#: #Package# </x:list>
    </x:body>
</x:headerlayout>
```

RELATED SAMPLE: XML initialization : Accordion RELATED SAMPLE: HTML initialization : Accordion

Datatable

Datatable features an array of columns each of which has a set of congiguration options:

JS Initialization

```
{
    view:"datatable",
    autowidth:true, autoheight:true,
    columns:[
        {id:"rank", header:"", width:250, sort:"int"},
        {id:"title", header:"Film title", width:200, sort:"string"}
    ]
}
```

XML Initialization

```
<x:datatable autowidth="true" autoheight="true">
        <x:column id="rank"  header=""           width="50"
sort="int"></x:column>
        <x:column id="title" header="Film title" width="200"
sort="string"></x:column>
</x:datatable>
```

RELATED SAMPLE: XML initialization : DataTable

Form

If placed in layout rows or cols, form can be inited with standard HTML tags. **Standalone form** not placed in layout requires tags with **x:** prefix.

```
<x:form id="log_form" width="300">
    <x:text label="Email"></x:text>
    <x:button value="Login" type="form"></x:button>
</x:form>
```

RELATED SAMPLE: XML initialization : Fieldset

HTMLform has its own initing patterns including via webix markup class.

Tabview

Tabview is a hybrid layout that features several view cells and dedicated tabs to switch between these cells.

JS initialization

Tabs and cells are connected by their **IDs** that must coincide.

```
{
 view:"tabview",
 tabs:[
    { value: 'List 1', id: '1'},
    { value: 'Text 2', id: '2'}],
 cells:[
    { view: 'list', id: '1', url:"ata/data.json", template:"#id#:
#Package#"}
    { view: 'template', id: '2', template:"Just some text here"}]
}
```

XML Initialization

During XML init you needn't define IDs - just place necessary conponents within **body tags** that substitute **cells**. At the same time, **header attrbute** of this tag defined **text value** for a **tab**.

```
<x:tabview>
    <x:body header="List 1">
        <x:list url="data/data.json"> #id#: #Package# </x:list>
    </x:body>
    <x:body header="Text 2">
        <x:template>Just some text here</x:template>
    </x:body>
</x:tabview>
```

RELATED SAMPLE: XML initialization : Tabbar

Altering Namespace

You can change the default **x:** namespace as well as completely get rid of it. You need to define the new namespace before initialization from markup:

```
webix.markup.namespace = ""; //empty namespace
webix.markup.namespace = "wx"; //new namespace

webix.markup.init();
```

From now on, you can use initialize Webix components using standard tags, or tag with your custom namespace:

```
Empty Namespace

<ui>
    <form id="log_form" width="300">
        <text label="Email"></text>
    </form>
</ui>
```

RELATED SAMPLE: XML init : empty namespace

```
Custom Namespace

<wx:ui>
    <wx:form id="log_form" width="300">
        <wx:text label="Email"></wx:text>
    </wx:form>
</wx:ui>
```

RELATED SAMPLE: XML init : custom namespace

Webix in Touch Environment

Webix works equally well on desktop and touch devices **out of box**.

Touch support is enabled in all the Webix components with the **touch events** being translated into **native Webix ones** to ensure their flawless work. For instance, when you tap a list item, the onItemClick event fires for it. If selection is enabled for this list, list item is selected.

In addition there are global touch events that are not component-specific and fire for Webix application as a whole. All of them are described in detail in API Reference section.

Working with Webix Touch Module

Touch events are controlled by Webix Touch module where configuration for different aspects of touch behavior such as **touch speed**, **touch** and **scroll** delay can be changed.

Also the module allows for **disabling Webix touch events** in favour of native ones in either of the two ways:

- total disabling of Webix touch support - neither global events nor component-specific events fire;
- limiting Webix touch support - only in-component events work, global ones are disabled.

Related Articles

- Adjusting Mobile Apps to Screen Size

Loading and Storing Data

- Data Loading
- Defining Data Scheme
- Server-side Integration
- Data Binding and Syncing
- Data Updating

Beginner

Data Loading

After a data component has been inited, you need to populate it with data in any of supported formats: JSON, XML, CSV, JSArray, HTMLtable.

There we have two major possibilities that are true for all the data components.

- Loading from External Resource - File or Database
- Parsing from String

After data has been pushed to component (loaded or parsed) it should be rendered within the component with the help of Webix templates.

Loading from External Resource

To load data from an external file or database, you should specify the **path** to this file/or serverside script that fetches data:

1 . As value of the component **url** property. Data will be loaded right after component initialization. Type of incoming data should be specified (if it's not JSON) as well.

```
{
    view:"list"
    url:"data.xml"
    datatype:"xml"
}
```

2 . As a parameter to the load() function you apply to the component on the necessary event.

```
$$("mylist").load("data.xml");
```

Load function has three parameters:

- **path** to the necessary file or script;
- datatype - in case of JSON data this param can be omitted;
- **callback** function if needed.

```
$$('component_id').load(".../data.xml", "xml", function(){
  //callback function
});
```

RELATED SAMPLE: Form: Loading

It can be called on the necessary page event (button clicking, etc.) as well as on component inner event.

```
As soon as the component is parsed:

webix.ui({
view:"list",
ready:function(){
    this.load("data/books2.xml","xml");
    this.select(this.getFirstId());
    }
});
```

Loading via Proxy Objects

At the same time, you can set rather a complicated loading pattern in a **proxy object** you pass into the **url** property or **load()** function.

Webix features predefined proxy objects for offline support, connector usage, etc. To learn more about proxy functionality, consult the dedicated article.

```
view:"list"
url:{
    $proxy:true,
    load: function(self){
        self.load(a.json);
    },
     save:function(){
     ...
    }
}
```

Furthermore, if a serverside dataset is too big, dynamic loading allows to load limited quantity of data items first and then load new data only as you scroll or page through the component.

Note that there's a separate article dedicated to Serverside Integration. There you'll also find customization patterns for loading from server.

Setting an Alternative Datasource for a Bound Component

With **bound components** you can enable data loading not from master component (as expected by default), but directly from server.

```
{
    view:"list"
    url:"data/list.php" //the default datasource
},
{
    view:"form",
```

```
    id:"form11",
    ...
    dataFeed: "data/form.php" //datasource for the form only
}

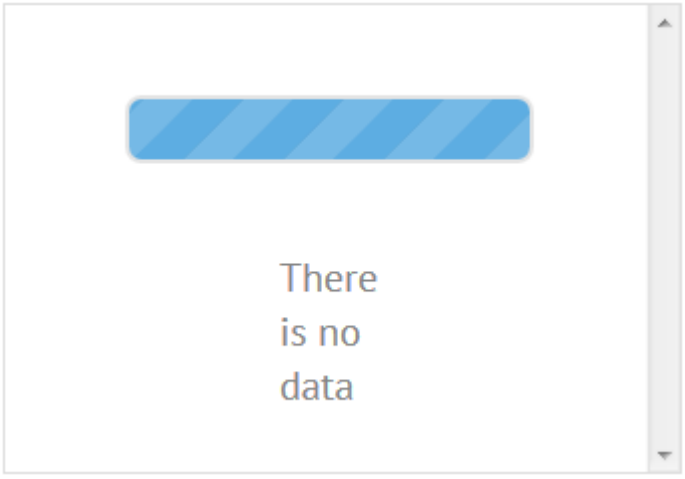
// form takes data from selected record in list
$$('form1').bind($$('list1'));
```

More on Data Binding

Loading Messages

Webix provides info boxes that are laid over the component for a while to display desired data, e.g.:

- to inform users that data is being loaded (in case of big datasets);
- to inform users that there's no data for this component.



Webix (datatable) features info message support out of the box in the form of **overlay boxes** with any text or html content:

```
dtable.showOverLay("Loading");
dtable.hideOverlay();
```

RELATED SAMPLE: Loading screen

More info on datatable loading screen support.

For other components there exists a possibility to define **Overlay Boxes** and **Progress Bar** (**Progress Icon**) with the help of the dedicated modules.

```
webix.extend($$("list1"), webix.OverlayBox); // on "this" pointer if the
method is called within the component

webix.ui({
    view:"list",
```

```
    id:"list1",
    url:"...",
    ready:function(){
        if (!this.count()){ //if no data is available
            webix.extend(this, webix.OverlayBox);
            this.showOverlay("<div style='...'>There is no data</div>");
        }
    }
});
```

RELATED SAMPLE: Empty list

More info on overlay boxes and progress bar

Data Loading Events

Webix offers the following events to control the beginning and the end of the loading process:

- **onBeforeLoad** - fires just the moment XML data loading begins;
- **onAfterLoad** - fires when XML data loading is complete;
- **onStoreLoad** - fires when raw external data is fully loaded (regardless of data type).

```
mygrid.attachEvent("onAfterLoad", function(){
    webix.message("Data loaded");
})
```

RELATED SAMPLE: Loading screen

In case something goes wrong with loading (and sometimes it does!) learn how to handle errors on the client side in the Processing loading errors article.

Parsing

Here the data in supported formats comes as it is (string or array).

Firstly, it may be the value of data parameter.

```
JSON array within the Dataview object

webix.ui({
    view:"dataview",
    template:"#title# Year: #year#, rank: #rank#",
    data:[
        { id:1, title:"The Shawshank Redemption", year:1994, rank:1},
        { id:2, title:"The Godfather", year:1972,  rank:2},
```

```
        { id:3, title:"The Godfather: Part II", year:1974, rank:3}
    ]
});
```

```
JSON object is stored in a variable declared beforehand

var tabledata=[
    { id:1, title:"The Shawshank Redemption", year:1994, rank:1},
    { id:2, title:"The Godfather", year:1972,  rank:2},
    { id:3, title:"The Godfather: Part II", year:1974, rank:3},
    { id:3, title:"The Godfather: Part II", year:1974, rank:3}
];

webix.ui({
    view:"dataview",
    template:"#title# Year: #year#, rank: #rank#",
    data: tabledata
});
```


The variable may be declared either in the same file or in an external js-file you refer to in the document's head.

Secondly, data can be parsed with the help of **parse();** method that takes data and datatype as parameters.

```
Data Parsing from XML string

webix.ui({
    view:"dataview",
    id:"dataview",
    datatype:"xml"
});

 //function is executed on pre-defined event, e.g. on button click

function parse(){
     var str = "<data>
            <item id='10'><title>The Lord of the Rings</title>
            <year>2003</year><rank>10</rank></item>
        </data>";
     $$("dataview").clearAll();
     $$("dataview").parse(str,"xml");
   }
```

RELATED SAMPLE: Dataview: XML Dataset

If you want to parse data in addition to the existing dataset, specify **position** of parsing, namely the **index** from which you insert new data:

```
$$("datatable").parse({
      pos: $$("datatable").count(), //number of records will be right the
last index +1
      data:dataset
  });
```

Data Parsing Events

- **onParse** - fires when the component is ready to receive inline data

```
dtable.attachEvent("onParse", function(driver, data){
    //for json data
    webix.message("Count of records "+data.length);
});
```

Promise API in Data Loading

Webix is integrated with Promiz.js library to treat the result of asynchronous operations (like data loading) without callbacks.

It means that any Ajax request returns a **promise** object than can be treated with Promiz.js API.

```
"Promise" objects is returned by either of these methods

var promise = grid.load("my.php");
var promise = webix.ajax("my.php");
var promise = webix.ajax().get("my.php");
var promise = webix.ajax().post("my.php");
```

Bonuses

1 .You may forget about callbacks and process the result of a request in Promiz.js way:

```
promise.then(function(realdata){
    //success
}).fail(function(err){
    //error
});
```

2 .You can "parse" **promise** objects directly into the component:

```
list.parse( webix.ajax("my.php"));
```

The data will be parsed the moment it comes from server.

3 . You can make use of **waitData** property each data component is supplied with and apply Promiz.js methods to it:

```
//"promise" approach
grid.waitData.then(function(){
    //when we have data - do some actions
});
```

Alternatively, Webix onAfterLoad event can be used:

```
//standard approach
grid.attachEvent("onAfterLoad", function(){
    //when we have data - do some actions
});
```

More about Webix and Promiz.js integration.

Beginner

Supported Data Types

The majority of data-presenting components take data in **XML**, **JSON**, JSArray and **CSV** formats. Still, three of them, namely grouplist, treetable and Tree support only XML and JSON datatypes since only they can show hierachy which is a key feature of these components.

- **Data sources** and **loading methods** are described in Data Loading section.
- **Data Drivers** that with processing lofic for each format are descriped in Data Drivers;
- **Customization of Data Types** by changing default Driver behavior - Customizing Data Types.

Data Type Samples

Here we'll see how one and the same dataset looks in different formats while paying attention to non-hierarchical and hierarchical data.

Non-hierarchical data

Person	Place	Age
Nanny	Alabama	45
Derek	New York	23
Samuel	Oregon	32

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<data>
    <item id="1" person="Nanny" place="Alabama" age="45"/>
    <item id="2" person="Derek" place="New York" age="23"/>
    <item id="3" person="Samuel" place="Oregon" age="32"/>
</data>
```

JSON Object

```
[
    { id:"1", person: "Nanny", place: "Alabama", age: "45" },
    { id:"2", person: "Derek", place: "New York", age: "23" },
    { id:"3", person: "Samuel", place: "Oregon", age: "32"}
]
```

JSArray

```
[
    [ "Nanny", "Alabama", "45" ],
    [ "Derek", "New-York", "23"],
    [ "Samuel", "Oregon", "32"]
]
```

CSV

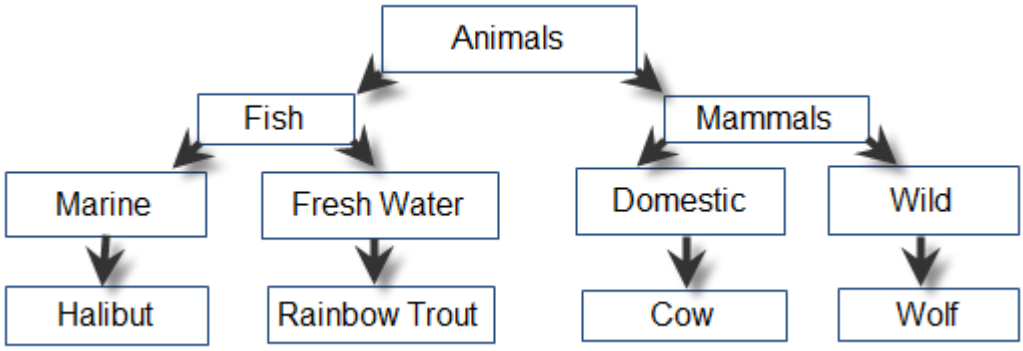
CSV, or Comma Separated Values, is a sequence of characters with rows and fields being separated by some breaks (comma, tab, new line).

```
Nanny, Alabama, 45
Derek, New-York, 23
Samuel, Oregon, 32
```

By default line anf field **delimiters** are "\n"(new line) and ","(comma) respectively.

To change the logic of CSV parsing for the application components, look up the related article.

Hierarchical Data



XML

By default records of any hierarchy level and their children are introduced with **item** tags:

```
<?xml version="1.0" encoding="UTF-8"?>
<data>
    <item value="animals" open="1">
        <item value="Fish">
            <item value="Marine">
                <item>Halibut</item>
            </item>
            <item value="Fresh Water">
                <item>Rainbow Trout</item>
            <item/>
```

```
        </item>
        <item value='Mammals'>
            <item value="Domestic">
                <item>Cow</item>
            </item>
            <item value="Wild">
                <item>Wolf</item>
            </item>
        </item>
    </item>
</data>
```

RELATED SAMPLE: Tree: XML Dataset

JSON

By default child records are stored in the **data** object property of the parent record.

```
[{id:"1", title:"Animals", data:[
    {id:"1.1", title:"Fish", data:[
        {id:"1.1.1", title:"Fresh Water", name:"Halibut"},
        {id:"1.1.2", title:"Marine", name:"Rainbow Trout"}
        ]
    },
    {id:"1.2", title:"Mammals", data:[
        {id:"1.2.1", title:"Domestic", name:"Cow"},
        {id:"1.2.2", title:"Wild", name:"Wolf"}
        ]
    }
    ]
}]
```

RELATED SAMPLE: Tree: JSON Dataset

Custom Tags for XML and JSON Data

Advanced

Customizing Data Types

By default JSON children are stored in the **data** property object while XML records (both parents and children) are introduced by an **item** tag. However, any custom tags will do provided that you **create a custom data format** based on a prebuilt one.

Parsing logic for each of the supported data types is stored in the **webix.DataDriver** object (webix.DataDriver.json, webix.DataDriver, xml, etc.). There you can change default tags in two ways:

- change the logic of the existing data driver - desrcibed here. All incoming data of the state type will be processed according to the new logic.
- copy the existing driver object and create a custom one on the base of it. The new processing logic will be applied only to data with the new data format you specify as **datatype.** Described below.

Custom Tags for JSON

```
Default JSON data with 2 nesting levels

[
    { id:"1", open:true, value:"The Shawshank Redemption", data:[
        { id:"1.1", value:"Part 1" data:[
            {id:"1.1.1", value:"Page 1"},
            {id:"1.1.2", value:"Page 2"}
        ]},
        { id:"1.2", value:"Part 2" }
    ]},
    { id:"2", value:"The Godfather", data:[
        { id:"2.1", value:"Part 1" }
    ]}
]
```

Declaring the new format:

```
var myjson = webix.DataDriver.myjson = webix.copy(webix.DataDriver.json);
```

Changing tags:

Elements of the first nesting level will be stored in **parts**, while the second-level ones will be stored in **pages.**

```
myjson.child=function(obj){
                if (obj.$level == 1)
                    return obj.parts;
                if (obj.$level == 2)
                    return obj.pages;
```

```
            };
//or, if you need the same property for all children
myjson.child="parts";
```

New Dataset:

```
[
    { id:"1", open:true, value:"The Shawshank Redemption", parts:[
        { id:"1.1", value:"Part 1" pages:[
            {id:"1.1.1", value:"Page 1"},
            {id:"1.1.2", value:"Page 2"}
        ]},
        { id:"1.2", value:"Part 2" }
    ]},
    { id:"2", value:"The Godfather", parts:[
        { id:"2.1", value:"Part 1" }
    ]}
]
```


Don't forget to specify the name of the new format as the **datatype** for your component!

```
webix.ui({
    view:"treetable",
    //..config
    datatype:"myjson"
});
```

RELATED SAMPLE: Tree: JSON Dataset

Custom Tags for XML

```
Default XML Data

<data>
    <item value='The Shawshank Redemption' open='1'>
        <item value='part1'></item>
            <item>Page 15</item>
            <item>Page 16</item>
        <item value='part2'></item>
    </item>
</data>
```

Declaring the new format:

```
var myxml = webix.DataDriver.myxml = webix.copy(webix.DataDriver.xml);
```

Changing tags:

```
myxml.records = "/*/book";

myxml.child = function(obj){
    if (obj.$level == 1)
        return obj.part;
    if (obj.$level == 2)
        return obj.page;
}
//or, if you need the same property for all children
myxml.child="part";
```

New Dataset:

```
<data>
    <book value='The Shawshank Redemption' open='1'>
        <part value='part1'>
            <page>Page 15</page>
            <page>Page 16</page>
        </part>
        <part value='part2'></part>
    </book>
</data>
```


Don't forget to specify the name of the new format as the **datatype** for your component!

```
webix.ui({
    view:"treetable",
    //..config
    datatype:"myxml"
});
```

RELATED SAMPLE: Tree: XML Dataset

Customizing CSV parsing logic

Global changes

The default parsing logic of CSV data can be changed **globally**:

```
webix.DataDriver.csv.cell = "\t";
webix.DataDriver.csv.row = "|";
```

Now, all the components that use "csv" datatype, will be parsed according to new rules.

Creating custom datatype

A custom datatype should be created in case you wish to apply it to one or several components while leaving the global CSV parsing unchanged.

```
webix.DataDriver.mycsv = webix.extend({
    row:"|",
    cell:"-"
}, webix.DataDriver.csv);
```

Now, you can use it in component instances:

```
var grid = webix.ui({
    view:"datatype",
    datatype:"mycsv",
    data:'1-The Shawshank Redemption|2-The Godfather'
});

//or for loading data in that format
$$('mylist').load("data.csv","mycsv");
```

Intermediate

Dynamic Loading

Dynamic loading means loading data on request, from the server. For example, a PHP script could supply data dynamically to a Webix component.

Dynamic loading is useful for long datasets since data populates the component in smaller portions. Initially data fills the view to some extent and later on it is dynamically loaded each time you scroll up and down the view or switch among pages.

For **linear structures** (used in the majority of Webix data components) you can control the process by specifying:

- the item position to start loading with;
- how many data items should be loaded each time the user scrolls/pages the element.

Read more at Dynamic Loading for Plain Structures.

For **hierarchical structures** like Tree and TreeTable you can specify:

- the node ID for which you need to load child data.

Read more at Dynamic Loading for Hierarchical Data Structures.

Note that dynamic loading works only if the functionality is **supported by the server**.

Here's an example of dynamic loading using ServerSide Connectors.

```
Server Script "data_dyn.php"-file

<?php
  require_once("../../common/config.php");
  $data = new JSONDataConnector($conn, $dbtype);
  $data->dynamic_loading(30);  // the number of records loaded initially
  $data->render_table("packages_plain", "id", "package, size,
architecture, section");
?>
```

Related Articles

- Dynamic Loading for Plain Structures
- Dynamic Loading for Hierarchical Data Structures

Dynamic Loading for Plain Structures

Data can be loaded to non-hierarchical components in portions:

- Firstly, a limited number of records is loaded;
- Then, data is loaded at the moment when:
  - user **scrolls** the component;
  - user switches to the new **page** within the component (in case pager is defined);
  - loadNext method is applied;

Currently dynamic loading for linear data is implemented for the following components: datatable and dataview.

Configuring Dynamic Loading Behavior

To enable dynamic data loading you need to:

1 . Set the component datasource that will load initial data. It can be done either with the help of url property or load method;

```
var dview = webix.ui({
    view:"dataview",
    url:"data/data_dyn.php"
});
//or
dview.load("data/data_dyn.php");
```

2 . Optionally, define various aspects of dynamic loading such as:

- datafetch - the number of records that should be loaded from server side during each dynamic loading request. **50** by default;
- datathrottle - time in milliseconds between subsequent dynamic loading calls, which helps to minimize the quantity of requests during quick paging or scrolling;
- loadahead - the number of records that should be loaded ahead of the "datafetch" parameter.

```
webix.ui({
    view:"dataview",
    datafetch:50,
    datathrottle: 500,
    loadahead:100
});
```

Related Samples

- RELATED SAMPLE: Setting the Number of Items to Load

- RELATED SAMPLE: Setting Polling Interval for Loading
- RELATED SAMPLE: Dynamic loading. Load Ahead

Serverside Request

When scolling or pagination occurs, the component will automatically issue a request based on the datasourse, e.g. **"data/data_dyn.php?continue=true&count=100&start=130"** where:

- **continue** - flag that indicates that this request was formed automatically;
- **count** - value of *datafetch* parameter;
- **start** - ID of the last data item in the data component before issuing a request.

Note that your server script should be able to handle these parameters.

Serverside Response

Serverside response should include the following information:

- **data** - the array of data records;
- **pos** - position in the data component to add the loaded data to (*required during each dynamic loading call*);
- **total_count** - the total number of records available on the server (*required only for initial loading call*);

```
Sample of JSON response

{
    "data":[
        {"id":1,"package":"acx100-source"},
        {"id":1,"package":"acx200-source"}
    ],
    "pos":,
    "total_count":999
}
```

Dynamic Loading by Api

The **loadNext** method can be used to send a request that loads the specified number of records and parses them into the component. The arguments are:

- **count** - the number of records that will be loaded on function execution;
- **start** - the start position to insert the loaded data to;

- **callback** - function to be executed after the response comes (or *null* if you don't need the one);
- **url** - the loading url (if it wasn't specified in the component constructor);
- **now** - boolean to ignore *datathrottle* and start loading data immediately on *loadNext()* execution.

```
Using url set in the component constructor

grida = webix.ui({
    view:"datatable",
    url:"data/data_dyn.php"
});
grida.loadNext(10, );
```

RELATED SAMPLE: Dynamic loading with removing the existing content

```
Using given url

gridb.loadNext(50,900,function(){
    this.showItem(903);
}, "data/data_dyn.php");
```

RELATED SAMPLE: Setting loading position

Use Cases

The method allows the following scenarios:

- new data may be added to the end of current component data:

```
grida.loadNext(30, pos, function(){
    this.showItemByIndex(pos);
    pos += 30;
});

//pos - index of the last data item
```

RELATED SAMPLE: Dynamic loading with appending content

- new data may substitute current component data (it should be cleared beforehand);

```
webix.ui({
    view:"datatable",
    "data->onParse":function(){
        this.clearAll();
        this.data.url = "data/data.php";
    }
});
grida.loadNext(10, base);

//base - index of the last data item
```

RELATED SAMPLE: Dynamic loading with removing the existing content

Dynamic Loading Events

You can capture the moment just before a dynamic loading request is sent to server with the help of an onDataRequest event.

It can be used to cancel the default behavior and issue a custom data request:

```
dtable.attachEvent("onDataRequest", function (start, count) {
  var data = custom_load_data(start, count);
  dtable.parse(data, "json");
  return false;
));
```

Related articles

- Dynamic Loading for Tree and Treetable

Dynamic Loading for Hierarchical Data Structures

This article discusses dynamic loading for tree, treetable and TreeCollection. Refer to the main Dynamic Loading article if needed.

Data can be loaded dynamically into hierarchical components as follows:

- First, the top parent nodes are loaded. Initially, they are collapsed (unless their open property is true).
- Then, data is loaded under each node when it is expanded.

```
var tree = webix.ui({
  view: "tree",
  url: "data/data_dyn.php"
});
// or
var tree = webix.ui({});
tree.load("data/data_dyn.php");
```

By default, Webix expects JSON data. The server can also return XML - see Loading Tree Data.

Dynamic loading can be implemented in three different ways:

- by setting the `webix_kids` property to `true` on nodes that are expected to have children
- by calling the loadBranch() method
- by setting a custom handler for the `onDataRequest` event

Using the webix_kids flag

On parent data items that can have children, you can set the **webix_kids** flag to `true`. When the user expands these nodes, Webix will automatically send XHR requests to the server for more data.

```
webix.ui({
  view: "tree",
  url: "data/data_dyn.php",
  data: [
    // parent node below
    { id: "1", value: "Branch", webix_kids: true }
  ]
});
```

Webix will automatically generate the request URL to dynamically load data, based on the `url` property

specified when initializing the component - **"data/data_dyn.php?continue=true&parent=1"**, where:

- **parent** - ID of the node that has just been opened;
- **continue** - flag indicating that it is an auto-generated request.

If `url` is specified, Webix will also make an initial request for that URL, without parameters. Make sure the server-side script can handle both situations: the initial loading of data (top-level parent nodes), and loading children under a given parent node.

While you can write custom scripts for this, you can check the ready-made ServerSide connectors.

Using the `loadBranch()` function

You can catch the **node expansion events** (onBeforeOpen or onAfterOpen) to load branch data using the **loadBranch** method. Let's assume that we have initial data loaded:

```
webix.ui({
  view: "tree",
  data: [
    { id:1, value: "Layout Branch", webix_kids: true },
    { id:2, value: "Data Branch", webix_kids: true },
  ],
  on: {
    onBeforeOpen: function (id) {
      // if children have not been loaded yet
      if (this.getItem(id).$count === -1)
        this.loadBranch(id, null, "data/data_dyn.php");  // null means no
callback
    }
  }
});
```

RELATED SAMPLE: Loading Branch Data

The URL of the new request will be based on the `url` property of the tree, which can be overridden by the loadBranch method - **"data/data__dyn.php?continue=true&parent=1"**. The parameters are the same as above:

- **parent** - ID of the node that has just been opened;
- **continue** - flag indicating that it is an auto-generated request.

If you want to use the same data source, omit the last parameter to `loadBranch()`:

```
var tree = webix.ui({
  view: "tree",
  url: "data/data_dyn.php",
  ...
});
```

```
this.loadBranch(id, null);  // "data/data_dyn.php" will be used
```

Be sure the branch data sent by the server is an object with **parent** and **data** properties. A JSON response should look like this:

```
{ parent: 1, data: [ ...array of child items... ] }
```

Redefining onDataRequestEvent

Set the **onDataRequest** event handler to redefine the default dynamic loading behaviour and use any desired method to get the data from the server:

The event fires when the user expands a node that has the **webix_kids** property set:

```
{ id: "1", value: "Layout Branch", webix_kids: true }
```

As it is stated above, this event triggers dynamic loading. In other words, if a component features a datasource (defined either by url property or load method) a request is issued using this url.

Still, the **onDataRequest** event makes it possible to redefine the logic in favour of custom one.

Let's assume that we have initial data loaded in some of the ways:

```
webix.ui({
  view: "tree",
  data: [
    { id: 1, value: "Layout Branch", webix_kids: true },
    { id: 2, value: "Data Branch", webix_kids: true },
     ],
  on: {
    onDataRequest: function (id) {
    webix.message("Getting children of " + id);
    this.parse(
      webix.ajax().get("data/data_dyn_json.php?parent=" +
id).then(function (data) {
        return data = data.json();
      })
    );
    // cancelling default behaviour
    return false;
    }
  }
});
```

RELATED SAMPLE: Modifying onDataRequest event

Here you completely redefine the default event handling and cancel default dynamic loading behaviour.

Be sure the branch data you get is returned with **parent** and **data** properties. For JSON the returned data looks like this:

```
{ parent:1, data:[ ..array of child items..]}
```

Dynamic Loading with ServerSide Connectors

In case you use Data Connector to load data from a database into a Tree or TreeTable dynamically, you should call **dynamic_loading()** on the server side:

```
Dynamic loading from a server database. Server-side code.

require_once("../connector-php/codebase/data_connector.php");  // includes
the connector file
$conn = mysql_connect('localhost', "root","");  // creates a connection
mysql_select_db("sampleDB");// selects a database

$data = new TreeDataConnector($conn, "MySQL");  // initializes the
connector object
$data->dynamic_loading(30);  // enables dynamic loading
// loads data from the specified database table
$data->render_table("packages_tree", "id", "value, state", "",
"parent_id");
```

Note, inside **dynamic_loading()** you should specify a number of records that will be loaded at once.

Everything else is the same as in case of usual loading from a database.

```
Dynamic loading from the server. Client-side code.

webix.ui({
  url: "data/data_dyn.php"
});
```

Note that each data item is supplied with the **webix_kids** flag to ensure dynamic loading.

RELATED SAMPLE: Dynamic Loading

Related Articles

- Dynamic Loading for Plain Structures

Advanced

Management of loading errors

Since data loading depends on multiple factors, server side frequently returns loading error messages instead of requested data. The webix UI library allows trapping such errors and treating them in some way on the client side.

Default Error Management

By default if an error occurs, there appears a **webix.message** with details being shown in console. This functionality works only in _debug code version while release version doesn't show any error messages.

To switch off the default error handler, use the following code:

```
webix.detachEvent(webix.debug_load_event);
```

Global Error Events

You can create your own global **loading error hander** that will catch serverside error responces regardless of the component:

onAjaxError(request_obj) - fires when any Ajax request results in an error; **onLoadError**(text, xml, xhttp, obj) - fires when an error has happened during loading the data into any component on the page.

Global events are attached to Webix object:

```
webix.attachEvent("onLoadError", function(text, xml, ajax, owner){
      //text - response text
      //xml - response xml object
      //ajax - xmlHttpRequest object
      //owner - component which triggered error

      ... custom code ...
});
```

Component Error Events

At the same time, each data component features its own **onLoadError** event with the same parameters:

```
$$('list').attachEvent("onLoadError", function(text, xml, ajax, owner){ ..
}
```

Furthermore, you can connect a custom error handler to a specific data requests.

```
$$('list').load("data.php",{
    success:function(text, xml, ajax){ ... },
    error:function(text, xml, ajax){ ... }
});
```

The same syntax is used for **webix.ajax** requests:

```
webix.ajax("data.php", {
    success:function(text, xml, ajax){ ... },
    error:function(text, xml, ajax){ ... }
});
```

Intermediate

Defining Data Scheme

Data scheme defines schemes for data processing, the default actions for loaded, updated, serialized as well as grouped and sorted.

Object scheme is defined as in the **scheme** property. Generally, it allows you to set a default scheme for data records of DataStore. So, in cases when you add an empty record to DataStore, the record will be populated with the values set by the scheme.

Any changes you apply through data scheme are directly applied to datastore items.

```
var list1 = webix.ui({
    view:"list",
    ...
    scheme:{
        name:"Unknown",
        age:16
    }
});

list1.add({}); //adds { name:"Unknown", age:16} record
```

Add() is the method of the DataStore component.So, we use the data property to access it.

In addition, there are 6 special keys you can use within the property:

- **$init** - runs when data is initially being loaded to the component + while reloading data for new DataStore items + when the add method is called;
- **$update** - runs for existing items while reloading data (for new items **$init** runs) + when the update method is called;
- **$save** - runs when saving to server is triggered in any of the ways;
- **$change** - runs each time data in the table is changed (i.e. loaded,added,updated,deleted). Note, the key runs when either the **$init** or the **$update** key runs;
- **$serialize** - runs when the serialize method is called;
- **$group** - the object with grouping parameters. Passed to the group function that automatically runs when data is initially being loaded to the component;
- **$sort** - the object with sorting parameters. Passed to the sort method that automatically runs when data is initially being loaded to the component.

Below you can find examples of using each of the keys.

$init key

Input parameters

The **$init** function accepts 1 parameter - the **data item object**.

And called:

- for each data item, while initial data loading;
- for newly-added items, while data reloading;
- for the specified item, when the add method is called.

Usage example

Let's assume, you want to populate a combo with data from some database field, e.g. from the field with name 'categories'. To make the combo recognize the field values as options, the combo should get data as 'text'/'value' pairs.

And here the $init key will come in useful:

```
webix.ui({
    view:"list",
    layout:"x",
    scheme:{
        $init:function(obj){
            if (obj.year>=2000) obj.$css = "newtime";
            if (obj.year<=1970) obj.$css = "oldtime";
        }
    },
    ...
});
```

RELATED SAMPLE: Horizontal List

$update key

Input parameters

The **$update** function accepts 1 parameter - the data item object.

And called:

- for each data item, while initial data loading;
- for updated items, while data reloading;
- for the specified item, when the update method is called.

Usage example

Let's assume, in your database table you have a field with the name '*update_date*' where store the date of the

last item update. To have the field values up to date, you can use the $update key and specify it as in:

```
webix.ui({
    view:"list",
    ...
    scheme:{
        $update:function(obj){
            var today = new Date();
            obj.update_date = today.getDate();// 'update_date' is the name
of data property
        }
    }
});
```

$save key

The **$save** function accepts 1 parameter - the data item object.

input parameters

And called:

- each time new or changed data is send to server.

Usage example

In combination with **$init** key, **$save** gives you an ability to set loading and saving pattern for serverside data.

For instance, you can work with dates as **DateTime objects** on client side while storing them as **strings** on server. To avoid formatting after each operation, do it only on data loading and saving.

```
webix.ui({
    view:"datatable",
    url:"data.php", //loading data
    save:"connector->data.php", //saving data
    scheme:{
        //turning strings into objects on loading
        $init:function(obj){
            if(obj.birthday)
                obj.birthday = webix.i18n.parseFormatDate(obj.birthday);
        },
        //turing objects back to strings on saving
        $save:function(obj){
            if(obj.birthday)
                obj.birthday = webix.i18n.parseFormatStr(obj.birthday)
        }
    },
    columns:[...]
```

```
});
```

$change key

Input parameters

The **$change** function accepts 1 parameter - the **data item object**.

And called:

- each time data is changed in the table.

Usage example

Let's assume, you have a datatable with a list of films and want to color rows with films that less than 1980 year in green. Here you can use the **$change** that will run each time data is changed and check whether the changed item(s) should be colored in green.

```
webix.ui({
    view:"datatable",
    ...
    scheme:{
        $change:function(item){
            if (item.year > 1980)
                item.$css = { "color":"green" };
        }
    }
})
```

RELATED SAMPLE: Rows styling

$serialize key

Input parameters

The **$serialize** function accepts 1 parameter - the data item object.

And called:

- for each data item, when the serialize method is called.

The $serialize key is the only key that must return some value while the $serialize function must return data item object that will go to the result serialized JSON object.

Usage example

Let's assume, in the database you store date values as Date() objects. But on the client you want to operate with such data as with strings. In this case, you can specify the **$serialize** key and call the serialize method to get data in the required format:

```
var mylist = webix.ui({
    view:"list",
    ...
    scheme:{
        $serialize: function(obj){
            return {name: obj.name, date:
dhx.i18n.dateFormatStr(obj.date)};
        }
    }
)};


var items = mylist.data.serialize();
//serialize() is the method of the DataStore component.
//That's we use the [data](data) property to access it.
```


Note, the method accepts a DataStore item and generally, changes made through the function change DataStore dataset.The example above shows how to get formatted data without changing the DataStore object.

$group key

Input parameters

The **$group** function accepts 1 parameter - so-called group object that has 2 properties:

- **by** – a data property according which items will be united in groups;
- **map** – an object that defines properties of the items in the group;

Properties are specified by an array. The first element of this array is a template with a property from original data, the second one – the functor that will be applied to all values of the property (set by the first element) in the group.Grouping provides the following functors: - **sum** – gets the sum of the values; - **max** – gets the maximum value in the group; - **min** - gets the minimum value in the group.

It’s possible to define custom functor. Read on the topic in the related article - Custom Functor for Data Grouping in Charts.

The **$group** function is called:

- for each data item, while initial data loading

The function doesn't run when you programatically call the group method.

Usage example

Let's take the following example: you've got the sales information over the last years in several companies and want to show total sales per companies in a chart. So, you need to get objects of the same companies and sum up their sales. Grouping helps to resolve this problem.

```
webix.ui({
    view:"chart",
    ...
    scheme:{
        $group:{
            by:"#company#", // 'company' is the name of data property
            map:{
                sales:["#sales#","sum"]
            }
        }
    }
});
```

RELATED SAMPLE: Chart: Grouping

$sort key

Input parameters

The **$sort** function accepts 1 parameter - so-called sorting object that has 3 properties:

- **by** - a template according which items will be sorted by,
- **dir** - the sorting direction: “asc” or “dsc”,
- **as** - the sorting method
  - '*int*'
  - '*string*'
  - '*string_strict*' (case-sensitive “string”)
  - or custom.

A **custom sorting method** is defined by a function. This function is called for each pair of adjacent values and returns 1,-1 or 0: - **1** - the object with the first value in pair must go before the second one - **-1** - the second object goes before the first - - the objects are equal

The $sort function is called:

- for each data item, while initial data loading.

The function doesn't run when you programatically call the sort method.

Usage example

Let's assume, you need to sort data by time property that consists of minutes and seconds divided by ”:”. Through such an example, we can show both use of the **$sort** key and how to define a custom sorting

method:

```
webix.ui({
    view:"chart",
    ...
    scheme:{
        $sort:{
            by:"#time#",
            dir:"asc",
            as: sortTime
        }
    }
});

function sortTime(a,b){
    a = a.split(":");
    b = b.split(":");
    if(a[] > b[]) return 1;
    else if(a[] < b[]) return -1;
    else{
        if(a[1] > b[1]) return 1;
        else if(a[1] < b[1]) return -1;
        return ;
    }
}
```

Intermediate

Server-side Integration

Serverside integration is possible with Webix data components, client-side datastores (called Data Collections) and forms. It is enabled in several ways:

- you can write your own **custom scripts** in you favourite language to load data and save changes back;
- you can use server side rest api
- you can make use of a **Server Side Connector** (ready to use solution for data loading and saving). Connectors are available in PHP, Java, .Net, ColdFusion versions;
- You can use MVC.Net, Ruby on Rails, PHP Yii **MVC frameworks** coupled with your scripts or connectors.

Data Loading

Basic client-side code that enables loading data to a component during its initialization is as follows:

```
webix.ui({
    id:"dtable",
    view:"datatable",
    url:"data.php"
});
```

Or, to load data after component init, apply **load()** method to it:

```
$$("dtable").load("myscript.php", "xml");
```

Note that

- Server script you state as **url** parameter or pass into **load()** function executes GET request and returns data from server in either of the possible data formats;
- You should specify **datatype** parameter if incoming data is other than JSON (default);
- In case of long datasets dynamic loading functionality will be helpful;
- Default loading pattern can be customized - specific loading modes, ajax helper for passing params into load script.

More info about custom scripts is in the dedicated article.

If you are unsure what to do in case you need to get data from a **different domain** with relations to your app's one, consult the our solution to cross-domain data loading.

Data Saving

Basic client-side code that enables data saving from a component to database is as follows:

```
webix.ui({
    id:"dtable",
    view:"datatable",
    save:"save_script.php"
});
```

Note that

- Save script gets data in POST request and saves them to database.
- Save script typically returns **operation status** and **ID** of the changed item. To tune server-side response, look into the dedicated article.
- Saving can be implemented via Webix Ajax helper, which adds more flexibility.

In addition, **save** property automatically initializes DataProcessor for this component so that data changes are transmitted to your save script. DataProcessor listens to component events, defines editing operation (as **insert, update** and **delete** one) and sends changed data to server script in POST request.

Defining Operation Type

By default, operation status comes as **webix_operation** in POST request. Usually, save script is single and contains patterns for different operations.

At the same time you can define a **specific script** for each operation. Dataprocessor will define the type of operation and the necessary script will be executed:

```
save:{
    "insert":"data/insert.php",
    "update":"data/update.php",
    "delete":"data/delete.php"
}
```

RELATED SAMPLE: Server-side Integration: Custom URL

If you need to change the default pattern of data processing - look up Dataprocessor docs;

Saving Uploaded files

If you use Webix Uploader, you may wonder about how saving of the files is performed.

```
{
    view:"uploader",
    id:""uploader1,
    name:"uploader"
```

```
    upload:"upload.php"
    autosend:false
}

$$("uploader1").send();
```

Note that

- **Autosend** should be set to *false* in order that saving starts only when you call the **send** method. When you call it, the script specified in **upload** property will be executed;
- **Server part of file upload** is described in detail in the dedicated part of uploader documentation;
- The information about how to **save form and uploader data** separately - is nearby.

Saving Form and Input Data

Form data is either

- pushed to the component it is bound with;
- or (more typically) is sent to server by Webix Ajax request

Further Reading

- Customizing Loading Pattern. Ajax Loading and Parsing
- Saving Data with Webix Ajax Helper
- Loading and Saving with Custom Scripts
- Loading and Saving with Server Side Connectors
- Customizing DataProcessor
- Offline Support
- Working in REST Mode
- Working with Proxy Objects

Advanced

Custom Server Scripts

In case of custom server scripts you write all the logic of database connection, execute queries that get data from the necessary tables as well as queries for data inserting, updating and deleting.

Basic client-side loading and saving pattern remains unchanged:

```
webix.ui({
    view:"list",
    url:"load.php",
    save:"save.php"
})
```

Url loads data returned by dedicated script while **save** points to script that saves data to database each time you make changes on client side.

The following basic instructions will help you tune your scripts to return data and provide responses for editing operations. Here we work on PHP with SQLite3 database.

- Data Loading
- Data Saving
- Tuning serverside response

If you are unsure what to do in case you need to get data from a **different domain** with relations to your app's one, consult the our solution to cross-domain data loading.

Data Loading

Webix components support incoming data in JSON, JSArray, XML, HTMLtable and CSV formats, so ensure that your script returns data properly formatted.

For JSON, encode the resulting data array with the dedicated function(**json_encode()** in PHP)

```
//connect to database
$db = new SQLite3('../../../common/testdata.sqlite');

//select data
$res = $db->query("SELECT * FROM films");

//convert data to json
$data = array();
while ($rec = $res->fetchArray(SQLITE3_ASSOC)) //each row
    $data[] = $rec;
//output json
```

```
echo json_encode($data);
```

Datatree loading

Datatree can be populated with inline data grouped on client side or with **hierarchical data** grouped by server-side script.

For hierarchical data you need to have:

- **relation_id** field in database table that for each record will store the ID of a record that will be **parent** for this one. For **root** record this parameter will be ;
- (optionally) - **open** field for each record indicating whether this record will be initially opened (1) or not (0) - if it has children, of course.

Dataset is compiled with a function that contains SQL query with WHERE clause, which gets data from root record (parent = 0) down to all its children. JSON data for datatree should store chilren for an item in its **data** property:

```
function get($id = ) {
    $db = new SQLite3('../../../common/testdata.sqlite');
    //connect to database

    //select data
    $res = $db->query("SELECT * FROM films_tree WHERE parent={$id}");

    //convert data to json
    $data = array();
    while ($rec = $res->fetchArray(SQLITE3_ASSOC)) {
        $rec["data"] = get($rec["id"]); //putting chilrden into item
"data" property
        $data[] = $rec;
    }
    //output json
    return $data;
}

//formatting output
echo json_encode(get());
```

Form Loading

On client side, **load()** method is applied to an already inited form. The method contains loading script that sometimes requires record ID as part of GET request.

```
$$("form1").load("server/form.php?id=3");
```

Then on server you'll have the ID to use in WHERE clause of SELECT query.

```
$id = $_GET["id"];

$res = $db->query("SELECT * FROM films WHERE id={$id}");
```

Data Saving

In essence defining **save** initializes DataProcessor that gets data ready for sending to server and triggers script executing each time changes are made. Defaut DataProcessor logic can be customized.

Scripts gets changed data in body of POST request. For each inserted, updated and deleted record, all properties of its object are sent together with **webix_operation**

```
id  8
rank    99
rating
title   New
votes   100
webix_operation delete
year    2012
```

Additional item properies can be set as well by updating this item on client side:

```
var item = this.getItem(id);
item.old_rank = some; //setting new property and value for it
this.updateItem(id);
```

The new property *old_rank* will be passed in POST like native item properties.

RELATED SAMPLE: Datatable: Reordering

You can easily get to these parameters to use them for making corresponding database quieries.

```
$operation = $POST['webix_operation'];

if($operation=="insert")
    $db->query("INSERT SQL query");
```

Don't forget to escape user data to avoid SQL injections:

```
$title = $db->escapeString($_POST["title"]);
```

And convert data to JSON according to common pattern above.

Datatree Saving

With datatree use the same pattern as above and don't forget to update **parent** field with **parent** item value in INSERT and UPDATE queries.

It goes without saying that you should render this field during loading (SELECT * guarantees this for sure) and update it on client side while working with tree data.

```
$parent = $_POST["parent"];

//queries
```

RELATED SAMPLE: Tree: Saving

Form Saving

Form is typically saved on some page event (e.g. button clicking) so there's no need in initing DataProcessor for it. As a result, POST request will contain only form data and **no webix_operation** type.

You decide what should be done:

- **UPDATE** data on server side for a record that was previously loaded to this form;
- **INSERT** data into necessary database table (i.e. create a new record);
- **update data on client-side** - in this case form should be bound with a component or Data Collection so that form gets data from them and send changed to them while their **save** scripts handle these changes.

Typical Ajax request to save form data to server look like this:

```
webix.ajax().post("server/form_save.php", $$(""myform).getValues(),
function(response) {
        webix.message(response.status);
});
```

Form getValues() returns data from all fields as JSON object - just what is required for params sent via Ajax.

For tuning responses scroll down a little bit. For learning Webix API for Ajax operations - go there.

Tuning ServerSide Response

For each query you should tune response, which can be either **plain text** or **JSON object** with necesary params:

```
//plain text on failure
echo "Data saving was not performed";

//JSON with item ID and status for failure
echo '{ "id":"'.$id.'", "status":"error" }';

//JSON with item ID and status for success
echo '{ "id":"'.$id.'", "status":"success" }';
```

```
//JSON for successful inserting operation
echo '{ "id":"'.$id.'", "status":"success",
"newid":"'.$db->lastInsertRowID().'" }';

$res = $db->query(SQL query);

if($res)
    //response for success
else
    //response for failure
```

For INSERT operations, new ID is important as on client side any newly added item gets **temporary ID** randomly generated by **webix.uid()** method (something like 1369905287157).

After adding this item to database table it gets normal ID generated by auto increment, and this ID will be loaded to client side on page refresh. So, you might require this new ID at once to work with recently added item.

If you use **Ajax** request, these responses are easily derived in **callback**. If response is a JSON object, any of its properties can be treated separately.

```
webix.ajax().post("server/save.php", {}, function(response) {
        webix.message(response.status);
});
```

More about Ajax operations in Webix.

Intermediate

REST Mode Support

Webix features full support for REST load and save protocol.

To enter REST mode, use prefix **rest** before load and save scripts. This will enable the necessary proxy object.

```
{
    view:"datatable",
    ..config..
    save: "rest->server/datatable_rest.php",
    url: "rest->server/datatable.php"
}
```

RELATED SAMPLE: Datatable: REST

You can as well init **REST proxy object** and set is as value of url and save properties. For these needs **webix.proxy()** function receives *rest* prefix and path to your serverside *save script*.

```
var rest = webix.proxy("rest", "server/datatable_rest.php");
```

REST Load and Save Pattern

- On loading GET request is executed and data in any of supported data formats is returned;
- When editing happens or update() method is called - PUT request is generated, item ID and other params are sent to the script. Item ID is shown in the address line;
- When item is added inside the component - POST request is generated, item ID and other params are sent to the script. Item ID is shown in the address line;
- When item is removed from th component - DELETE request is generated, item ID is sent to the script.

Save script treats adding, editing and deleting operations separately and should contain different code on the base of request method.

Serverside Response

Serverside response is set like with any custom script.

Note that Webix DataProcessor can be tuned to update client-side data from response automatically:

```
view:"datatable",
save: {
    url:"rest->server/datatable_rest.php",
    autoupdate:true
}
```

Intermediate

Saving Data with Webix Ajax Helper

Webix ajax helpers helps save component data on demand as well as form and input data.

Note that DataProcessor isn't inited this time and you need to catch the necessary events (add, edit, delete) yourself. For forms saving request is usually send on button click.

Data is sent to server script in the body of Ajax POST request:

```
webix.ajax().post("myscript", params, callback);
```

Note that

- **"myscript.php"** can be your own script on any language you choose. It can be as well a connector-based script);
- **params** are sent in form of JSON object (which is exactly what the form's getValues method returns:

```
webix.ajax().post("my_script", {id: some, name: some, address:some },
callback);
webix.ajax().post("my_script", form.getValues(), callback);
```

If combined with connector-based script, simplified data transfer protocol can be used.

Additionally, you can pass **extra parameters** as GET request:

```
webix.ajax().post("data/load.php?id=1&action=update", {prop1:value1,
prop2:value2} function(text, data){
    ...//callback
});
```

There you'll have in serverside script:

```
$id = $_GET['id'];
$action = $_GET['action'];
$prop1 = $_POST['prop1];
$prop2 = $_POST['prop2];
```

Learn more about the possibilities of Webix Ajax interface.

Intermediate

Customizing Loading Pattern. Ajax Loading and Parsing

By default Webix loads data to its component by an asynchronous Ajax GET request. However, various modifications are possible.

Loading Data via POST Request

By default data is loaded via GET request. To change it to POST, use **post** prefix.

```
{
    view:"datatable",
    ..config..
    url: "post->load.php"
}
```

Loading Data via Synchronous Ajax Request

You can change the request to a synchronous one (sometimes it may be useful).

```
{
    view:"datatable",
    ..config..
    url: "sync->load.php"
}
```

Modifying Background Ajax Requests

There exists no possibility to modify background Ajax requests sent by the above pattern.

However, you can catch Webix **onBeforeAjax** event to modify ANY Ajax request on the page (so be attentive):

```
Sending Headers

webix.attachEvent("onBeforeAjax", function(mode, url, data, request,
headers){
  headers["Content-type"]= "application/json";
})
```

Note that Webix Ajax module (described below) features a **built-in functionality** for sending **headers** with

serverside requests.

Loading with Webix Ajax Helper

To load data on demand without breaking application flow, you can resort to Webix Ajax interface.

```
//get request (default)
webix.ajax("data/load.php",function (text,data){
    $$("list").parse(text);
});
```

Serverside script should return **inline data** in any of supported data types so that it can be parsed into the necessary component.

Above snippet will set the new data for list. To add new data to the existing list data, define start position for parsing:

```
webix.ajax("data/load.php",function (text,data){
    $$("list").parse({
        pos: $$("list").count(), //start position
        data:text
    });
});
```

For setting the necessary **serverside response** manually, see the dedicated article. As to Server Side Connector, they return data in JSON or XML format depending on connector type.

Ajax loading allows **passing params to serverside script** as GET request:

```
webix.ajax("server/load.php?id="+id, function(text, data){
    ...//callback
});
```

Learn more about Webix Ajax Operations.

Adding Extra Functionality to Loading Functions

Here you should consult:

- Ready-to use proxy objects for offline support;
- Proxy object for working in REST mode;
- Guidlines about proxy objects and custom proxy objects in particular.

Intermediate

Offline Support

Offline support allows storing serverside data locally in your browser cache which helps you go on working with data components after refresh when connection to server fails.

Saving the whole page via Cache Manifest

Files subject to caching, should be listed in a cache manifest, a typically HTML5 feature, which allows for differentiating between files available in an offline mode from those available when you are connected.

Firstly, the manifest is included into each webpage you'd like to be cached.

```
<!DOCTYPE html>
<html manifest="app.appcache">
```

Secondly, manifest details are specified in the *app.appcache* file. As a rule, the following files should be cached:

```
offline_app.html //each application file itself,

../../../codebase/webix.js //library files
../../../codebase/webix.css

data/data.json //data-loading file
```

Offline Support for Webix Component Data

Offline support is implemented via three load and save modes:

- **offline** and **cache** - both working with serverside data that is additionally stored locally;
- **local** - works with local storage data only.

Offline Mode

Offline mode ensures that serverside data will be stored locally in browser and later on, the component will always try to reach server to get or save data BUT in case of failure, the data will be loaded from cache and stored in cache, accordingly. When you are connected again, all changed data should be derived from cache and sent to server (otherwise on next page refresh it will be replaced by outdated serverside data).

Webix works with cache through its **webix.storage.local** interface. See the description of caching logic.

To enable this, use **offline** prefix before load and save scripts:

```
{
    view:"datatable",
    id:"datatable1",
    ..config..
    save: "offline->server/datatable.php",
    url: "offline->server/datatable.php"
}
```

RELATED SAMPLE: Datatable: Offline mode

Or, if you load data after component init, pass this prefix (*offline or cache*) to load fucntion together with load script itself:

```
//two methods are equal
$$("datatable1").load(offline->server/datatable_load.php);
$$("datatable1").load(webix.proxy("offline",
"server/datatable_load.php"));

//or, if you want to load data in POST request:
$$("datatable1").load( webix.proxy("offline", "post->url.php"));
```

In essense, the prefix signifies integration with a dedicated **proxy object** that contains all the logic of offline storage. Proxy objects are described separately.

Cache Mode

When you enter cache mode, data is first loaded to component and put to browser local storage and then you work ONLY with this cached data: it's loaded from cache and stored to it without trying to connect to server.

Webix works with cache through its **webix.storage.local** interface. See the description of caching logic.

To enable this, use **cache** prefix before load and save scripts:

```
{
    view:"datatable",
    id:"datatable1",
    ..config..
    save: "cache->server/datatable.php",
    url: "cache->server/datatable.php"
}
```

RELATED SAMPLE: Datatable: Cache mode

Or, if you load data after component init, pass this prefix (*offline or cache*) to load fucntion together with load script itself:

```
//two methods are equal
$$("datatable1").load(cache->server/datatable_load.php);
$$("datatable1").load(webix.proxy("cache", "server/datatable_load.php"));
```

```
//or, if you want to load data in POST request:
$$("datatable1").load( webix.proxy("cache", "post->url.php"));
```

In essense, the prefix signifies integration with a dedicated **proxy object** that contains all the logic of offline storage. Proxy objects are described separately.

Local Mode

When you work in local mode, your component deals with local storage objects only. You can load data from local storage and save it there without any serverside part.

Webix works with cache through its **webix.storage.local** interface. See the description of caching logic.

In local mode, you should provide names of **localStorage objects** instead of urls:

```
{
    view:"datatable",
    id:"datatable1",
    ..config..
    save: "cache->mydata",
    url: "cache->mydata"
}
```

Neither offline nor cache modes allow accessing localStorage objects, but local mode makes it possible.

Working with Cached Data

Both of these proxy objects, **offline** and **cache**, offer API for working with currently cached data. Methods are applied to the component's **url** property object:

```
$$("datatable1").config.url.someMethod();
```

- **clearCache()** - to remove cached data of this component;
- **setCache**(data) - pushes any data to cache. Data should be passed as JSON object or JSON string;
- **saveAll()** - serializes current component data and pushes it to cache;
- **getCache** - gets the latest copy of cached data for this component in the form of JSON object. It can be used to populate the component with the help of parse method.

Setting Static Cache

In addition, you can manually set data for component cache. Here you manipulate Webix proxy objects.

Firstly, you should create **cache proxy object**. It includes server script for data loading:

```
var static_proxy = webix.proxy("cache", "server/datatable.php");
```

This object should be used for data loading(**url**) and saving(**save**):

```
var grid = {
    view:"datatable",
    ..config..
    save: static_proxy,
    url: static_proxy
};
```

Then, you set necessary data to this object via the **setCache()** method:

```
if (!static_proxy.getCache())
    static_proxy.setCache([
        {"id":1,"title":"The Shaushenk Redemption ",
            "year":"1998","votes":194865,"rating":"7.5","rank":1},
        {"id":2,"title":"The Godfather",
            "year":"1974","votes":511495,"rating":"9.2","rank":2}
    ]);
```

RELATED SAMPLE: Datatable: Cache static mode

Proxy objects are described in detail in the corresponding article.

Advanced

Webix Proxy Objects

Either the component's url property or its load function can be used not only for passing loading script, but can include additional loading logic in **proxy objects**.

Built-in Proxy Objects

There's a list of ready-to-use Webix proxy objects that can be used by adding prefixes to a loading script.

- **offline** and **cache** - for application offline support of applications that use serverside data;
- **local** - for saving component data into browser local storage and working with it. Described as part of offline support article;
- **connector** - for saving data via Server Side Connector;
- **rest** - for working with server in REST mode;
- **post** - for loading data in POST request (GET by default);
- **sync** - for loading data via a synchronous Ajax request (by default, loading is performed by an asynchronous request);
- **indexdb** - for working with indexedDB in-browser database storage;
- **faye** - for enabling live data update on all clients currently using the application. Look here for details.

Prefixes are used before the path to you serverside script:

```
var list = {
    view:"list"
    url:"offline->load.php",
    save:"offline->load.php"
};

grid.load("rest->load.php")
```

Using methods of Built-in Proxy Objects

Methods of a proxy object are called from the **url** of the component it's used for:

```
//returns cached data for this list
list.config.url.getCache();
```

You can also save an instance of the necessary proxy object in a **variable** and call all methods from it:

```
var static_proxy = webix.proxy("cache", "server/datatable.php");

var grid = {
    view:"datatable",
```

```
    ..config..
    save: static_proxy,
    url: static_proxy
};

if (!static_proxy.getCache())
    static_proxy.setCache([
        {"id":1,"title":"The Shaushenk Redemption ",
            "year":"1998","votes":194865,"rating":"7.5","rank":1},
        {"id":2,"title":"The Godfather",
            "year":"1974","votes":511495,"rating":"9.2","rank":2}
    ]);
```

RELATED SAMPLE: Datatable: Cache static mode

Customizing Built-in Proxy Objects

You can add extra functionality to built-in proxy objects or redefine existing properties or methods by extending their functionality.

For instance, you can change the default place for storing cached data of an **offline** proxy object, which is webix.storage.local by default:

```
webix.proxy.myCustomName = {
    init:function(){
        webix.extend(this, webix.proxy.offline);
    },
    cache: some_place_you_choose
};
```

And then use it as prefix to load and save scripts:

```
{
    view:"datatable",
    url:"myCustomName->load.php",
    save:"myCustomName->load.php"
}
//or, after component init
$$("datatable1").load("myCustomName->load.php");
```

Creating Custom Proxy Objects

Proxy object is a plain JSON object you create to add extra functionality to your loading and saving pattern.

It features

- compulsory **$proxy** property set to *true*;

- other properties;
- functions to fire on loading and saving;
- other methods.

Proxy objects

```
webix.proxy.myCustomName = {
    $proxy:true
    prop1:value1,
    ...
    load:function(view, callback){
        //you loading pattern
        webix.ajax(this.source, callback, view);
    },
    save:function(view, update, dp, callback){
        //your saving pattern
        webix.ajax().post(url, data, callback);
    }
};
```

And then use it as prefix to load and save scripts:

```
{
    view:"datatable",
    url:"myCustomName->load.php",
    save:"myCustomName->load.php"
}
//or, after component init
$$("datatable1").load("myCustomName->load.php");
```

Intermediate

Working with IndexedDB

Webix offers a simple way for efficient work with IndexedDB in-browser storage.

Complying with Webix concept of serverside proxy objects IndexedDB support is implemented via **indexdb** proxy object that contains necessary logic for creating databases as well as working with existing ones.

Webix **indexdb** proxy object is used for:

- creating IndexedDB database in current browser and populating it with data;
- getting IndexedDB data and loading it into Webix **data components** (datatable, dataview, list, chart);
- saving changes you perform over component data into the corresponding database. Each time you refresh the page up-to-data data is loaded.

Indexdb proxy object features three built-in **methods**:

- **load** - loads database data from the chosen database into the component;
- **save** - saves component changes into the chosen database;
- **create** - creates an IndexedDB database with objectstore(s) and loads data to it(them).

Loading data from IndexedDB Store to Webix Component

To load data from an existing IndexedDB database, provide its name together with an objectstore name according to the following pattern: **indexdb->db_name/store_name**.

Both database and objectstore names are **must-have** paramaters:

```
{
    view:"datatable",
    id:"datatable",
    url:"indexdb->mydb/mycollection"
}

//or, for loading data after component init
$$("datatable").load("indexdb->mydb/mycollection");
```

RELATED SAMPLE: Connector DataTable loading

Component's url property as well as load method define datasource. In case of IndexedDB, they call **load** method of an **indexdb** proxy object and pass database and objectstore names into it.

Saving Component Data to IndexedDB Database

To save changes performed over component data, you need to pass database and objectstore names into the component's **save** property according to the following pattern: **indexdb->db_name/store_name**:

```
{
    view:"datatable",
    url:"indexdb->mydb/mycollection",
    save:"indexdb->mydb/mycollection"
}
```

RELATED SAMPLE: Connector DataTable saving

Component's save property calls **save** method of an **indexdb** proxy object.

Creating IndexedDB Datatase and Populating it with Data

The **indexdb** proxy allows using its methods directly. For instance, you can call **create** method with the following code:

```
webix.proxy.indexdb.create(dbname, collection_data, dbversion, callback);
```

Where:

- **dbname** - name of a databases being created;
- **collection_data** - JSON object that contains **collection names** as keys and **collection data** as values;
- **dbversion** - version of the database being created. It you prefer not to bother about versons, pass *null*;
- **callback** - callback function.

Here's an example of a function call, where *mydb* database with *mycollection* objectstore is created and data is loaded to the objectstore.

```
webix.proxy.indexdb.create("mydb", {
    mycollection:[
        {"id":1,"title":"The Shaushenk Redemption
","year":"1998","votes":194865,
            "rating":"7.5","rank":1},
        {"id":2,"title":"The
Godfather","year":"1975","votes":511495,"rating":"9.2",
            "rank":2},
        {"id":3,"title":"The Godfather: Part
II","year":"1974","votes":319352,
            "rating":"9.0","rank":3},
        {"id":4,"title":"The Good, the Bad and the
Ugly","year":"1966","votes":213030,
            "rating":"8.9","rank":4},
        {"id":5,"title":"My Fair
Lady","year":"1994","votes":533848,"rating":"9.1",
            "rank":5},
        {"id":6,"title":"12 Angry
Men","year":"1957","votes":164558,"rating":"8.9",
```

```
            "rank":6}
    ],
    //mycollection2:[data], //as many stores as you like
    //mycollection2:[data]
}, null, init_demo);
```

RELATED SAMPLE: Connector DataTable loading

Webix **data loading** pattern in described separately.

Websockets (Faye)

Faye proxy object is used to ensure **live data update** on all the clients currently using the application with this feature. It is based on Faye publish-subscribe messaging system which is in turn based on the Bayeux protocol.

Faye proxy object is useful for creating corporate chats clients. It can be used with any Webix **data component** while list is the most suitable one a chat.

Proxy object is defined within the component constructor with the help of url and save properties:

```
{
    view: "list",
    id:"chat",
    url: "faye->/data",
    save: "faye->/data"
}
//or, to load data after component init
$$("chat").load("faye->/data");
```

- **Url** property, as well as load method is used to define datasource for the component. Here it includes **proxy name** (faye) and **storage name** on your faye server (any);
- **Save** property enables data saving to the chosen storage. On page refresh up-to-date data is loaded into the component.

Storage name must contain **slash** at the beginning.

Starting Faye server

Besides, you need to provide standard logic to staring **Faye server**:

```
var http = require('http'),
faye = require('faye');

var server = http.createServer(),
bayeux = new faye.NodeAdapter({mount: '/'});

bayeux.attach(server);
server.listen(8000);
```

Initializing Faye client

And define **Faye client** and its **ID** that are required for **faye** proxy:

```
webix.proxy.faye.client = new Faye.Client('//localhost:8000/');
webix.proxy.faye.clientId = webix.uid(); //or any custom pattern
```

Related Sample

To play with the feature, load and unzip the demo. Don't forget about NodeJS server running.

Intermediate

DataProcessor

Webix DataProcessor is a functional library that lets you to 'communicate' with server-side backend. It

- resides on the **client side** as a mixin of Webix library;
- listens to **component events** - data adding, updating, moving (should be enabled separately) and removing - and passes **changed data** as well as **operation** performed (insert, update, delete) to the server script in **POST request**.
- handles **data validation** before passing to server script;
- can be used for any UI component and DataCollection.

Without DataProcessor, you need to attach corresponding functions to component events (onAfterInsert/Update/Delete) to get data ready for processing.

DataProcessor Initialization

DataProcessor object can be inited in both long and short forms. Compulsory parameters include:

- **url**, a path to your own server script or connector-based one (described in Server Side Connector). In any case, it should be the script that enables data loading from the database into the component;
- **master**, a component or DataCollection you will use the dataProcessor with.

Other parameters (**mode**, validation **rules**, **on**, etc.) and optional and can be found here.

Implicit Initialization via Save Property

When you define **url** (path to the necessary script) as value of **save** property for the needed component (**master**) - DataProcessor is automatically inited:

```
webix.ui({
    view:"datatable",
    //..config..
    url: "data.php", //script that links to DataConnector and loads data
    save: "connector -> data.php" //the same script used for data
processing

    //or
    url:"data_load.php", //your custom script for loading
    save:"data_save.php" //you custom script for saving
});
```

Short Form

A short form connects the previously created dataProcessor to the component, but if there isn't any, it creates the new dataProcessor.

```
Short Form (master, url)

webix.dp("mylist", "some_script.php");
```

Long Form

With a long form you make use of the **new** constructor where you create an object through object literals.

```
Full Form

dp = new webix.DataProcessor({
    url: "data.php",
    master: $$('mylist'),
    //optional properties
});
```

RELATED SAMPLE: Server-side Integration: List

A **long initialization form** allows for setting an **ID** for the dataprocessor. Otherwise, it is given an auto-generated one.

Getting DataProcessor Object

Dataprocessor features a set of methods and properties that can be used to change default processing pattern. To make use of them, you need to get to a dataprocessor object, which can be achieved with the help of Webix dp method:

- getting a dataprocessor initialized for a component with the specified ID:

```
var dp = webix.dp("mylist"); //returns dataprocessor object for "list"
view
```

- getting a dataprocessor with the specified ID

```
var dp = webix.dp.$$("mydp");
```

Data Processing Operations

DataProcessor interpretes client-side operations performed on each data item and defines the type of data action for it:

The default processing looks as follows:

- If **remove()** function is called within the component, action type is **delete**;
- If **editing** is done, or **update()** function is called, data action is of an **update** type;
- If you **add()** data to the component, the data action is **insert**.

This **data action** together with **data** of edited/added/deleted record is sent to server script the moment any editing operation is performed. Data action is passed as:

- **!nativeeditor_status** parameter in case of implicit DataProcessor initing (with **save** property) and simultaneous usage of connector;

```
view:"list",
save:"connector->myscript.php" //link to Server Side Connector
```

- **webix_operation** parameter in all other cases.

It looks like this:

```
id  7
title   The Shawshank Redemption
webix_operation delete
```

And other protocol with implicit Dataprocessor initing for connector-based scripts:

```
1_!nativeeditor_status  update
1_birthday  1965-08-19
1_id    1
```

The moment Dataprocessor returns data, script execution begins (if other is not stated):

- If serverside integration is enabled with a Server Side Connector, the connector automatically **generates database request** corresponding to action type to treat changed data;
- For custom scripts, you get **webix_operation** and other data via **POST** request and write corresponding queries for each type of operation.

DataProcessor for Drag-n-Drop and Data Moving

DnD operations and data item moving can be tracked by DataProcessor like any other CRUD operation provided that you switch on DataProcessor trackMove functionality.

```
new webix.DataProcessor({
    master: tree,
    url: "...",
    trackMove:true
});
```

RELATED SAMPLE: Server-side Integration: Tree

DataProcessor EventSystem and Error Tracking

DataProcessor events (listed in API reference) can be used for different purposes. For instance, the events can help you:

1) **Modify data** before it's gone to server with the **onBeforeDataSend** event that takes the whole dataobject as parameter:

```
dp.attachEvent('onBeforeDataSend', function(obj){
    obj.data.StartDate = webix.i18n.dateFormatStr(obj.data.StartDate);
});
```

2) **Track successfull** and **unsuccessful serverside responses** separately with the help of **onAfterSync** and **onAfterSaveError** events respectively:

```
Successful server response

dp.attachEvent('onAfterSync, function(id, text, data){
    var response = data.xml(),
        hash = response.data;
    //hash {type:'', tid: '', sid:''}
});
```

- **type** - type of **action** performed (*insert, update, delete*);
- **sid** - item ID that was sent to server for update;
- **tid** - item ID that returned from server after update.

tid and **sid** will only be different in case on **insert** operation. In this case, DataProcessor automatically changes client-side ID to the ID that was generated for the item on server.

```
Unsuccessful server response

dp.attachEvent('onAfterSaveError', function(id, status, obj){
     var operation = this.getItemState(id).operation; //operation that was
performed
});
```

In case of an error during saving, response **status** will always be 'error' while **type** of operation is derived by getItemState method.

Getting to a master component from DataProcessor

Inside DataProcessor event handlers you can reach the master component througn DataProcessor

configuration objetc:

```
dp.attachEvent('onSomeEvent', function(id, status, obj){
   var grid = this.config.master; //this == DataProcessor
});
```

Data Updating from Server Response

As it was stated earlier, the **new ID** from response of **insert** request will automatically replace the temporary client-side ID. This is part of default DataProcessor behavior, and no specific actions are required.

Additionally, yuo can enable **automatic data update** for all fields taking part in **insert** and **update** operations. It requires the following additions to the code:

- tune serverside response to return the whole data object (not only status or IDs);
- enable DataProcessor **autoupdate** functionality:

```
Either during explicit DataProcessor definition

new webix.DataProcessor({
    updateFromResponse:true,
    master:"datatable1",
    url:"..."
});
```

```
Or when defining DataProcessor implicitely

view:"datatable",
save:{
    url:"...",
    updateFromResponse:true
}
```

RELATED SAMPLE: Datatable: Saving

It can be uselful for REST-full applications or when you need to fill in client-side fields which values can be calculated only on server side, etc.

Changing Default Processing Logic

The event system for dataProcessor helps change the default processing logic right on client-side.

To alter processing, you should manually attach processing function to the necessary DataProcessor event (*onBeforeDelete, onBeforeInsert, onBeforeUpdate*). For instance, set **update** action type on **onBeforeDelete** thus making connector issue UPDATE request instead of DELETE.

This is how we change deletion from database by updating this record:

```
webix.ui({
    view:"datatable",
    id:"grid",
    ..// config options
    url:"data/employee.php",
    save:"connector->data/employee.php"
 });

..//function redefining removal into updating

webix.dp($$("grid")).attachEvent("onbeforedelete", function(id, action){
        action.operation = "update";
        action.data.deleted = webix.i18n.parseFormatStr(new Date());
});
```

where..

- **id** - the ID of the item being processed;
- **action** - DataAction object;
- **operation** - the database operation being performed (action type);
- **data** - refers to component inner DataStore. Here we can get to the desired data item property (*action.data.property_name*).

Now, when a record is removed from the component, a field **deleted** of this record is updated in the database table. The current date is set into it to track the moment when the record was removed on the client side.

Learn more about the possibilities of data manipulation on clent side in related articles:

- Data Adding and Deletion;
- Data Updating
- Data Editing

Cancelling and Enabling Dataprocessor for Some Operation

Cancelling DataProcessor

Not any client-side update is to be saved to server. To temporarily cancel DataProcessor you can:

- either apply ignore fucntion to the Dataprocessor object.

```
webix.dp("grid").ignore(function(){
    $$("grid).add(data);
});
```

- switch Dataprocessor off during the update operation:

```
dp.on();
$$("grid).add(data);
dp.off();
```

- cancel automatic data sending via DataProcessor autoupdate property.

```
//initially during dp configuration
new webix.DataProcessor({
    updateFromResponse:true,
    master:"datatable1",
    url:"...",
    autoupdate:false
});

//dynamically
webix.dp($$("datatable1")).define("autoupdate", false);
```

Enabling Dataprocessor

The following concerns only disabled DataProcessor or the one with the changed behavior or if you need to emulate DataProcessor logic. In other cases DataProcessor performs the below described actions automatically.

To enable DataProcessor, and hence, trigger, sending its data to server, save method should be used with **item ID** and **operation name** as parameters:

```
webix.dp($$("datatable1")).save(1, 'update');
```

To get all changes in the component and send them to server, send method should be used:

```
webix.dp($$("datatable1")).send();
```

Data Validation with DataProcessor

Data validation is enabled by including specific rules for input field. You can read about them here. With rules specified, the validation process starts each time you try to save data to the database.

```
dp = new webix.DataProcessor({
    rules:{
        $all:webix.rules.isNotEmpty
    },
    url: "save.php",
    master: $$('mylist')
});
```

Sending Headers with DataProcessor-based Requests

There exists no possibility to send headers with DataProcessor requests as they are executed in background. However, you can catch Webix **onBeforeAjax** request to modify ANY Ajax request issued from the page:

```
webix.callEvent("onBeforeAjax", function(mode, url, data, request,
headers){
    request.setRequestHeader("Content-type","application/json");
});
```

Note that Webix Ajax module features a **built-in functionality** for sending **headers** with serverside requests. The above solution is only for DataProcessor Ajax requests.

Related Article

- Dataprocessor API - Methods, Properties and Events
- Server Side Connectors

Advanced

Data Connector

The Server Side Connectors together with DataProcessor use their inner logic to interpret CRUD operations and generate the code to update database tables according to the changes made.

- Connectors come in ready-made **PHP, Java, .Net, ColdFusion** versions;
- they simplify **data loading** from database and **data saving**. You needn't write database queries yourself;
- allow basic **CRUD operations on server-side**;
- enable **server-side sorting** and **filtration**;
- feature built-in **protection** against **CSRF** and **XSS** attacks.

As a rule, one and the same connector is used for loading data and saving.

Connector Documentation

- PHP connectors;
- Java Connectors;
- .Net Connectors;
- ColdFusion Connectors.

There you should search for standard **DataConnector** and **JSONDataConnector** that are intended for use with our touch-oriented libraries including Webix.

Data Loading

Connectors perform database queries and return data in **XML** format (standard DataConnector) or in **JSON** (JSONDataConnector). Data formats are described separately.

To load data with the help of the necessary connector, you should:

- write a **connector-based script** that connects to the necessary database AND contains **path to the necessary connector**. In addition, it may contain extra data rendering logic (described below).
- specify this script as **url** for the Webix component you work with.

```
webix.ui({
    view: "datatable",
    ..config..
    url: "server/data.php"
    datatype:"xml"
});
```

```
//or, in case you load data after component init:

$$("grid").load("server/data.php");
```

- Datatype can be omitted in case of JSONDataConnector usage since it returns data in default JSON format;
- In case of long datasets, make use of dynamic loading functionality;
- If you are unsure what to do in case you need to get data from a **different domain** with relations to your app's one, consult the our solution to cross-domain data loading.

Connector Based Script (PHP Solution)

Below is the guide how to write the **script** that will **link component** and **dataConnector** (call it **data.php**, e.g):

1 . Connect to the database and select to the necessary connector.

Specify path to desired Data Connector, and connect to your database:

```
require_once("../../data_connector.php"); //!connector
$dbtype = "MySQL";
$conn = mysql_connect(localhost, "login","password");
```

2 . Initialize dataConnector:

```
$data = new JSONDataConnector($conn, $dbtype);
```

Here you can enable protection against CSRF and XSS attacks:

```
CSRF Protection

ConnectorSecurity::$security_key = true;
```

```
XSS Protection

//safehtml
ConnectorSecurity::$xss = DHX_SECURITY_SAFEHTML;

//trusted
ConnectorSecurity::$xss = DHX_SECURITY_TRUSTED;
```

By default, SECURITY_SAFETEXT protection is used, so you needn't specify it directly.

3 . Render data from the needed table/ tables:

In case of a **single table** you specify its name, the primary key field and field names you'd like to render:

```
$data->render_table("users","id","user, email, status");
```

In case of **hierarchical data**, you also specify **relation id**, which is the name of the field that stores ID of the parent branch for this record:

```
$data->render_table("users", "id", "user, email, status", "",
"relation_id" );
```

In case of **several tables** you need to write a SQL-query, specify the primary key field and fields to render.

```
$data->render_sql("SELECT user, email, status, project FROM users,
projects", "id", "user, email, status, project");
```

Here you can enable dynamic loading, which is useful in case of long datasets. The argument states the number of records loaded initially while the others will be loaded as you scroll or page through the data in the component:

```
$data->dynamic_loading(30);
```

Comments:

Render_table and **render_sql** have common parameters:

- table name / SQL-query;
- name of primary key (ID) field;
- names of fields to take data from;

4 . Customize rendered data

Apart from these two basic functions, you can add extra functionality by using the following methods:

- **configure** - configuring a temporary table for further usage the same way you do it with **render_table**.
- **limit** - customizing the amount of rendered data. Takes for parameters: start line, the number of lines to be rendered, column name to take data from, and sorting direction in this column;

```
$last_value -> configure("wages", "id", "values, start_date");

$last_value -> limit(, 1, "date_start", "desc");
```

Here values from the "wages" table are sorted by date while the descending direction means that the highest line will be the latest value. The result of the query will be the first line of the sorted table.

- **mix** - defining data for a field/column in the UI component. The data comes from the table different from that you use to render data for this component.

```
$data -> mix("wage", $last_value, array("empl_id" => "id"));
```

Here the method works with a column/field name of the component, the value, you'd like to render there and ID correspondence between the tables.

- **filter** - filtering data on serverside. Takes column name and desired value as arguments

```
$data -> filter("field_name = ''");
```

Here the data is filtered by a "field_name" field and shows all records where it is empty.

Connectos have big possibilities of customization, so consult their documentation.

Saving Data with Connectors

To enable saving data with the chosen connector, you should specify it as value of the component **save** property (in addition to mentioning as **url** for loading).

```
webix.ui({
    view: "datatable",
    ..config..
    url: "server/data.php"
    save:"connector->server/data.php"
    datatype:"xml"
});
```

Note that

- The must-have detail here is a **connector** prefix before the path.
- As soon as you define you connector-based script as **save** script, DataProcessor is inited to send changed data to this script.

DataProcessor passes changed data from the component alongside with the operation (*update, insert, delete*) performed on it. This operation corresponds to query type the connector will execute over the database table. If you neeed to customize this behavior - go to DataProcessor docs.

There are two protocols for data transfer with Server Side Connectors

- **Standard** - used for automatic data sending from a component to server (each time you add, edit, remove item);
- **Simplified** - used for manual triggering of server script execution, e.g. issuing Ajax request.

Standard Protocol

- **get** request automatically receives mark *?editing=true*
- **post** request receives data for the record being edited. It includes **data** from all fields of this record AND **!nativeeditor_status** (update, insert, delete) that is defined automatically on the base of user actions;
- **response** comes as in XML that includes type of performed action, original item ID and item ID after processing.

```
ServerSide Response

<?xml version='1.0' ?><data><action type='update' sid='1' tid='1'
></action></data>
```

Original ID (sid) and **ID after processing** (tid) may differ when item adding is performed on client side, during which item gets temporary ID that is changed to database record ID after inserting this item into the database.

Response can be changed. See connector documentation for the needed pattern.

Simplified protocol

- you should specify desired operation (action type) manually in **get** request as "myscript.php?action=insert";
- **post** - properties you pass within Ajax request in form of JSON object;
- **response** - **true/false** for update operation plus **item ID** for insert and delete.

More about simplified data transfer here.

Simplified protocol can be used for Ajax requests:

```
webix.ajax().post("myscript.php?action=update",
    {id: new_id, name: new_name}, //params
     function(response){ //callback
        if(response) //for update it's either true or false
            webix.message("Done");
        else
            webix.message("Update Failed");
});
```

Cross-Domain Data Loading

It's quite uncommun that your app may require data from domain different to the one it's hosted on. Take for instance that you app is hosted at *http://app.mydomain.com*, data is located at *http://data.mydomain.com* and the data-loading script is called "*data.php*".

Standard loading will fail in this case as all browsers block all cross-domain operations for the sake of safety even if you try using a different port.

Nevertheless, there are techniques to overcome these problems. With Webix lib, you can use two of them:

- Cross-Origin Resource Sharing to perform types of requests;
- JSONP Loading for GETting data only.

In both cases you cannot load data with load method that data components normally use for external data. On the contrary, you issue data requests and parse the returned data in callback.

Cross-Origin Resource Sharing or CORS

CORS requires additional server configuration, namely you need to configure data server so that it allows any program from the app's domain (and hence, the app itself) to get data from it.

When you issue such **Ajax request** with Webix ajax function like this:

```
webix.ajax("http://data.mydomain.com/data.php", function(text, data){
    table.parse(data.json());
});
```

The browser sends the following header with the HTTP request:

```
Origin: http://app.mydomain.com
```

You need to make data server (http://data.mydomain.com:) to respond with a specific "allowing" header as well. Like here, app's server is allowed to get data from data server:

```
Access-Control-Allow-Origin: http://app.mydomain.com
```

This can be done in three ways:

- You configure the server’s settings;
- You adjust the code of your app to output the header as a part of server response;
- You use some thirdparty API with server already already set for CORS protocol.

More detailed information about CORS in the Wiki article.

Advantages

CORS technique is standard based, safe and can be used with any kind of data (structured or not) while supporting all types of requests – GET, POST, PUT, DELETE, etc.

Drawbacks

CORS protocol can't be used in IE8 and IE9.

JSON with Padding or JSONP

In frames of Webix JSONP technique requires changes in client-side code, namely you get data with a special **webix.jsonp()** method that hides all technical details of JSONP handling but at the same time you need to do some coding:

```
//client-side pattern
webix.jsonp("http://data.mydomain.com/data.php", {}, function(data){
     table.parse(data);
});
```

Note that in comparison to **Ajax**, **JSONP** always provides **structured data**, not a raw response text.Thta's why the server script must return encoded data:

```
//server-side pattern
$data = get_data_in_some_way();
$json_data = json_encode($data);

$jsonpname = $_GET["jsonp"];

header('content-type: application/json; charset=utf-8');
echo $jsonpname."(".$json_data.")";
```

As you can see, the response data is wrapped, encoded as JSON and is sent back combined with “jsonp” parameter.

Wikipedia article about JSONP.

Advantages:

JSONP works in all browsers and doesn’t require server reconfiguration.

Drawbacks:

JSONP requires extra coding but is limited to work only with structured data as well as suitable only for GET requests. At the same time, it is prone to XSS attacks. It's recommended to resort to JSONP only if you need to support IE8 and IE9 browsers.

Intermediate

Data Binding and Syncing

When you have multiple related views (components) on the page you can **bind** them together or **sync** their data in order that any changes in one of the views would trigger changes in the bound/synced ones. The main component is called a **master** one while the bound one is a **slave**. Master view appears to be the datasource for the slave one.

There are 2 main ways to link data:

- when data in master component changed - slave control gets new data; // **syncing**
- when selection in master component changed - slave control gets new data. // **binding**

In both cases, based on configuration

- slave can take new data from master component;
- slave can take new data from server side.

Binding and Syncing - Difference

When you **bind** data to any component it means that when you **select** an item from one component, this item will be the datasource for another component. Most typically, binding is used with some data-presenting component and a form / htmlform.

When you **sync** data, it means that the whole visible data from one component is at once **copied** to the other.

Data Binding

Data binding is implemented with the help of the **bind** function that is called from the slave component and has a master component as parameter.

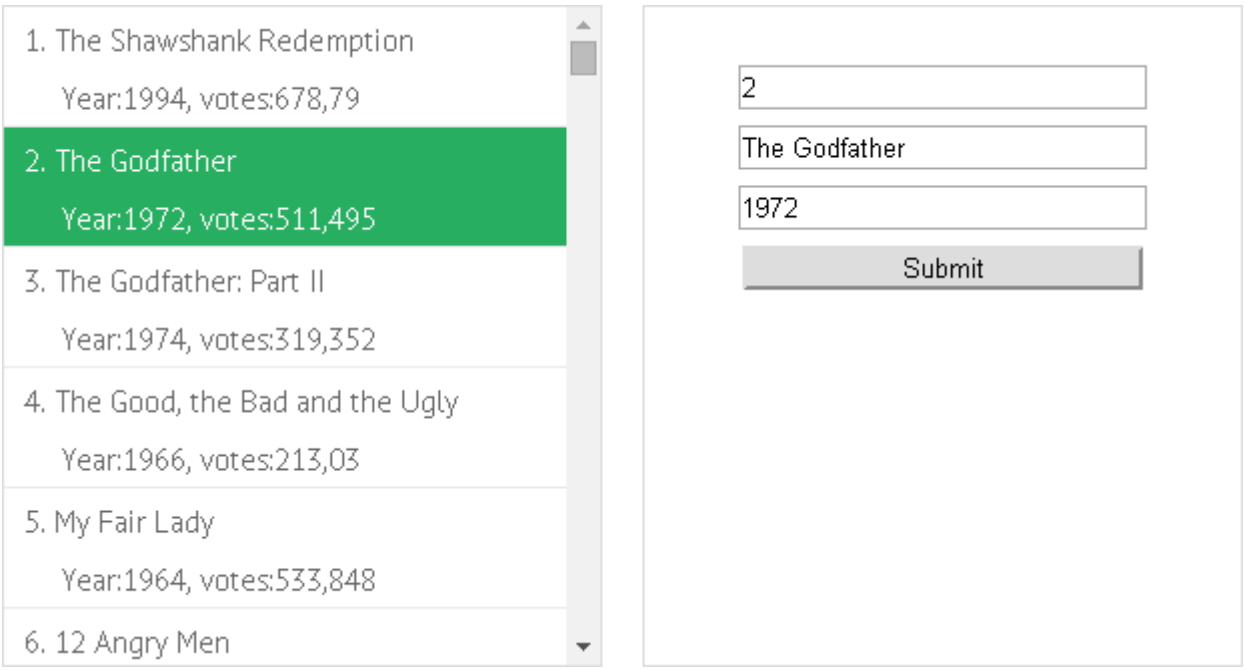
```
$$('htmlform1').bind($$('list1'));
```

In this case, selection of any list item will result in the form filling with data from the list. Be careful to specify **name** property for input fields in the form/htmlform. Its values must coincide with template items of the master component (in case of datatable look at column ID-s):

- component template item = field **name** in a form;
- datatable column ID = field **name** in a form.

Changes in the slave component will affect master one, which means that if you edit the info in the form, then save it (click sumbit button), the list item will have this new data.

```
$$("htmlform1").save();
```



RELATED SAMPLE: Binding to a native HTML form

Slave can as well be **unbound** from the master by the opposite method:

```
$$('htmlform1').unbind();
```

From now on, list changes cannot affect form data and vice versa.

Adding data to master collection When no item is selected in a master component (*here:list*), data in the slave one can still be pushed to it.

In other words, if you call `save()` method for the bound form when no item is selected in the master, a new item will be added to master collection.

The only thing you should do is to remove selection in the master:

```
$$("list1").unselectAll();
//or
$$("list1").setCursor(null);
```

RELATED SAMPLE: Binding to List

Advanced Binding

Data binding features a lot more advanced options to explore:

- it is subject to **customization** with the help of additional parameters;
- it is suitable for binding hierarchical structures;

- it features **event system**;
- it has navite API to set **default data** for cases when no record is selected in a master control;
- it utilizes **cursor concept** to bind to an invisible DataCollection;
- a lot more..

Study the corresponding article for details.

Syncing Data of Two Components

Data syncing allows making a full or partial copy of data from one data-presenting component and pass it to the other one. Any change in the master component results in the same change in the slave one.

There can be more than one slave component. In this case, all slaves change simultaneously on master component change.

Note that this functionality works with components datastores, that's why **data** property is used everywhere.

Full Syncing:

The datasets of the two components are absolutely identical.

```
$$('slave_component').data.sync('master_component');
```

Partial (Filter-Based) Syncing

Slave view shows only the data items that comply to a certain criterion defined by the function. Here only the films shot after 1994 will be shown.

```
$$('dview2').data.sync($$('listA'), function(){
        this.filter(function(data){
        return data.year > 1994;
    });
});
```

Unsyncing

If the syncing functionality is no longer needed, you can call the unsync command for the component that was previously synced with another one:

```
$$('dview2').data.unsync();
```

Related into on data filtering.

Simultaneous Operations within Synced Componens

In case of two and more component synced to one datasource (either a visible component or non-visible dataCollection), it seems nice to **synchronize user operations** as well.

For instance, if an item is selected in one of components, the same item should be selected in the synced one. This is easily done programmatically.

Take for instance we have a dataview and datatable, both synched with the same dataCollection.

```
var data = new webix.DataCollection({
    url:"../my_data.json" //load data from an external file
});

$$("dataview_1").sync(data);
$$("datatable_1").sync(data);
```

The event system for selection offer the **onAfterSelect** event. The function that is fired on this event takes the **ID of selected item** as parameter:

```
$$("dataview_1").attachEvent("onAfterSelect", function(id){
    $$("datatable_1").select(id);
}); //item ID of dataview coincides with that of datatable  row

$$("datatable_1").attachEvent("onAfterSelect", function(id){
    $$("dataview_1").select(id.row);
}); //item ID of datatable row coincides with that of dataview item
```

RELATED SAMPLE: Data Binding

Data Binding and Syncing with Non-UI objects

One-Time Syncing

One-time syncing allows for synchronizing the data of two components (one of them can be DataCollection) at a chosen time of the application flow **without tracking data changes afterwards**.

Such syncing presupposes populating one component with the data of another one and is done with importData method:

```
$$("listB").data.importData($$("listA"));
```

Note that it's a DataStore method, so it should be called for component **data**.

Data Binding and Syncing with Non-UI Objects

The webix library contains **view-less** yet meaningful elements. They have no graphic representation and are used to store data in them.

- **DataValue** - stores a single value like any input control (text, button, label, etc.);
- **DataRecord** - stores a single record like form and htmlform;
- **DataCollection** - stores a non-hierarchical collection of records (like list, dataview, datatable, etc.);
- **TreeCollection** - stores a hierarchical collection of records (like tree and treetable).

These collections can store data and have the same API as their "visible" siblings. Such methods as **bind()** and **sync()** can be used with them as well.

How can you benefit from them?

DataCollection and TreeCollection Usage

Non-ui components store some info you'd like to use later in visible components. For instance, in the DataCollection you can specify the needed dataset and later on, sync all the data-presenting components with it so that they all are fed with the same data.

```
var data = new webix.DataCollection({
    template:"#rank#. #title# - #year#",
    data:big_film_set //the datasource
});
```

Now let's init the list and sync it with out invisible DataCollection

```
webix.ui({
    view:"list",
    id:"list",
    template:"#rank#. #title# - #year#",
    data:"" // we needn't specify anything here
});

$$('list').data.sync(data);
```

Now, if we bind a form to the DataCollection, it will be a slave view in relation to the synced list as well! And any data from the form will be added to dataCollection, and then to list.

However, data will not fill the form unless we refer to cursor concept.

Cursor Concept

Cursor position is the ID of an item that the cursor is set on. Cursor should be set in case several components are bound to/synced with one and the same source, e.g. one and the same DataCollection, in order to avoid conflicts between different component referring to one and the same datasource and making changes upon it.

The invisible **dataCollection** is a master component for both the form and data-containing components synced with it. But since it's invisible, data cannot be operated on. The issue is solved by using any component to set cursor to an item from dataCollection each time this item is selected in the chosen component.

For instance, you'd like a list to be "volonteer" to get to dataCollection during form filling.

```
$$('list').attachEvent("onAfterSelect", function(id){  data.setCursor(id);
});
```

As a result, data is saved to dataCollection after editing as well as to all the synced components without any additional manipulations.

RELATED SAMPLE: DataCollection Syncing

Server-Side Integration with non-UI Components

Server-side work is enabled with **dataConnector** to load data and **dataProcessor** to save changes back to the database. It can be used for both "visible" and "invisible" (non-ui) components.

To use dataProcessor with dataCollection synced with some component on the page, you should go through a three-stage process.

1 . Define a **dataCollection** and populate it with data from the database:

```
store = new webix.DataCollection({
    url:"../common/connector.php",
    datatype: "xml"
});
```

2 . Declare a **dataProcessor** for the dataCollection:

```
var dp = new webix.DataProcessor({
    master: store,
    url: "../common/connector.php"
});
```

3 . **Sync** a data-presenting component with the dataCollection:

```
list.sync(store);
```

RELATED SAMPLE: Data Syncing with DataCollection

DataCollection and Datatable Syncing

DataProcessor and server-side integration work the same way with all components yet in case of datatable extra functionality appears.

DataConnector allows rendering data from one table into a component (**render_table** method) or displaying data from several tables using custom SQL-query (**render_sql** method). More info about this in the Serverside Integration article.

Working with Database Tables: ID-to-Text Transformation

Tables in the database are often connected by ID while on client-side, within the component, we require that text values should be shown. The solution hides in loading data from the needed tables into **different dataCollections** and refer to **datatable columnn configuration** to render the database data.

First, create several **dataCollections** and load data from connected database tables.

```
Table Rendering

var employee = new webix.DataCollection({
    url:"data/employees.php",
    save:"connector->data/employees.php" //dataprocessor, short init form
});

var customers = new webix.DataCollection({
    url:"data/customers.php"
});

var orders = new webix.DataCollection({
    url:"data/orders.php",
    save:"connector->data/orders.php"
});
```

What is there in these collections?

While database connect parameters are the same for two collection, they differ in tables they render:

```
--library connectors for working with server side
require_once("./config.php");
$data = NEW JSONDataConnector($db,"SQLite3");

--for employee collection
$data->render_table("employees","id","email, fullname, login");

--for customers collection
$data->render_table("customers","id","lastname, firstname, job, address,
city, zip, country");
```

```
--for orders collection
$data->render_table("orders","id","date, employee, customer, status
[,other fields]);
```

Then create a datatable and sync it with either fo these collections.

```
$$("dash-ord-active").sync(orders);
```

This will be the main datasource for the component. As far as in the **"orders" table contains just IDs** of dedicated customers and orders, they **should be transformed into text** while rendering to datatable .

For these needs there's a **collection** property in datatable column API to point to a datasource different from that of the whole component.

```
{ view:"datatable", id:"dash-ord-active",
 columns:[
        { id:"date", header:"Order Date" },
        { id:"employee", header:"Employee", collection:employee},
                { id:"customer", header:"Customer", collection:customers},
                { id:"status", header:"Status" },
                { id:"fee", ...},
                { id:"taxes", ...},
                { id:"total", ...} //other columns
        ]
}
```

Two columns for the datatble are fed directly from **employee** and **customers** collection where they get text values instead of ID-s.

Datatable column configuration is described in related datatable documentation.

Working with Bound Components

Binding makes selected record of one component a datasource for another. The basic binding tie is established between two components using their IDs of variables they are stored in (if any).

```
$$("myform").bind(list);
```

It has already been described.

Here different aspects of bata binding are examined.

Binding Rules

The **bind()** method can take two optional parameters to customize the binding pattern - **rule format**.

The customization is only possible in case slave component is based on DataStore (all data management components except for tree-like) or TreeStore (tree and treetable).

Binding rule defines a scheme according to which records in the slave component are filtered. It can be either a string or a function.

Binding rule as a function

As function, binding rule takes 2 parameters:

- **slave** - slave data object;
- **master** - master data object.

```
Slave list displays records with category equal to master value

$$("list").bind($$("richselect"), function(slave, master){
    return slave.category == master;
});
```

Here list data is filtered according to the option you choose within a richselect control.

RELATED SAMPLE: List Data Bound to Richselect

Additionally, binding rule can cancel bind applying in runtime.

```
Slave datatable displays only records with movie property equal to master record
ID

gridb.bind(grida, function(slave, master){
        if (!master) return false; //cancelling bind applying
```

```
    return master.id == slave.movie;
});
```

Here binding is cancelled in case no data is sent from master (i.e. no record is selected in master).

RELATED SAMPLE: Linking tables

Binding rule as a string

If master component is loaded with hierarchical data you can define a binding rule with the help of the following flags:

- **$level** - only immediate children of a selected item are pushed to slave component. By default children are introduced by **data** key in all the supported data formats:

```
Grid will show only children of a selected tree node

$$("grid").bind( $$("tree"), "$level");
```

Input JSON of such a tree is as follows:

```
{ id:"..", value:"..", data:[
    //will be shown in slave component
    { id:"..", value:".." },
    { id:"..", value:".." }
]}
```

RELATED SAMPLE: Data binding

- **$data** - the object of a selected item (without children) is pushed to a slave component. Here you should pass the **format** of subdata presentation to specify data items necessary for the slave.

Format of bound data presentation

Format can be a string that defines the **key** by which desired data is set in the item object. As a rule, it differs from the key by which children are set (if they are set via "data" (default), it's more reasonable to use **$level** rule):

```
$$("grid1").bind( $$("tree"), "$data", "records");
```

Input JSON data of such a tree is as follows:

```
{ id:"..", value:"..", records:[
    //will be shown in slave component
    { id:"..", value:".." },
    { id:"..", value:".." }
]}
```

RELATED SAMPLE: Data binding - subdata

Format can be set via a function where you describe binding behavior manually:

```
$$("grid2").bind( $$("tree"), "$data", function(obj, source){
    if (!obj) return this.clearAll();
    var fulldata =
[].concat(source.data.getBranch(obj.id)).concat(obj.records);
    this.data.importData(fulldata, true);
});
```

Here we get item chilren with the help of a **getBranch()** method, combine them with data set by **records** key and import the resulting array into slave datatable.

RELATED SAMPLE: Data binding - subdata

Binding Events

- **onBindApply** – fires the moment binding is applied;
- **onBindRequest** - fires when the component is ready to receive data from the master component;
- **onBindUpdate** – fires when the value in the slave view changes and *save()* is called to update master.

```
$$("files").bind($$("folders"));

$$("files").attachEvent("onbindapply", function(){
    // makes the preview template empty
    $$("preview").setHTML("");
});
```

Preventing from Undefined Values

What happens when **no selection** is made in the master component? If no measures are taken, the slave component would show **undefined** values.

This concerns only data components that have **template** property in their constructor and doesn't concern form and htmlform where binding is set according to the **names** of their inputs.

The **defaultData** property of the master one helps solve the problem.

```
webix.ui({
    rows:[
        {view:"list",
        template:"#rank#.#title#",
        data:...
```

```
        defaultData:{
            rank:"0",
            title:"default Item"
            }
        },
        {view:"template",
        template:"#rank#.#title#"}
    ]
})

$$("template1").bind($$("list1"));
```

RELATED SAMPLE: Default Data

Now, when no item is selected in the list, template will show default values (as set by the defaultData) instead of undefined ones.

Look to CollectionBind API for more details.

Cursor Concept for Bound Collections

Cursor concept is used to control focus within the application with bound and synced data-management components. Its position is the **ID of the active data item.**

Simply put, cursor stores the ID of a data item from the master control for which binding is performed at the moment.

You can get and set cursor position with the help of the following methods:

- **setCursor(cursor);** - string, number - sets the position of the cursor, the ID of the necessary data item.
- **getCursor();** - gets the current cursor position.

```
var cursor = master.getCursor();
```

Cursor can be deleted to remove the current bind link:

```
master.setCursor(null);
```

If a **form** (slave) is bound to a **list** (master) and cursor is removed from the list - form will be emptied.

Cursor is as well useful when working with non-visible DataCollections.

Using Binding for Server Side

As a rule, slave component doesn't communicate directly from server-side. It receives data from the master view it is bound with and the changed data is sent first to the master that handles communication with the server.

However, there exists a possibility to get serverside data for the slave based on master selection - dataFeed functionality.

```
webix.ui({
    view:"form",
    id:"myform",
    ...//config
    dataFeed: "slave_data.php"
});

$$("myform").bind($$("mydatatable"));
```

Datafeed defines URL that will be used by slave control to sent serverside request the moment selection in the master component changes.

The functionality works the same for form and collections (data components) yet the URL parameters differ:

- for forms the request URL looks like: "slave_data.php?action=get&id="+obj.id
- for data components the request URL is as follows: "slave_data.php?filter[id]="+obj.id

where **obj** is selected data item in the master component.

Saving Data of Multiple Bound Forms

When a form is bound to a component, data from the selected item is transmitted to the form as well as any changes you make within the form are saved automatically to the selected item.

It happens on calling the **save();** method

```
$$("myform").save();
```

But if you have **two forms** bound to one and the one component and you save each of the forms separately, the data from the previously saved one is lost. The solution is the saveBatch() method that makes simultaneous saving of several forms.

It is called from the master component rather than from either of the slave forms.

```
$$("mydatatable").saveBatch(function(){
     $$("myform1").save();
     $$("myform2").save();
});
```

Intermediate

Data Updating

Component items can be edited within the component itself, within its rows and cells, provided that the component is switched to **editable** mode.

Data updating allows for editing info in dedicated **input fields** like UI text. It works regardless of the component's editability state.

Updating data for selected items can be implemented in two ways:

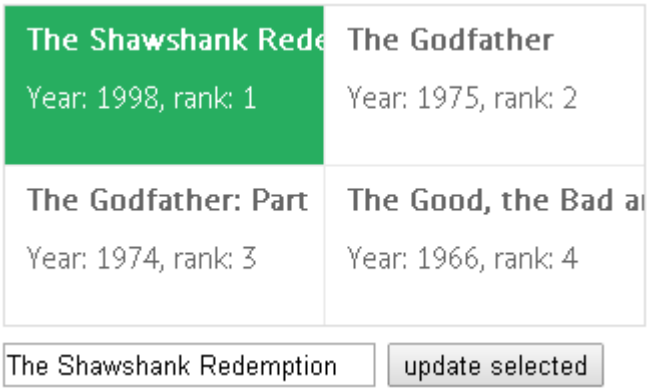
- one by one - with the help of **updateItem()** function;
- in groups - by pushing data from **bound** form / htmlform. Data Binding

UpdateItem() Function

Updating process happens in two stages:

1 . Placing selected data to some editor (ui-related text or dedicated HTML input field) with the help of an attached event.

```
<input type="text" id="value" style='width:300px' value="new value" />
...
$$('mylist').attachEvent("onAfterSelect", function(id) {
    var value = this.getItem(id).title;
    document.getElementById("value").value = value;
}); // as a result, the item title from the dataset appears in the text
field
```



2 . Saving the edited data back to the component. It is implemented with an update function that takes ID of selected item as argument. Called from component object, it is included into the complex update function.

```
function update_item() {
    var data = $$('data'); // ID of the data-presenting component
```

```
    var sel = data.getSelectedId(true);
    if (!sel) return;  //if nothing is selected, there's nothing to
update!
    var value = document.getElementById("value").value; //deriving new
value from the input
    for (var i = ; i < sel.length; i++) {
        var item = data.getItem(sel[i]);
        item.title = value; //setting the new value for the item
        data.updateItem(sel[i], item); //the dataset is updated!
    }
}
```

RELATED SAMPLE: Server-side Integration: Dataview

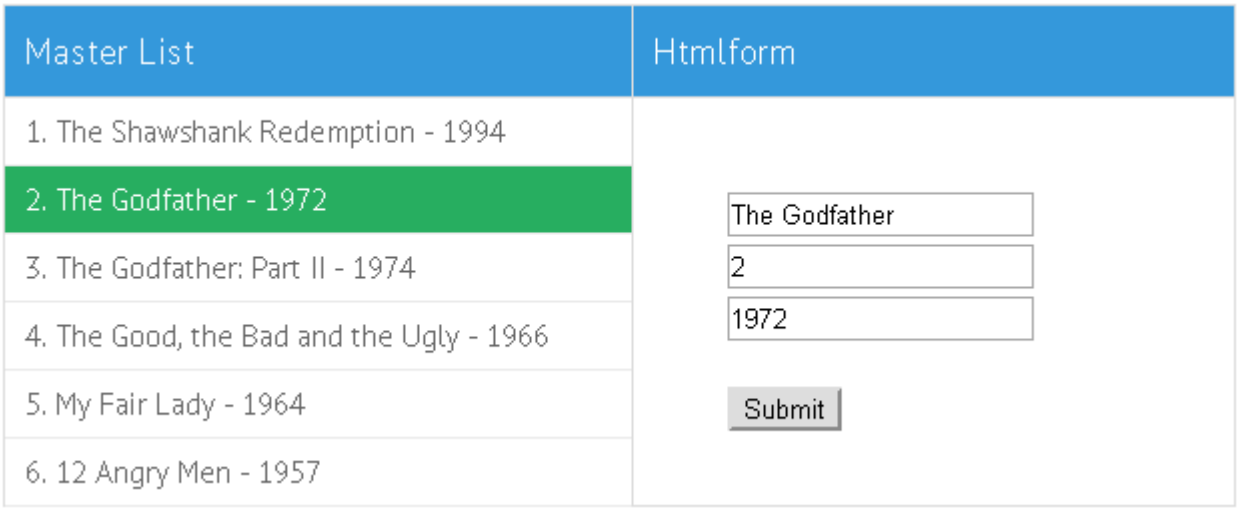
Form / HTMLform binding

Data binding helps greately simplify the process and makes editing of several item values convenient. It ensures that a data component (dataview, list, etc.) transmits data to a form / htmlform each time an item there is selected.

More info on Data Binding

```
$$('htmlform1').bind($$('listA'));
```

The function above makes list a datasource for form, so when an item in the list is selected, the form is filled. Input names in the form must coincide with item properties from teh initial dataset (those you include into #hash signs# within data template) for the list.



Having finished editing data in form fields, call the **save** function to push new data into the list.

```
<input type="button" value="Submit" onclick='$$("htmlform1").save()'/>
```

RELATED SAMPLE: Data Binding

The same functionality is used when updating data items loaded from the database. Read about Server-side Integration in the corresponding chapter of the documentation.

Configuring Components

- Managing Visibility of Components
- Disabling/Enabling Components
- Adding Headers to Components
- Dynamic UI Modifications
- Adding and Deleting Multiview Cells Dynamically
- Changing Properties of Components
- Sizing components
- Resizing
- Scroll Control
- Date and Number Localization

Beginner

Managing Visibility of Components

By default, all ui-related components except for window and popup are visible on the page after their initialization.

Show() and Hide() methods

To show and hide any component in layout you should call **show()** and **hide()** methods respectively:

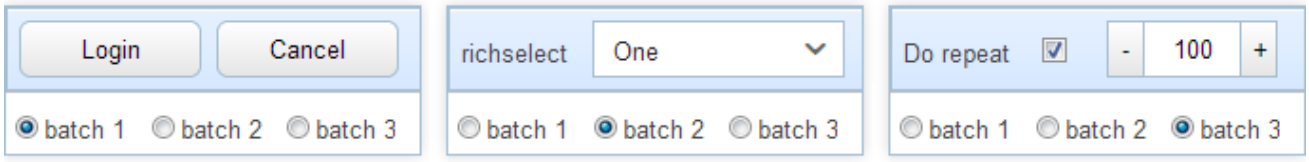
```
$$("my_window").show();
$$("my_button").hide();
```


The alternative to the **hide()** is a **hidden:true** parameter that is included into the component constructor. The component can be shown later with the **show()** method.

When manipulating visibility of components it seems useful to know whether the component is visible on the page at the moment. This can be checked with the help of **isVisible()** method that return true or false depending on the component's state.

```
$$("my_window").isVisible();
```

Manipulating Visibility of Control Groups



At the same time the library allows grouping controls and changing their visibility in groups. It's useful when you want to define different button sets for different states of the application.

To show and hide groups of elements you make use of the **batch** property and associated with it **showBatch()** method.

Batch groups elements by one and the same value that is as well assigned to each of the needed controls.There can be as many batches as you wish depending on the quantity of the manipulated controls.

One of the batches(groups) is shown initially. It is defined by **visibleBatch** property.

Take following steps to use this functionality

1 . Include batch property into desired controls.

```
webix.ui({
        view:"toolbar", visibleBatch:"1", // batch "1" is visible
initially
        cols:[
            {view:"button", value:"first", batch:"batch 1", align:
"left"}, //various controls
            {view:"button", batch:"batch 2", ... },
            {view:"button", batch:"batch 3", ... },
            { ....}
            ]
     });
```

2 . Assign the function to change visibility of controls to a suitable control.

Here radio options coincide with batch values from the code above:

```
    { view:"radio", id:"rad", label: "select", click:"change_batch",
options:[
        "batch 1","batch 2", "batch 3"
        ] } //chanding radio buttons triggers the function execution
```

3 . Write the function to show the controls with the chosen batch value.

```
function change_batch(){
    var mode = $$("rad").getValue(); // returns "batch 1", etc..
    $$("mybar").showBatch(mode); //the value is passed into the function
}
```

RELATED SAMPLE: Hiding and Showing Groups of Controls

Beginner

Disabling/Enabling Components

webix UI library allows tempoparily disabling functionality of some component. It remains visible yet becomes dim, and neither of it's elements works.

The functions to disable and enable components are as follows:

```
$$("cell").enable();
...
$$("cell").disable();
```

RELATED SAMPLE: Disabling Components

When can it be useful?

- For instance, you have a dataview and a form and want to disable a form until an item from the dataview nearby is selected.
- Or you want to temporarily disable a component until server operation is performed (e.g. data saving) in order that users can't click buttons and make selection.

To check whether the component is enabled or disabled now, use the appropropriate method:

```
$$('cell').isEnabled(); //returns boolean values
```

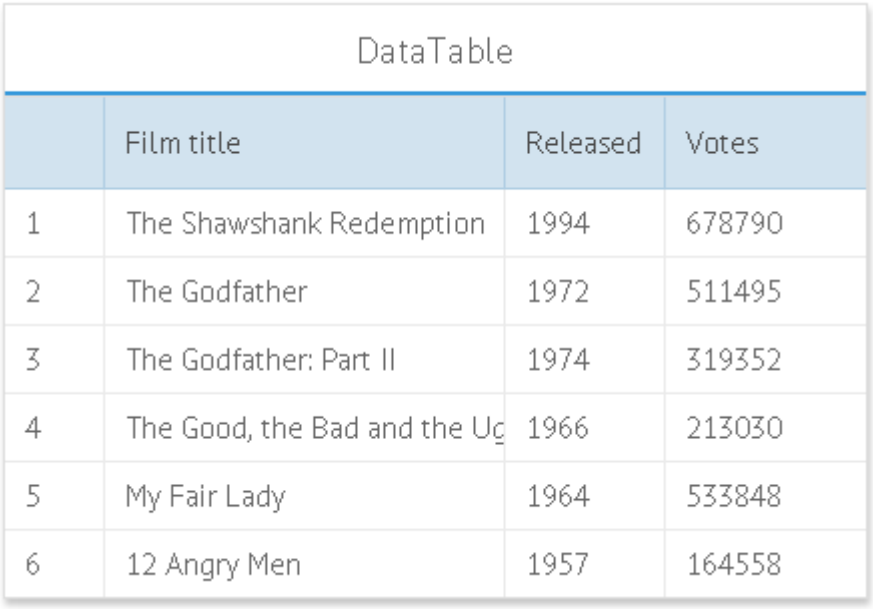
Intermediate

Adding Headers to Components

Header is a blue section above the main contents of a component or layout row/column.

Depending on the context, header is defined in two ways:

- as template type;
- as component property.



RELATED SAMPLE: Window with a Component Inside

Header as Template Type

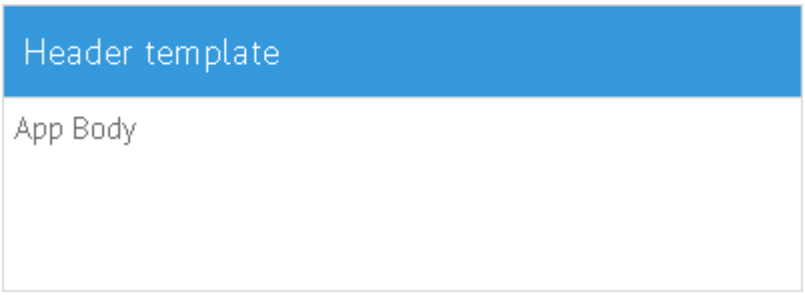
This way of header initialization is used within layout rows and columns. Here header is just one of template types.

If you want to make a header for, say, dataview, make two rows and place header in the first row while the dataview should be in the second one.

```
Type:"header"

webix.ui({
    rows:[
        { view:"template", template:"some text", type:"header" }, //1st
row
        { ....} //2nd row
        ]

});
```



BTW, another template type, namely **section**, allows for setting a specific header that looks like a thin line that encircles the component and has a heading above.

```
Type:"section"
```

```
{ view:"template", template:"some text", type:"section" }
```



During header initialization the **view:template** line is optional.

RELATED SAMPLE: Template Types

Header as Component Property

Still, some components have built-in means of header building.

- Accordion Items
- Tabview Tabs
- Property Sheet

Accordion item is divided into **body** and **header**

```
{ view:"accordionitem", header:"Pane ", body:"text' }
```

Headers of **tabview tabs** are defined as header properties of tabview **cells**:

```
{ view:"tabview",
  cells:[
    {
      header:"List",
      body:{
```

```
        view:"list", ...
      }
    }
  ]
}
```

RELATED SAMPLE: Tabview: Basic Initialization

Property sheet looks like a grid with sections separated by blue headers. In essence, any line without data becomes header, the heading text being defined by **label** property.

```
{ label:"Layout" }, //header
{ label:"Width", type:"text", id:"width"}, //line with data
```

RELATED SAMPLE: Property Sheet: Basic Initialization

UI-related **window** features header defined by its **head** property.

Dynamic UI Modifications

Webix lib offers several ways to alter the already initialized layout:

- you can **add any view** to the existing layout, including complex views that have a number of child components;
- you can **remove any view** from existing layout;
- you can **reconfigurate** layout-like comopnent by passing an array of new children views to it;
- you can **replace** any view in layout by another view.

Let's consider these possibilities in detail:

Adding/Removing Views Dynamically

You can add and remove components dynamically after you've already inited layout and set some components. It works for:

- layout rows and cols;
- multiview cells;
- accordion and carousel panels;
- toolbar controls;
- form controls.

Adding and removing views is implemented with the help of the **addView()** and **removeView()** methods.

Adding View

To add the new view (layout row and column, multiview cell, accordion panel, as well as any toolbar or form control) - specify the **view object** and **position** to insert.

```
Adding a button to the toolbar

var id = $$("toolbar").addView({
    view:"button", value:"new button",
}, index);
```

Index defines position where the new component or control should be inserted among the children of the parent component. It can be

- first (index 0);
- last (index -1);
- any. Index of any view can be derived with the help of index method that can be called for layout-like Webix components (layout, multiview, accordion, carousel, form, toolbar).

```
Adding View

webix.ui({
    id:"layout1",
    rows:[
        {view:"text", id:"text1"},
        {view:"button", value:"Add View", click:addView}
    ]
});

var pos = $$("layout1").index($$("text1"));
$$("layout1").addView({view:"text", id:"text2"}, pos);
```

RELATED SAMPLE: Basic Initialization

The newly added view can be sized during adding according to common rules as well as later on via property (re)defining.

Removing View

removeView() method removes components from layouts and is called from the parent component to remove any of its children. The method takes a child **view object** or its **id** as an argument.

```
Removing a button from the toolbar

$$("toolbar").removeView(id);
```

RELATED SAMPLE: Basic Initialization

Rebuilding Application Layout

Here we look at advanced possibilities of webix.ui() method.

Besides its main purpose - instantiation of Webix component, it method can also be used for changing layout provided that additional params are passed to it.

All in all, its parameters include:

- **configuration object** - (object, array) - a JSON object with the application config of any level of complexity;
- **parent element** - (id or object) - Webix component that acts as a parent to an object you are going to init (first param);
- **replacement object** -(id, index or object)- Webix component in a parent object that will be replaced by an object you are going to init (first param).

Thus *webix.ui()* constructor can be used to:

- **redraw layouts** (layout, multiview, form, accordion) that feature arrays in their structure:

```
webix.ui({
    view:"form", id:'myform', elements:[...]
});

//redraw form with new elements
webix.ui([..new elements..], $$('myform'));
```

RELATED SAMPLE: Dynamic Layout Rebuilding

- **replace** any existing object:

```
webix.ui({
    id:'mylayout',
    rows:[
        {view:'toolbar', ...},
        {view:'datatable', id:'mydatatable' ...},
    ]
});

//replace datatable
webix.ui({..new config..}, $$('mylayout'), $$('mydatatable'));
```

RELATED SAMPLE: Dynamic Layout Rebuilding

Reloading layout from the server

To load a new configuration from the server - you can use the following technique:

```
webix.ajax("config.json", function(text){
    webix.ui(webix.DataDriver.json.toObject(text), $$('mylayout'),
$$('mydatatable'));
});
```

where the *config.json* contains the new configuration.

Adding and Deleting Multiview Cells Dynamically

Here two approaches are possible:

1 . If you are using **multiview** with a switching control (**tabbar** or **segmented**) - you should treat multiview and a switching control separately. In other words you should **add** and **remove options** for tabbar (segmented button) and at the same time **add** and **remove views** for the multiview.

2 . If you are using **tabview** (integral component that consists of a built-in multiview and tabbar) - note that it features dedicated methods to **add** and **remove views** together with the corresponding **tabs**.

Using Tabbar and Multiview

Both Tabbar and Segmented Button feature separate parts - tabs or segments - that are defined by **options** property.
Each of its options can be connected with a separate view or a separate click handler.

Table 1 Switch controls

`view:"segmented"`	One | Two | Three
`view:"tabbar"`	One | Two | Three

Tabs/segments can be added and deleted on the go with the help of the following methods:

- **addOption()** - adds a new tab/segment according to the provided configuration;
- **removeOption()** - removes the tab/segment according ot the provided ID.

Multiview cells can be added and deleted denamically by using:

- **addView()** - adds view to layout (multiview is layout) according to the provided configuration;
- **removeView()** - removed layout cell by the given ID.

```
Adding

rows:[
    {view:"tabbar", id:"tabs", multiview:true, options:[], ..}, //empty
tabbar
    {view:"multiview", id:"views", cells:[
        {template:"..."} //multiview should have at least one cell
    ]}
]
...

function open_new_tab(id){
    var item = $$('list1').getItem(id);
```

```
    //adding a new cell to multiview
    $$("views").addView({ view:"template", id:item.id,
template:item.title});
    //adding option to the tabbar
    $$('tabs').addOption(item.id, item.title, true);
};
```

Deletion

```
function del_tab(){
    //getting the ID of the active tab
    var id = $$("tabs").getValue();

    //removing tabbar option
    $$("tabs").removeOption(id);

    //removing corresponding view
    $$("views").removeView(id);
}
```

RELATED SAMPLE: Dynamic Adding/Deleting of Tabs in Tabbar

The logic of view adding and removal can be studied in detail in the Dynamic UI Modifications.

Using Tabview

Tabview is a hybrid of a multiview and tabbar.

It features two methods that allow adding and removing cells together with corresponding tabs.

Each combination of a tab and a cell is configured as:

```
{ header:"header_title", body:{ ..view config ..}
```

The same configuration is passed into the addView() method:

```
$$("tabview1").addView({
    header:"New Tab",
    body:{ //webix.uid() generates a random number
        template:"New content "+webix.uid()
    }
});
```

The tabview cell can be removed by removeView() method that required an ID if this cell:

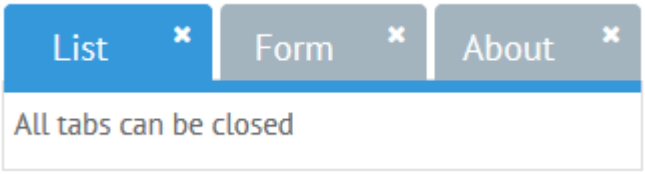
```
var id = $$("tabview1").getValue(); //id of active cell
```

```
$$("tabview1").removeView(id);
```

RELATED SAMPLE: Tabview: Adding and Removing tabs

Built-in Tab 'Close' Button

Webix tabbar can be equipped with a 'Close' button:




```
//all tabs
{view:"tabbar", close:true, options:[]}

//some tabs
{view:"tabbar", options:[
    {id:1, value:"Tab 1", close:true},
    {id:2, value:"Tab 2"}
]}
```

If a closable tab is used in a **separate tabbar**, you should provide logic to remove the corresponding view.

```
tabbar.attachEvent("onOptionRemove", function(id, value){
    $$("multiview").removeView(id);
});
```

ID of the tab concides with ID of the view, normally.

If a closable tab is used within a **tabview**, the corresponding tab will be removed from the app automatically.

Intermediate

Changing Properties of Components

Defining New properties

The properties of Webix components can be changed after initialization each time the **define()** method is applied to them. It takes the following arguments:

- a **component's property** you want to change;
- the **new value** for this property as arguments.

The method can be applied to any object pointed to by its ID:

```
//loads new data to the list
$$("mylist").define("url", "new_data.xml");

//sets new height for the list
$$("mylist").define("height", 300);

//changes the label of the button
$$("button1").define("label", "New value");
```

The method as well takes **hash** of **property-value** pairs to change multiple properties at once:

```
//set new data and height for the list
$$("mylist").define({
    url: "new_data.xml",
    height: 300
});
```

Applying Property Changes

In order that new properties came into force, you should apply either of these methods to the redefined component:

- **refresh()** - used to repaint a single component and doesn't affect page structure. It's used for data-containing components (list, datatable, chart etc. and all controls).
- **reconstruct()** - used to rebuild the component that affect page structure. It concerns components that build application layout and house other components and control: layout, multiview, accordion, carousel, form and toolbar.

```
$$("layout").define("rows", [ {view:"button", ...}, {view:"label", ...}
]);
```

```
$$("layout").reconstruct();
```

- **resize()** - adjusts a component to the new size.

Changing the Component's Config

The desired configuration parameter can be accessed and changed through the **config** property as well.

```
//reading height
var height = $$("myList").config.height;
//setting new height
$$("myList").config.height = 400;
$$("muList").resize();
```

Changing component properties via config comes handy when getting and setting values for **component items** that are stored in an **array** that is a property itself.

```
Defining width of a datatable column

webix.ui({
    view:"datatable",
    id:"mytable",
    columns:[
        { id:"col_1", header:"Film title",width:200},
        { id:"col_2", header:"Released" , width:80}]
        })
//reading 1st column width
var colWidth = $$("mytable").config.columns[].width; // 200

//resizing the 1st column

dtable.resize();
```

Redefining Item Properties

Type with data-containing components denotes the way each data item looks like rather than look-and-feel of the whole component.

```
webix.ui({
    view:"list",
    type:{
        width:200, //item width
        height: 50
        },
    width: 250 //list width
    ..
});
```

To redefine width of the item, you should use **customize()** method instead of **define()** - the arguments are the same:

```
$$("list1").customize({ width:210}); //item width changes
$$("list1").refresh();
```

More about Type Implementation.

Beginner

Sizing components

You can set dimensions for the components and their items in 5 ways:

- Fixed Sizing
- Auto Sizing
- Adjusting mobile apps to screen size
- Relative Sizing
- Dynamic sizing

Later on, you can dynamically resize the components.

Fixed Sizing

Fixed dimensions are set by **width** and **height** parameters that can be applied to components and their items as well as to the whole layout.

```
Layout Sizing

webix.ui({
    id:"layout",
    height:600,
    width:700,
    rows:[
        {template:"row 1",height:20},
        {template:"row 2"},
        {cols:[
            {id:"a1", template:"column 1", width:150}
        ]}
    ]
}).show();
```

RELATED SAMPLE: Layout and Resizer

When giving fixed dimensions to the components in layout, don't forget that layout rows and columns will adjust to the sizes of the nested components automatically.

Control Sizing

Controls feature **inputWidth** and **inputHeight** properties that define dimensions of input areas and button text values.

```
Button Sizing

{ view:"button", value:"Prev", type:"prev", width:100, inputWidth: 70,
```

```
height:20 }
```

Read more about the possibilities of control sizing in the Controls Guide

Component Item Sizing

If you want to specify the dimensions of an item rather than the component, include height and width parameters into the component's **type** property .

```
Dataview Component and Items with Fixed Dimensions

webix.ui({
    view:"dataview",
    width: 520,          //  component's dimensions
    height: 270,
    type:{
        height: 90,      //  dimensions of each dataview item
        width: 250
    }
    ...
});
```

RELATED SAMPLE: Dataview Italic

Sizing Limits

During dynamic resizing or when the component is inited in layout with at least one resizer line, minimum and maximum dimensions are very helpful. They are set with the help of the following properties:

- **minWidth** and **minHeight** (number) - on resizing the component cannot takes less space than specified. If more space is available, the component will take it.
- **maxWidth** and **maxHeight** (number) -on resizing the component cannot take more space than specified. At the same time, the component may take smaller width nd height values and take smaller space.

Auto Sizing

1.Autosizing to the dimensions of the parent container.

By default components are autosized to the dimensions of the parent HTML-container.

```
Inline CSS

<div id="areaA" style="width: 320px, height: 400px"></div>
<script type="text/javascript" charset="utf-8">
    webix.ui({
```

```
        container: "areaA", //corresponds to the value of <div> id
 parameter
        view:"list",
        ...
    });
</script>
```

Auto sizing adjusts the component or its item to available space (parent container, if these's the one; otherwise, to the document body). If there're several components or several layout columns on the page, each of which, they will equally distribute available space among them.

2.Content Autosizing.

If you define **autoheight** as *true* the height of layout rows will be adjusted to their contents.

How does this property work? By setting autoheight to the template we let it 'go' to its contents (text, as a rule) and calculate itself the height of the text on the screen. This height will be the template's height and the corresponding layout row will adjust to this parameter.

```
Autoheight for view Template

rows:[
    { template:"html->my_box1", autoheight:true},
    { template:"Area 1"},
    { template:"html->my_box2", autoheight:true}
]
```

RELATED SAMPLE: Autosizing to Content

Some components also feature the **autowidth** property.

3.Item Content Autosizing

To adjust item's width to its contents you should use **adjusting** or **auto height** depending on the component.

Datatable column width is adjusted to its contents:

```
Datatable Column Sizing

columns:[
    { id:"rank", header:"", css:"rank", adjust:true },
    ...
]
```

RELATED SAMPLE: Adjusting columns to the content

Auto height is set within component **type** that defines properties of an item (not the whole component). Each item is adjusted to its content, which may result in **variable dimensions** of items:

```
//list with variable item height
webix.ui({
    view:"list",
    width:250,
    type:{
        height:"auto"
    }
});

//dataview with variable item width and height
webix.ui({
    view:"dataview",
    width:250,
    type:{
        height:"auto",
        width:"auto"
    }
});
```

In addition, component width and height can be adjusted to the **predefined dimensions of HTML elements** of its items. For these needs, set **sizeToContent** property for a component:

```
Dataview Item Sizing

webix.ui({
    view:"dataview",
    template:"#title#<br/> Year: #year#, rank: #rank#",
    sizeToContent:true
});
```

Note that even in case these HTML elements are different for different items, the dimensions are measured only once (for the first input, as a rule) and then sizing is performed. At the same time, some of them may be smaller or bigger.

RELATED SAMPLE: Autosizing Items to Content

Adjusting Mobile Apps to Screen Size

Webix offers “full screen” mode to make the application take all available screen space while at the same time hiding browser’s toolbars. In this mode, the app will take up the entire screen of the mobile device.

Fullscreen mode is enables with webix.ui.fullScreen() command before UI initialization:

```
webix.ui.fullScreen();
webix.ui({
    rows:[...]
});
```

Relative Sizing

Relative dimensions can be defined with the help of the **{gravity:x}** parameter which makes one component **x** times bigger then the other.

```
One of the 'Save' buttons is twice bigger then the other one.

webix.ui({
    view:"toolbar"
        { view:"button", value:"Load", width:200 },
        {  margin:4, borderless:true, rows:[
                        { view:"button", value:"Save",  gravity:2 },
                        { view:"button", value:"Save"}
                        ...
         })
```

RELATED SAMPLE: Toolbar: Multiple Rows

Dynamic Sizing

Dynamic sizing ensures adequate visibility of components regardless of screen size and makes them dynamically respond to changing window dimensions.

By default, dynamic sizing is true for components that feature no sizing of its own and are not placed into any sized HTML container. Such components take space of the entire screen and are resized as the browser window resizes.

To enable dynamic sizing in other situations, take the following steps:

- create div container and give **relative dimensions** to it;
- put a Webix component into it (define the **container** property within the component);
- ensure that the component is **adjusted** to its container each time you resize browser window.

```
<div id="testA" style='width:50%; height:50%;'></div>
...
grid = new webix.ui({
    container:"testA",
    view:"datatable",
    ...
});

webix.event(window, "resize", function(){ grid.adjust(); })
```


Note that events to HTML nodes are attached via event() method rather than attachEvent().

Related Articles:

- Event Handling
- Component Resizing

Intermediate

Resizing

The component can be resized in two ways:

- Setting new dimensions to the component itself and applying **resize().**
- Setting new dimensions to the component container and applying **adjust()** to the component.

On-page resizing with resizer lines and responsive behavior on window resizing is discussed separately.

Resizing the Component

The component is resized by changing values of its width and height properties.

The properties can be changed in two ways:

- with the help of **define()** method:

```
Resize the dataview from 500px to 700px in width

dtable = new webix.ui({
    view:"datatable",
    width: 500,
    ...
})

dtable.define("width", 700);
dtable.resize();

//or both width and height
dtable.define({
    width:700,
    height:500
});
dtable.resize();
```

RELATED SAMPLE: Dynamical adjusting to the parent container

- by getting directly to the component **config**:

```
Changing dataview width and height

$$("dview").config.width = 261;
```

```
$$("dview").config.height = 700;
$$('dview').resize();
```

RELATED SAMPLE: Resizing

Resizing the Component's Container

Container is an HTML node you place your component into.

When you resize the container size and (optionally) call **adjust()** method for the component in it - component will change its dimensions to match with the container new size.

```
Placing datatable in testA container

<div id="testA" style="width:500px;"></div>
...
grid = new webix.ui({
    view:"datatable",
    container:"testA",
    ....
});
```

Container can be resized in two ways:

- via **DOM reference**

```
document.getElementById('testA').style.height="455px";
grid.adjust();
```

RELATED SAMPLE: Resizing

- using webix **toNode()** method:

```
webix.toNode("testA").style.width = "700px";
grid.adjust();
```

RELATED SAMPLE: Dynamical adjusting to the parent container

Component can be as well adjusted to the dynamically-sized container.

Resizing Several Component at a Time

If you need to resize several components lying inside one and the same parent component (like dataviews, lists, datatables inside one and the same layout with its own ID, you can redefine their dimensions, call the **resize()** function from the parent and pass the *true* parameter to it.

```
webix.ui({
```

```
    view:"layout", //optional
    rows:[
        {view:"dataview"},
        {view:"list"},
        {view:"datatable"}
    ]
});


$$("dataview1").define("width", 300);
$$("list1").define("width", 350);
$$("datatable1").define("width", 420);

$$(layout1).resize(true);
```

For altering dimensions of the component children use the corresponding method:

```
$$('accordion').resizeChildren();
```

Study the Sizing Components article to learn about initial component sizing.

Intermediate

Scroll Control

Webix provides extensive scrolling management tools.

- Firstly, Webix components are supplied with **vertical** and **horizontal scrollbars** that can be shown and hidden programmatically;
- Secondly, scrolling in the application can be controlled with the help of **common methods** and **properties**;
- Thirdly, Webix library features a separate component called **scrollview**, a container view with scrollbars for placing any content there (plain text, HTML, Webix views);

By default, Webix uses **native** browser scrollbars. In **Webix Pro** edition custom Webix-made scrollbars are available.

Tuning Scrolling

By default, component features vertical scrolling. It can be modified via the dedicated **scroll** property that may take the following values:

- "x" - sets a horizontal scrollbar;
- "y" - sets vertical scrollbar (default);
- "xy" - sets both horizontal and vertical scrollbars;
- false - defines non-scrollbale component.

```
webix.ui({
    view:"list",
    id:"mylist",
    scroll:"x", //"y", "xy", false
    ..//config
});
```

Some components feature such properties as **scrollX** and **scrollY** that take boolean values to enable/disable the specified scrolling direction, so check a component's API Reference.

Scrolling and Dynamic Loading

With dynamic loading, only part of the stored records are loaded into the component during init. Each time you scroll down/up the component, a data request is sent to server to load the next portion of data.

Read more about dynamic loading.

Scrolling to a definite position in the view

If the dataset if too long (e.g. a collection of verses) it seems nice to have an ability to scroll to a definite poem automatically with a click of a button.

Here you have several possibilities:

1 . Make use of the component's **scrollTo(x,y)** method:

The function is called from a scrollview object and takes horizontal and vertical position of the necessary line into it. If you state this position as button's ID, the function will look like this:

```
function scroll(id){
    id=this.config.id * 1; //gets ID of the clicked button
    $$("verses").scrollTo(, id); //scrolls through the view with the
specified ID
}
```

Scroll position can be calculated during development state with the **getScrollState()** method that returns an object with X and Y parameters.

You can as well as set **time** for the **scrollTo()** function to perform.

```
webix.ui({
    view:"mylist",
    scrollSpeed:"100ms",
    ...
})
```

2 .Scrolling via **focusing** on the necessary **view** withtin the scrollview.

Here the button's ID should be connected to the ID of the row in the scrollview:

```
{ view:"button", value: '..', id: '1', click:"scroll"},
{ view:"button", value: '..', id: '2', click:"scroll"},
```

Then the **showView()** set focus to the scrollview item with the corresponding **ID**.

```
webix.ui({
    view:"scrollview",
    id:"verses",
    body:{
       rows:[
           { id:"verse_1", template:".."},
           { id:"verse_2", template:".."},
           { id:"verse_3", template:".."}
       ]
```

```
});

 function scroll(id){
            $$("verses").showView("verse_"+id);
        }
```

RELATED SAMPLE: Scrollview

3 . Make the view **show the definite item** specified by its ID:

```
$$("mylist").showItem(5); // the list should be scrollable
```

RELATED SAMPLE: Horizontal List

Within datatable scrolling via showing includes the following methods:

- **showItem**(*rowId*) - scrolls to a definite row indicated by its ID;
- **showItemByIndex**(*rowIndex*) - scrolls to a definite row dfined by data item index in the datasource;
- **showCell**(*rowID, cellID*) - scrolls to a definite cell (both vertical and horizontal scrolling modes work).

Defining scroll position

Scroll state is defined as the combination of a **top** and **left scroll position** (how much is the component scrolled from its top and left border). In case you have either a horizontal or vertical scrollbar, the scroll state includes only one value - X or Y respectively.

Scroll position can be derived with the **getScrollState()** method to learn the current values:

```
var scPos = $$("mylist").getScrollState(); // returns data as
({x:30,y:200})
var posx = scPos.x; // 30
var posy = scPos.y; // 200
```

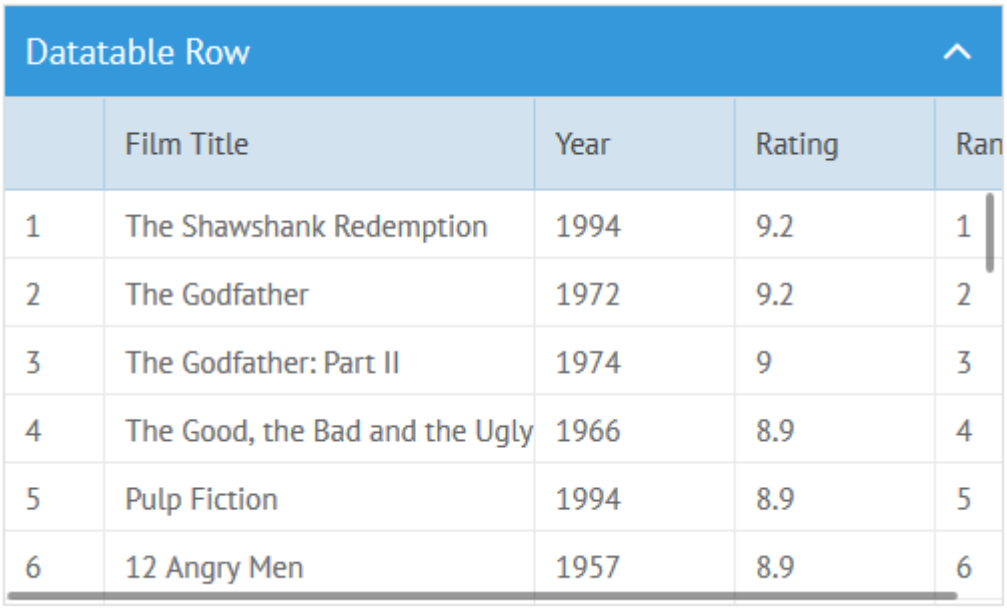
Webix Custom Scroll



The feature is available in **Webix Pro** edition only.

Webix offers custom scrollbars to replace native browser ones. The advantages are as follows:

- semi-transparent thin grey bars feature neat design;
- the bars are initially and hidden appear only when scrolling movement is detected on the page (provided scrolling is enabled).



RELATED SAMPLE: Custom Scroll

The feature is provided by a separate **CustomScroll** module that you need to **enable** before use.

Make sure you wrap everything into a **webix.ready()** function that is executed after page loading:

```
webix.ready(function(){
    //enabling CustomScroll
    if (!webix.env.touch && webix.ui.scrollSize)
        webix.CustomScroll.init();

    //your webix app
    webix.ui({
        ...
    });
});
```

Intermediate

Date and Number Localization

Different countries and regions have their own rules for writing dates, numbers and monetary units. All these issues should be taken in account in case a target group for your app is wider than one country or region.

The set of rules for this or that country is called **locale** and includes methods for date, time, number and price formatting. Localization methods belong to the i18n class and treat data according to he set format.

By default **Date()** constructor outputs raw data. Unformatted, it looks barely readable.

```
new Date(2012,10,30); //-->Tue Nov 30 2010 00:00:00
```

Date and time should be formatted either with Date or **i18n** class.

Numbers differ by delimiter before the fractional part and between thousands while prices should be rendered with an appropriate currency mark. Formatting is done either in the Number or **i18n** class.

Built-in Locales

The **standard package** of the Webix library includes 9 locales, namely:

- **'en-US'** - North American (used by default);
- **'ru-RU'** - Russian;
- **'fr-FR'** - French;
- **'ja-JP'** - Japanise;
- **'be-BY'** - Belorussian;
- **'de-DE'** - German;
- **'es-ES'** - Spanish;
- **'it-IT'** - Italian;
- **'zh-CN'** - Chinese.

Webix Pro edition includes over **300 locales** to match a great variety of cultures.

Locale Structure

The way formatted date is presented depends on **locale** you use. Locale is a set of rules for dates, numbers and price units for this or that region or country. By default **en-US** locale is used.

```
en-US Locale

webix.i18n.locales["en-US"]={
  groupDelimiter:",",
```

```
  groupSize:3,
  decimalDelimiter:".",
  decimalSize:2,

  dateFormat:"%m/%d/%Y",
  timeFormat:"%h:%i %A",
  longDateFormat:"%d %F %Y",
  fullDateFormat:"%m/%d/%Y %h:%i %A",

  price:"${obj}",
  priceSettings:{
     groupDelimiter:",",
     groupSize:3,
     decimalDelimiter:".",
     decimalSize:2
  },
  fileSize: ["b","Kb","Mb","Gb","Tb","Pb","Eb"],

  calendar: {
    monthFull:["January", "February", "March", "April", "May", "June",
        "July", "August", "September", "October", "November", "December"
    ],
    monthShort:["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug",
        "Sep", "Oct", "Nov", "Dec"
    ],
    dayFull:["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
        "Friday", "Saturday"
    ],
    dayShort:["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"],
    hours: "Hours",
    minutes: "Minutes",
    done:"Done",
    clear: "Clear",
    today: "Today"
  },

  controls:{
    select:"Select"
  }
};
```

Numbers:

- *groupDelimiter* (string) - a stop mark to divide number groups;
- *groupSize* (number) - the number of digits in each group. Typically it's 3 since thousands are the most frequent group;
- *decimalDelimiter* - a stop mark to divide even number from fractional part;
- *decimalSize* (number) - the number of digits in the fractional part;

Date and Time:

- *dateFormat, longDateFormat, fullDateFormat* - define the way day, month and year look in stated modes;
- *timeFormat* - defines the way hours, minutes and seconds look.

Start Date:

By default, a week in the calendar starts on Sunday. However, you can set any day as a start day using the **webix.Date.startOnMonday** property

```
webix.Date.startOnMonday = true;
```

Price states monetary unit applied to the data from data object (**{obj}**).

Calendar:

- *monthFull, dayFull* (string) - an array of full and short month names for this location;
- *monthShort, dayShort* (string) - an array of full and short day names for this location.

When formatted to a short/long date, the date from the first snippet looks like this:

```
Formatted with en-US locale

format:webix.i18n.dateFormatStr
    //--> 11/30/2010

format:webix.i18n.longDateFormatSt
    // --> 30 November 2010
```

- *hours, minutes* - localization options for 'Hours', 'Minutes', 'Done' labels in the calendar.
- *today, clear* - configuration for "Today" and "Clear" buttons, used for selection of current date and removing selection

Controls

Text labels for some parts of webix components that are more likely to be changed in a scope:

- **select** - text label of the 'Select' button of the multisuggest (used in multiselect input, editor and filter). The default value for North Americal locale is "Select".

Applying a Locale

Locales other than **en-US** should be set manually. It can be done with the help of a **setLocale()** function that takes **locale name** as parameter.

```
webix.i18n.setLocale("fr-FR");
```

After this any method applied to the **webix.i18n** object will redraw date, and numbers according to the rules described in locale.

```
Formatted with fr-FR locale

format:webix.i18n.dateFormatStr
    // -->30/11/2010

format:webix.i18n.longDateFormatStr
    // --> 30 Novembre 2010
```

Altering a Locale

To change one or several parameters for the current locale, you need to redefine them and then apply the same locale so that your changes come into force:

```
webix.i18n.fullDateFormat = "%Y-%m-%d %H:%i";
webix.i18n.setLocale(); //locale name is not specified here
```

Setting Localized i18n Format

Localization is done via the i18n class and allows setting customized formats gathered under one and the same locale.

In case of datatable, format is stated by the dedicated **format** parameter while data is specified by column ID:

```
{ header:"Date",  id:"start", format:webix.i18n.dateFormatStr},
{ header:"Price", id:"price", format:webix.i18n.priceFormat}
```

RELATED SAMPLE: Internationalization

Other components like list, dataview and Tree have quite the other way of data presenting. To ensure proper formatting, you should make use of a **template function:**

```
template:function(obj){
    return webix.i18n.longDateFormatStr(obj.start)+"<br/>"+
       webix.i18n.priceFormat(obj.price);
}
```

The format in use is defined by current locale. EN-US (default) locale sets **dateFormatStr** as "*%m/%d/%Y*" while **price** format is set as "*{obj}currency*" where *obj* is price value.

RELATED SAMPLE: Dataview Formatting

Managing Data in a Component

Here you learn how to present and process data with the help of dedicated components differing in look and feel yet similar in functionality. They are as follows:

Chart, Datatable, Dataview, Grouplist, List, Property Sheet, Tree, Treetable, Unitlist

- Data Object Operations
- Selection
- Adding/Deleting Data Items
- Data Templates
- Working with Dates
- Data Filtering and Sorting
- Data Validation
- Editing Data Items
- Data editors
- Paging
- Drag-n-Drop within Webix Views
- Working with the clipboard
- Grouping Data Items

Beginner

Data Object Operations

Every UI component features inner storage for loaded data:

- DataStore - non-hierarchical (inline) store for list, dataview, datatable, chart;
- Treestore - hierarchical store for tree, treetable and grouplist provided that they are loaded with **hierarchical** XML or JSON.

Both of the stores feature common methods, properties and events to work with data items on client side while TreeStore offers extended API for treating data items observing hierarchy.

While **manupulating data** items, you can:

- apply methods directly to the **Data/TreeStore** of the component- *$$("list").data*. Only some functions require this.
- apply methods to the component by its **ID** - *$$("list")*, which implies addressing datastore. Most methods have this pattern.

```
$$("list").data.each(){function(obj){.....} }; //iterates through items array

$$("list").count(); //counts items in the list datastore
```

Each **data item object** posesses the unique **ID**, **index** in the data array and set of **properties**.

Getting the item object

To get the data item object, you should use the getItem method:

```
var mylist = webix.ui({
    view:"list",
    ...
    data:[
        {id:"book1", title:"The Shawshank Redemption", year:"1994"},
        {id:"book2", title:"The Godfather",            year:"1972"}
    ]
});

//'item2' variable will contain an object of the related list record
var item2 = mylist.getItem("book2");

//you can access data members directly
var title = item2.title;//-> "The Godfather"
```

Getting the item id

Each data item has its ID defined in the dataset. Knowing item ID you can get to all its properties through an item object (described above).

Table 1 Method(s) for getting the item id

Method	Description
id (*object*)	gets the ID of a data item
getFirstId()	gets the ID of the first data item (with the "zero" index)
getLastId()	gets the ID of the last data item
getNextId (*id*)	gets the ID of a data item that succeeds the one with the specified ID
getPrevId (*id*)	gets the ID of a data item that precedes the one with the specified ID
getIdByIndex (*index*)	gets the item ID by its index
getRange (*fromId, toId*)	gets an array of IDs in a predefined scope
getIndexRange (*fromInd, toInd*)	gets an array of IDs in a predefined scope
getFirstChildId (*id*)	gets ID of the first child of the specified item in the hierarchical treestore
getParentId (*id*)	gets ID of the parent of the specified item in the hierarchical treestore
getNextSibling (*id*)	gets ID of the next sibling of the specified item in the hierarchical treestore
getPrevSibling (*id*)	gets ID of the previous sibling of the specified item in the hierarchical treestore

Getting the item index

Each data item defined in a dataset has an index - the position in the dataset starting from 0.

Table 2 Method(s) for getting the item index

Method	Description
getIndexById (*id*)	gets the item index by its ID
getBranchIndex (*id*)	gets item index in a specific branch

Checking the existence of an item

To check whether a specific item exists in the data set, you should use the exists method:

```
var mylist = webix.ui({
    view:"list",
    ...
    data:[
        {id:"book1", title:"The Shawshank Redemption", year:"1994"},
        {id:"book2", title:"The Godfather",            year:"1972"}
    ]
```

```
});

mylist.exists("book2");// -> true
mylist.exists("book222");// -> false
```

Changing the item id

To change the current id of an item, you should use the changeId method:

```
var mylist = webix.ui({
    view:"list",
    ...
    data:[
        {id:"book1", title:"The Shawshank Redemption", year:"1994"},
        {id:"book2", title:"The Godfather",            year:"1972"}
    ]
});

mylist.data.changeId("book1", "book01");
```

Changing data properties of an item

To change the current value of some item property, you should use the following technique:

```
item2 = mylist.getItem("book2");
item2.title = "Star Wars";

mylist.refresh();
//or
mylist.updateItem("book2", item2);
```

Methods refresh and updateItem lead to one and the same result and don't have any specificity.

Manipulations with items in the data set

Table 2 Method(s) and properties for manipulating items in the data set

Method	Description
count()	returns the count of items in the data set
getVisibleCount()	(just for the List component) returns the number of items that can be seen with the current view height
data.each()	iterates through the collection of data items
data.order	returns an array of IDs of the currently displayable items
data.pull	returns the hash of all data items

Method	Description
data.eachChild(id)	returns the hash of all children (first-level sub items) of the specified item in the hierarchical dataset
data.eachSubItem(id)	returns the hash of all subitems (regardless of level) of the specified item in the hierarchical dataset
data.eachOpen()	returns the hash of all open data items in the hierarchical dataset

Unlike **order**, the **pull** property returns all the items whether they are displayed in the component or not.

```
var mylist = webix.ui({
    view:"list",
    ...
    data:[
        {id:"book1", title:"The Shawshank Redemption", year:"1994"},
        {id:"book2", title:"The Godfather",            year:"1972"}
    ]
});
mylist.data.pull["book2"];//-> {id:"book2", title:"The Godfather",
year:"1972"}

mylist.data.pull["book2"].title;//-> "The Godfather"

mylist.data.pull;
//- >[{id:"book1", title:"The Shawshank Redemption", year:"1994"},
//    {id:"book2", title:"The Godfather",            year:"1972"}]
```

Iterating data items

To iterate through the collection of data items, you should use the each method of the DataStore class.

```
var mylist = webix.ui({
    view:"list",
    ...
    data:[
        {id:"book1", title:"The Shawshank Redemption", year:"1994"},
        {id:"book2", title:"The Godfather",            year:"1972"}
    ]
});

var titles =[];
mylist.data.each(function(obj){ titles.push(obj.title); });
//-> titles=["The Shawshank Redemption", "The Godfather"]
```

The code is equal to a standard **for-loop**:

```
for(i = ; i<data.length; i++ ){
```

```
    titles.push(obj.title);
}
```

Moving items within the data set(s)

Basically, here we speak about changing item position in the data set. Item position is its **index** (zero-based numbering).

Table 2 Method for moving items within the data set

Method	Description
move(*sid, tindex, tobj, tid*)	moves the specified item to a new position
moveBottom (*id*)	moves the specified item to the last position
moveTop (*id*)	moves the specified item to the first position
moveDown (*id, step*)	increases the item index and moves the item to the new position
moveUp (*id, step*)	decreases the item index and moves the item to the new position

Moreover, the move method makes it possible to programmatically drag an item from one component instance to another (from one list to another list):

```
//moves item with id="book1" to the 2nd pos in the same list (remember
about 0-based numbering)
mylist.move("book1", 1);

//moves the item with id="book1" to the first position in the "mylist2"
var id = mylist.move("book1", , mylist2);
```

Beginner

Selection

Built-in Selection

Single-Item Selection

To enable selection you should include **select** property into the component constructor and set its value to *true* (or 1).



```
webix.ui({
       view:"dataview",
       select:1, //enables select mode
       ...//config
   });
```

RELATED SAMPLE: Selection

The items will be selected on mouse click.

Selection of Multiple Items

- To enable selection of several items clicked in succession with **'Ctrl'** key pressed, you should switch the component to the **multiselect** mode.
- To enable multiselection by clicking items (or tapping them on **touch** devices) you should additionally switch to **"touch"** variation of multiselection.

Dataview, List and Tree Multiselection

```
"Ctrl"+click mutliselection

webix.ui({
    view:"list",
    select:"multiselect",
    ..// config
})
```

```
"Touch" mutliselection

webix.ui({
    view:"list",
    select:"multiselect",
    multiselect:"touch"
    ..// config
```

```
})
```

Datatable and Treetable Multiselection

```
"Ctrl"+click mutliselection

webix.ui({
    view:"datatable",
    multiselect:true,
    select:'cell', //multiple seelction of cells
    ...// config
})
```

```
"Touch" mutliselection

webix.ui({
    view:"datatable",
    multiselect:"touch",
    select:'cell',
    ...// config
})
```

Moreover, components that present hierarchical data (Tree and treetable) feature the **level** multiselection pattern. If set, it allows multiple selection of items within one and the same tree branch, the items belonging to one and the same hierarchy level.

```
webix.ui({
    view:"tree",
    select:true,
    multiselect:"level", // in addition to the enabled selection
    ....//tree config

})
```

RELATED SAMPLE: Multiline Selection

Datatable and treetable feature more selection modes that are described in the datatable documentation.

Selection methods

You can select items with the help of selection methods triggered by various events;

- **select**(*IDs, [non_modal, continue]*) function selects the specified items. Check selection API for details.

If you don't pass any parameter into the function, all items will be selected.

You can select the first and the last item from the dataset without specifying their IDs. In this case, you should additionally apply **first()** and **last()** methods to the component.

```
$$('datalist').select($$('datalist').getFirstId();) //selects the first
data item in the list

$$('dataview').select(4); //the cell containing item with ID = 4 will be
selected
$$('dataview').select([2,3,4]); the items with IDs equalling to 2, 3 and 4
```

- **selectAll()** function equalls to select(); with no parameters and highlights all the items at once;
- **unselect()** and **unselectAll()** - the functions work similar to the above mentioned and deselect items according to the same principles;
- **isSelected(int)** - the function checks whether the item with this ID is selected. Returns *true* or *false*;
- **getSelectedId()** - the function doesn't take any parameters. It returns the ID of a selected item or an array of IDs in case of multiselect mode.

Keyboard navigation

Keyboard navigation can be used for the component with enabled selection.

```
webix.ui({
    view:"list",
    select:true,
    navigation:true
});
```

RELATED SAMPLE: Navigation in DataTable

Note than not all data components support navigation by default. Then **KeysNavigation** module should be added manually.

```
webix.ui({
    view:'dataview', id:"dataview1", select:true, navigation:true
});

webix.extend($$('dataview1'), webix.KeysNavigation);
```

The code above adds navigation only to this instance of a Dataview, still it's possible to provide the functionality for all dataviews in the app.
For more details, refer to Component Extending article.

Intermediate

Adding/Deleting Data Items

Here you should make use of **add()** and **remove()** methods included into your custom functions.

These methods are true for chart, datatable, list and dataview components. For Tree and treetable look up their documentation.

Adding and deletion is implemented in the form of event taking place on user action. Normally, it is done on clicking the button with a corresponding label.

Basic Adding and Deleting

Basic syntax here looks as follows:

```
$$("myview").add({
         id:"item_id",  //adding  an item with two properties
         property:"some_value"
})
$$("myview").remove("item_id"); //to delete a single item

$$("myview").remove(["idA","idB","idC","idD"]); //to delete multiple items
```

Adding User Data from Form

To add an item which is a string entered by user in a form, you need to derive this string from the input area and then add data to the desired component. Again, specify everything by their ID. It's a common rule for all methods and event handlers.

Form and HtmlForm components have different code for input fields - look for the library's text control and standard html input respectively. It means that different methods are used to get data from them:

```
Adding Data from JS Form

function addData(){
    $$("data_1").add({
        user:$$('form1').getValues().user,
        email:$$('form1').getValues().email,
    }, );
} //'user' and 'mail' are ID-s of dedicated text fields
```

```
Adding data from [HtmlForm]
```

```
function addData() {
    $$("data_2").add({
        title: document.getElementById("title").value, //data users enter
into an input field
        year: document.getElementById("year").value
    },)
}
```

Deletion of Selected Items

Since only selected items can be deleted, set the select parameter to true (**select: true**) during the initialization of the component. Before removal let's check whether an item is selected with the help of the simpliest *if-else* statement.

Deletion of one item

```
function removeData(){
    if(!$$("data").getSelectedId()){
    webix.message("No item is selected!");
    return; //'data' is an ID of the needed component
}
$$("data").remove($$("data").getSelectedId());
}}
```

RELATED SAMPLE: Adding/Removing Items

Deletion of several items

In **multiselect** mode several items from the dataset can be selected. To remove all selected items, make use of a *for-loop*:

```
function remove_item(){
    var list = $$('mylist');
    var sel = list.getSelectedId(true);
    if (!sel) return;
    for (var i = ; i < sel.length; i++)
    list.remove(sel[i]);
};
```

All the items can be removed from the component with the help of the following code:

```
$$("mylist").clearAll();
```


The above described methods are as well true for database-connected components. Learn more about server-side integration here .

Intermediate

Data Templates

Here we speak about the ways of presenting data inside the component after data loading.

After the data is loaded to the component it is stored in the `data` object and the component searches for the right value to display. So what is the **object value** for each data item?

Data Item Value

All supported data types may contain **more than one value per item** e.g. each item of a filmset may contain not only a film's title but also its rank, production year, and number of votes.

Still, different components have different possibilities and requirements:

- Components with **single-value items**: list and its variants (grouplist, unitlist and x-list), dataview and tree. Each of their items takes one value by default.
- Components with **multiple-value items**: datatable and treetable. They take as many items as many columns are defined.
- Chart requires a **pair of values** - one for each exle.

Item Value can be defined by:

- **JSON property/XML tag value**

```
//JSON
[ {id:1, value:"3", title:"" }, {id:2, value:"2", title:""} ]

//XML
<data><item value='3' other:""/><item value='2' other:""/></data>
```

RELATED SAMPLE: Tree: JSON Dataset

Such definition would suit single-value components. If a data item has other properties, their values won't be shown unless you define a template.

- **data template** that performs two tasks
  - main - defines which item property (-ies) should be treated as **value** to display;
  - adds formatting to this value.

Templates can be defined for any component you wish.

Templates can set one or several properties as the value to display:

```
webix.ui({
```

```
    view: "list",
    template: "#title#"
}); // this list will show 'titles'

webix.ui({
    view: "list",
    template: "#value#.#title#"
}); // this list will show 'values' and 'titles'
```

RELATED SAMPLE: List

- datatable and treetable **column ID** that act as templates for cells.

```
view: "datatable",
columns: [
    { id: "id", header: "...", width: 50}, // 1st col displays item 'IDs'
    { id: "title" }, // 2nd col displays item 'titles'
    { id: "value" }  // 3rd col displays item 'values'
]
```

Column IDs perform the main template function - they set the value to display. To add extra words or formatting to the displayed values, you can define a template for the column.

```
{ id: "title", template: "The title is <b>#title#</b>!" }
```

Template for CSV and JSArray

CSV and JSArray data types have no tags or properties.

```
//CSV
'3,4 \n 2,6 \n 5, 9'

//JSarray
[
    [3, 4],[2, 6],[5, 9]
]
```

The elements are defined by their position in the CSV string or JSArray - data0, data1, data3

```
//list template
template:"#data0#.#data1#"

//datatable columns

columns: [
    { id: "data0" },
    { id: "data1" }
];
```

RELATED SAMPLE: Loading from inline datasource

As far as you've noticed, only **JSON** and **XML** data can be displayed without refering to template, since only they can directly set **value.** Moreover, this can be applied only to components with **single-value items**.

Chart exception

Chart requires two values per item - one for each axis.

- **Y-axle value** is defined by the **value** property that can be additionally amended within the `yAxis` object property.
- **X-axis value** is defined by the template within the `xAxis` object property.

```
webix.ui({
    view:"chart",
    value:"#data0#", //in case of CSV or JSArray
    xAxis:{
        template:"#data1#"
    },
    yAxis:{
        template:function(obj){
              return (obj%20?"":obj
     }
});
```

In case of template function (described below) value is referred to as **obj** instead of standard obj.value (obj.title, obj.data0).

RELATED SAMPLE: Chart: CSV Dataset

Template Contents

The whole template comes in quotation marks and includes:

- **JSON keys / XML tags** from the initial dataset, surrounded by hash signs;
- **HTML tags** and **CSS selectors** (optional, adds styling);
- **plain text**.

Template Sample

```
template: "<span style='classname'>#title#</span>. Shot in #year#. Ranked
#rank#."
```

Title, *year* and *rank* signify repeated values from the dataset. All the titles will be styled with the help of the *.classname* CSS class. 'Shot in' and 'Ranked' are plain text to be shown to the user.

Optionally, you can define the **template for data copied to clipboard**. If not specified, data will be taken according to the above-mentioned template.

```
templateCopy: "#rank#.#title#"
```

In this case, only the `rank` and `title` values are copied.

Grouplist has its own templating pattern. Look into the related article for details.

The item in the intial dataset looks like this:

```
//JSON
{ title: "My film", year: 1994, rank: 7 }

//XML
<item>
    <title>My film</title>
    <year>1994</year>
    <rank>7</rank>
</item>
```

Inside the component it looks like this: ***My film.*** **Shot in 1998. Ranked 7.**

Don't confuse template property with the template component.

Template Types

Templates can be defined in five very flexible ways:

- as HTML strings within JavaScript objects
- by creating named templates
- by extracting them from an HTML container
- by retrieving them from an external file
- as a function returning a string

Templates are included into Webix components as well as into some of their properties that take objects as values.

- Tooltip implementation
- Type implementation

HTML inside JavaScript

```
JSON data in the dataview

var big_film_set = [
    { id: 1, title: "The Shawshank Redemption",
      year: 1994, votes: "67,879",
      rating: "9.2", rank: 1
    },
    { ...}
];

webix.ui({
  view: "dataview",
  template: "<div class='rank'>#rank#.</div><div
class='title'>#title#</div>"
});
```

RELATED SAMPLE: Template as a String

Named Template

Templates may be included inside the **type** parameter. Type applies to each item rather then the whole component.

1 . First you should create a type with the `webix.type` constructor. It is created for the specified UI component:

```
webix.type(webix.ui.dataview, {
    name: "typeA",
    template: "<div class=''>#rank#.</div>"+
            "<div class='title'>#title#</div>"+
            "<div class='year'>#year# year</div>"
});
```

2 . Then you specify the name of that type as value of `type` property for this component:

```
webix.ui({
    view: "dataview",
    type: "typeA",  // name of the webix.type object created in the first
step
    ...
});
```

RELATED SAMPLE: Named Templates

See the corresponding article to learn more about type implementation.

HTML Container with Data Template

The template may be defined inside an **html-container** and referred by its id:

```
<textarea id="template_container" rows="5" cols="60" style="margin-
left:20px">
   <div class='overall'>
        <div class='rank'>#rank#.</div>
        <div class='title'>#title#</div>
        <div class='year'>#year#</div>
   </div>
</textarea>

<script>
webix.ui({
    view: "dataview",
    // lookup the textarea by id and fetch its contents
    template: "html->template_container",
    ...
});
<script>
```

RELATED SAMPLE: Pure HTML Template

External HTML template

Templates can be defined in external files. The **template** parameter will be a link to this file, prefixed by **http->**:

```
webix.ui({
    view:"dataview",
    template:"http->template.html",
    data:big_film_set
});
```

The file contents (an HTML template from the example above):

```
<div class="overall">
    <div class="rank">#rank#.</div>
    <div class="title">#title#</div>
    <div class="year">#year# year</div>
</div>
```

Note that these HTTP requestes are not cached. If you use the same template repeatedly, it's better to retrieve it via AJAX and store it in a variable:

```
var objectDetailTemplate =
```

```
webix.ajax().sync().get('template.html').responseText;
```

RELATED SAMPLE: Pure HTML Template: Referring to External HTML

Template Function

`template` may as well be **function** that returns the string to render. No #-interpolation will take place in this case; you have to interpolate variables on your own.

```
...
template: function (obj) {
    return "<div class='overall'><div class='rank'>" + obj.rank +
".</div>" +
      "<div class='title'>" + obj.title + "</div>" +
      "<div class='year'>" + obj.year + " year</div> </div>";
}
```

RELATED SAMPLE: Template as a Function

- If value is defined by JSON/XML tag **value** - you refer to it as **obj.value.**
- If value is defined by template - you refer to it as **obj.title**, **obj.rank**, **obj.data0** (CSV, JSArray) etc.

XSS Safe templates

By default, `#...#` templates placeholders will be replaced with the data value without any processing. If you have HTML data in the data object, it will placed in the template. In most cases this is an advantage, but sometimes it may be necessary to output such data as text, not a HTML:

- when you have text data with special HTML characters (<, >, and &)
- when you are not sure is the data safe or not (e.g. was entered by the user)

In such cases you can add a `!` after the opening #:

```
{
  template: "My #data#" // will output data as HTML content
},
{
  template: "My #!data#" // will output data as Text content
},
```

Intermediate

Tooltip Implementation

Tooltip stands out of the fixed pattern. It lies above other ui-related components, which makes it similar to window and popup.

Tooltip is connected with some component or its item and becomes visible when you place a mouse pointer over it and disappears when it leaves the area. Doesn't it resemble onMouseOver/onMouseOut events? Still, event handling is quite the other story.



Here, you don't need any specific code to make it work. You should just include a tooltip into the necessary component abd specify the info to be shown inside a tooltip area.

Tooltip may include text as well as items from the dataset.

```
webix.ui({
    view:"dataview",
    ...
    tooltip:{
        template:"<span class='webix_strong'>Rating: </span> #rating#<br/>
        <span class='webix_strong'>Votes: </span> #votes#"
    }
});
```

RELATED SAMPLE: Template of Tooltips

By default, tooltip is placed on the same level with the mouse pointer, 20 px to the right from it.

To change this pattern, specify the coordinates relative to the mouse pointer as values of **dx** and **dy** properties.

```
tooltip:{
    template:"<span class='webix_strong'>Rating: </span> #rating#<br/>
    <span class='webix_strong'>Votes: </span> #votes#",
    dx: 10, //20 by default
    dy:5    // 0 by default
}
```

Datatable Tooltip

Datatable tooltip is defined is quite another way. Several steps are possible:

- Setting **tooltip:true** for the whole component. It enables tooltip functionality for the grid and shows tooltip for each column showing the row value that stands as ID for this column.

```
General datatable tooltip

webix.ui({
    view:"datatable",
    tooltip:true,
    columns:[
        {id:"name", header:"Name"},
        {id::"age", header:"Age"}
    ],
    data:[
        {id:1, name:"Ann", age:25},
        {id:2, name:"Tom", age:27}
    ]
});
//the tooltip for the first column of the first row is "Ann"
```

- **Customizing** tooltip data for each column. Tooltip should be switch on for the whole grid:

```
webix.ui({
    view:"datatable",
    tooltip:true,
    columns:[
        {id:"name", header:"Name", tooltip:"My name is #name#. I'm
#age#."},
        {id::"age", header:"Age"}
    ],
    data:[
        {id:1, name:"Ann", age:25},
        {id:2, name:"Tom", age:27}
    ]
});
//tooltip for the first column of the first row is "My name is Ann. I'm
25."
```

- Setting tooltip for datatable **header**. Tooltip should be switch on for the whole grid:

```
webix.ui({
    view:"datatable",
    tooltip:true,
    columns:[
        {id:"name", header:"<span title='My tooltip text'>Name</span>"},
        {id::"age", header:"Age"}
```

```
    ]
});
//tooltip for the Name column is "My tooltip text"
```

Advanced tooltip configuration

Instead of defining tooltip for each column separately, you can define tooltip once for all columns

```
webix.ui({
    view:"datatable",
    tooltip:function(obj, common){
        //obj - row object
        //common.column - configuraton of related column

        return "<i>"+obj[common.column.id]+"</i>";
    },
    columns:[
        {id:"name", header:"<span title='My tooltip text'>Name</span>"},
        {id::"age", header:"Age"}
    ]
});
```

Intermediate

Type Implementation

Type property allows treating each of the component's items separately - specify its **dimensions**, set **CSS** class, define data template and include elements that will be **common** for all the items.

Type is applicable to the following data components:

- dataview;
- list;
- grouplist;
- unitlist;
- menu;
- tree;
- datatable and treetable to define common properties for cells;

Webix **item type** is an object that is constructed of parameters you'd like to apply to each item of the component. These parameters can be native properties of Webix components or user-defined ones.

Native properties are described in the component API and may differ from component to component. They may include:

- **template** - the template of data visualization of each item;
- **templateStart** and **templateEnd** - wrappers for each data item, are not redefined, as a rule. More Info.
- **width** and **height** - width and height of each data item;
- **css** - item styling;
- **headerHeight** (unitlist specific) - the height of unit headers.

In the sample below it is shown how you size the dataview itself and how the dimensions of dataview cells are sized.

```
webix.ui({
    view: dataview,
    width: 550,
    data: ... ,
    type:{
        template: "<div class='rank'>#rank#.</div><div
class='title'>#title#</div>",
        width: 261,
        height: 90,
        css: "movie"
    }
});
```

RELATED SAMPLE: Template as a String

Note that the template set for the whole component will override the template defined within the component **type**.

```
// the "#rank#. #value#" template will be used
webix.ui({
    view: dataview,
    template: "#rank#. #value#",
    data: ... ,
    type:{
        template: "<div class='rank'>#rank#.</div><div
class='title'>#title#</div>",
    }
});
```

At the same time you can define the type separately by using **webix.type constructor** and then apply it to the necessary component:

```
webix.type(webix.ui.dataview,{
    name:"typeA",
    template: "<div class=''>#rank#.</div><div
class='title'>#title#</div>",
    width: 261,
    height: 90,
    css: "movie"
});

webix.ui({
    view: "dataview",
    type: "typeA", //name of the new object created in the first step
    ...
});
```

RELATED SAMPLE: Named Templates

Defining Common Elements

Common elements can be included in either of the two ways:

- **{common.property}** - allows including a constant value for all the items into the template:

```
webix.ui({
    view:"list",
    template:"{common.itemIcon} #rank#. #title#",
```

```
    type:{
        itemIcon:"<span class='webix-icon fa-film'></span>"
    }
});
```

- **{common.method()}** - allows including result of a function call into the template:

```
webix.ui({
    view:"list",
    template:"#rank#. #title# {common.itemYear()}",
    type:{
        itemYear:function(obj, common){
            if(obj.year>=2000){
                return "<span class='newtime'>"+obj.year+"</span>";
            } else if(obj.year<1970){
                return "<span class='oldtime'>"+obj.year+"</span>";
            } else
                return obj.year;
        }
    }
});
```

RELATED SAMPLE: List::Advanced Template

The **type** function takes item object as parameter, so all its properties are accessible in it.

Datatable and treetable already feature some common elements out of the box.

- Datatable built-in type templates;
- Treetable biult-in type templates.

Redefining default template elements

You can completely change the component's look and feel inside the **type** by altering HTML for the items. Type properties subject to altering are as follows:

- **template** - standard item template, can be used outside the **type** configuration with the same result. Includes data properties in #hash# signs and optionally, HTML content;
- **templateStart** - container element for data item, wraps the elements defined by **template**. Opening element;
- **templateEnd** - closing element for **templateStart**.

In other words, each data item HTML content includes *templateStart - template - templateEnd*. TemplateStart and templateEnd are not normally changed, but still the possibility exists.

This is how standard list can be redefined:




```
webix.ui({
   view:"list",
   type:{
      templateStart:"<div item_id='id' class='custom_item'>",
      template:"#rank#. #title#<br><div style='text-
align:right;'>#year#</div>",
      templateEnd:"</div>"
   }
});
```

RELATED SAMPLE: List::Advanced Template

Intermediate

Working with Dates

Dates with UI Components

- Loading and Formatting
- Editing
- Sorting
- Dates with Forms
- Date and Number Localization

Loading and Formatting

Dates are loaded to components according to common data loading rules.

Either in the database or in data files dates can be stored in two ways: as **DateTime** objects or as **strings**.

- DateTime objects should be formatted for adequate presentation. Otherwise, the output will be like "*Tue Nov 30 2010 00:00:00 GMT+0200*";
- Strings can be either shown in the way thay are stored or converted to objects and then formatted.

Only DateTime objects defined with the **new Date();** method are subject to formatting since it presupposes "to-string" conversion. Strings should be converted beforehand.

Pre-Formatting: String-to-Object Conversion

There exist two methods of date-to-string conversion:

1 . **Built-in (map property).** Datatable Only.

Conversion is provided by **map** property used for the column instead of column **ID.** Like ID, map refers to the necessary data item and allows for format specification:

```
webix.ui({
    view:"datatable",
    columns: [
        { map:"(date)#enddate#",editor:"date",header:"End date",
width:120}
    ],
    data: [
```

```
        {id:"1", startdate:"2013-09-26", enddate:"2011-05-15"},
        {..}
    ]

})
```

2 . **Custom** (using **webix.Date** object functions).

Applicable to all data management conponents including datatable.

Here we create the necessary conversion pattern and then define it as scheme for the loaded data.

```
var my_format = webix.Date.strToDate("%Y-%m-%d");
//hyphen-separated values are interpreted as year-month-date object

webix.ui({
    view:"datatable",
    ..//config
    scheme:{
            $init:function(obj){
                obj.startdate = my_format(obj.start);
            }
    }

})
```

RELATED SAMPLE: Converting strings to dates

Now we've got DateTime objects ready to formatting.

Defining Formatting Methods

There're two objects in the library that handle dates - **webix.Date** and **webix.i18n**. Both of them contain date formatting and processing logic while the later is responsible for date (as well as number and price) localization.

webix.i18n

Localization means **locales implementation** where locale is a collection of formatting methods and patterns for a certain area. They are defined separately and later applied to the necessady data. By default, if you format dates with the **i18n** object, all the dates and numbers will be formatted according to North American region rules.

Date and **Number localization** are described in the related article.

webix.Date

The Date object allows for formatting dates regardless of locales with the help of the **dateToSrt()** method. It requires format specification in its argument.

```
{ header:"m/d/Y", sort:"date", id:"start",
format:webix.Date.dateToStr("%m/%d/%y")}
```

The datatable column here shows the stored date in a "month number/day number/two-digit yeat" format.

Possible format specifiers are listed in the related article.

Applying Formatting Methods

Only datatable has a **format** property for its columns while other components require a specific **function template**.

```
Formatting Datatable Data

webix.ui({
    view:"datatable", columns:{
        [{ header:"WeekDay", id:"start",
format:webix.i18n.longdateFormatStr},
         {...}
        ]
})
```

```
Formatting Other Component's Data

//format is specified by locale
webix.ui({
    view:"list",
    template:function(obj){
            return webix.i18n.longDateFormatStr(obj.start);
                }
})

//format is specified separately
var myformat = webix.Date.dateToStr("%m/%d/%y");

webix.ui({
    view:"list",
    template:function(obj){
            return myformat(obj.start);
                }
```

Editing

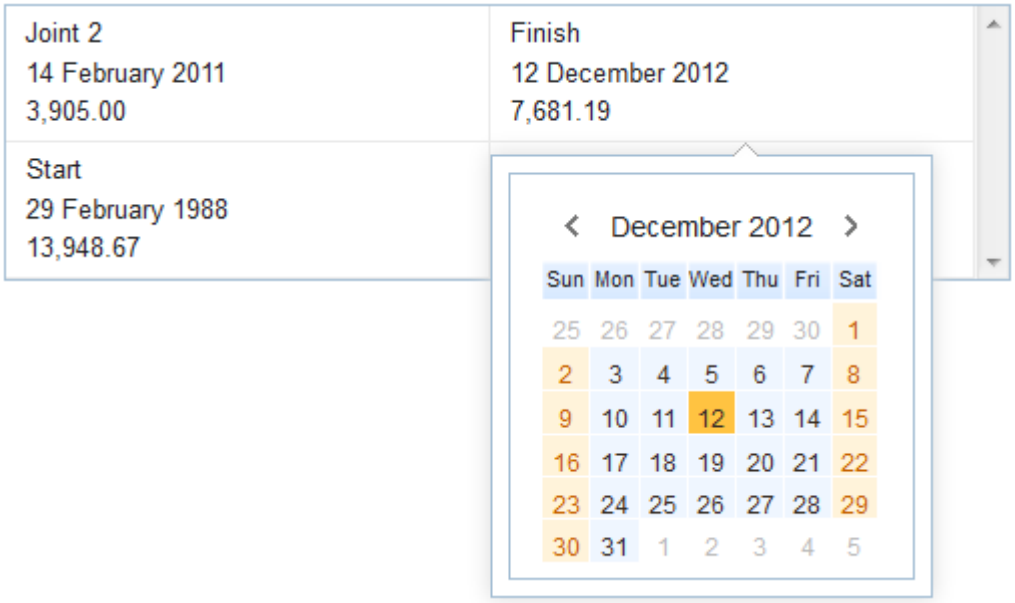
Dates within components are edited with the help of the dedicated **editor** (see Data Editors for details).

Requirements for Date editing:

- component should be **editable.**
- **editor type** should be defined. Here we need **"date"**;

- **editValue** (data item subject to editing) should be specified (excluding datatable). The thing is that components may have more than one data item in a cell.
- optionally **editaction** is defined. (*click* by default).

This editor is used as datepicker to call UI calendar on defined edit action to pick the necessary date.




```
webix.ui({
    view:"dataview",
    ..//config..
    editable:true,
    editor:"date",
    editValue:"start", //refers to data item
    data:[
            { text:"Joint 2", start:new Date(2011,1,14)},{...}]
})
```

RELATED SAMPLE: Dataview Formatting

When working with **server-side**, initialize a dataProcessor for the component and the changes will be automatically saved back to server.

Sorting

Though dates are presented as strings, they are **sorted as date objects**. So before sorting don't forget to convert strings to DateTime objects in case they are stores as strings (see above).

Sorting can be applied in two directions: **"asc"** (ascending) and **"desc"** (descending).

1 . **Built-in Sorting. Datatable Only.**

Datatable features dedicated **sort** property for its cloumns.

2 . **Custom Sorting.** Applicable to all components.

Here we make use of the **sort()** function that needs:

- template of the data item subject to sorting;
- sorting direction;
- sorting mode ("string", "string_strict", "int", "date").

```
HTML Button Triggers Sorting

<input type='button' value='Sort "Asc"' onclick='grid.sort("#start#",
"asc", "date");'>
```

RELATED SAMPLE: Using date templates

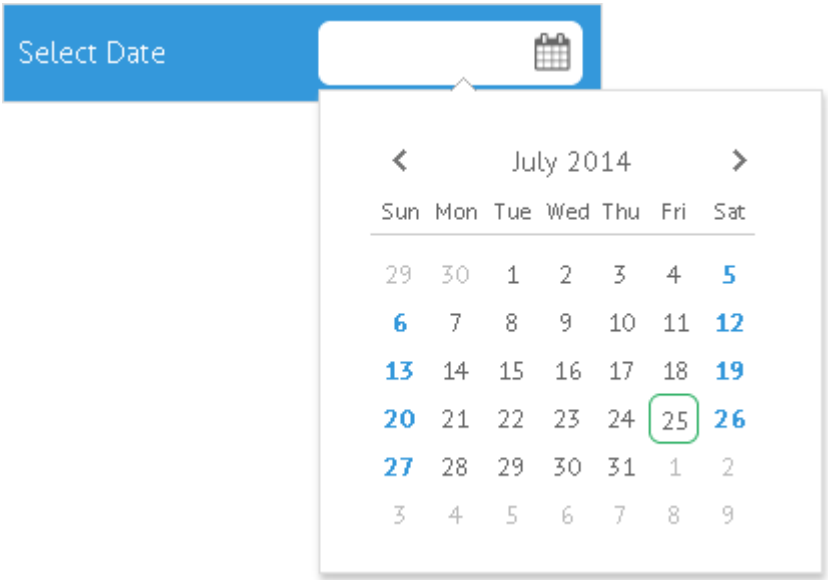
Data Sorting Api.

Dates Inputting in Forms

Form fields that require date inputting from users should be supplied with a calendar for user convenience.

There exists a special datepicker control that looks like a normal text field with a calendar icon.

When you click it a popup calendar appears. On selecting the necessary date, click somewhere outside the popup and the date will be shown in the text field. By default, date is shown formatted as "*%Y-%m-%d*" and stored as *DateTime* object.



RELATED SAMPLE: Date picker in Calendar

Customization

The calendar can be customized within the **webix.editors** object as well as localized according to common rules.

The default date for the control to show as well as the default date of the popup calendar is the current one. The pattern can be changed by setting the necessary value in the datepicker constructor:

```
{
    view:"datepicker",
    value:new Date(2011,1,12)
}
```

Default formatting can be changed as well. Formatting rules are described above and applied to the control as value of its **format** property.

```
{
    view:"datepicker",
    format: webix.Date.dateToStr("%d/%m/%y")
}
```

RELATED SAMPLE: Date picker in Calendar

If a form is bound to the component, it will load dates in the format specified by the component formatting pattern, or, if custom format is defined for datepicker, it will get the one from the latter.

Intermediate

Data Filtering and Sorting

Sometimes you need to view definite items rather then the whole dataset. Here you play around with the **filter()** method.

The task is as follows: you have a film set and want to filter it by title and year, with the possibility to enter the text criterion into the input area. At the same time filtration should start with the first letter while newly added letters would trigger data re-filtering.

Two major possibilities of data filtering are **content filtering and sorting** using built-in means and **custom filtering** where you should write your own function with the *filter()* method.

Content Filtering and Sorting

Available for datatable and treetable only. Filtering and sorting parameters are set for each column separately.

The **content** parameter set to '*textFilter*', '*selectFilter*', '*numberFilter*' or '*dateFilter*' value enables filtering. The former filters data as you type values into the input while the latter allows selecting from the predefined list

Data is sorted thanks to the **sort** parameter that may take '*int*', '*date*', '*string*' and '*string_strict*' (for case-sensitive filtering) values. You click headers of the dedicated columns to enable both ascending and descending sorting patterns.

```
webix.ui({
    view:"datatable",
    id:"films",
    columns:[
        { id:"title",   header:["Title", {content:"textFilter"}],
sort:"string"},
        { id:"year",    header:["Year",  {content:"selectFilter" ],
sort:"int"},
        { id:"votes",   header:"Votes",     sort:"int"}
    ],
    ...
});
```

RELATED SAMPLE: Filtering by several criteria (AND logic)

If you want to include an additional criterion to the built-in filter, use the **compare** property to define the additional filtering

Datatable is a peculiar component and has a number API peculiarities. That's why, refer to the corresponding

articles:

- Datatable Filtering
- Datatable Sorting

Treetable shares features of both datatable and Tree, that's it has similar fimilar sorting and filtering patterns. In addition, it borrows **filterMode** property from the tree API (described further).

Custom Filtering

Other components should be filtered with the dedicated **filter()** function that takes filtering value from the input area and shows only those data items that comply with it.

Filtering can be implemented in two ways:

Using data template

```
webix.ui({
    rows:[ //firstly define input area and data object
        {
            view:"toolbar",
            elements:[{view:"text",
            id:"list_input"]
        },
        {
            view:"list",
            id:"list",
            template:"#rank#. #title#",
            select:true,
            data:big_film_set
        }
    ]
});

$$("list_input").attachEvent("onTimedKeyPress",function(){
    var value = this.getValue().toLowerCase(); //input data is derived
    $$("list").filter(#title#, value);
}); //the titles are filtered
```

The same pattern is observed with components featuring similar data templates (dataview and Tree).
In the list above the template includes *rank* and *title* values for each film. More about data templates.

Using a function

Filter() method can have a function as argument. It returns data values that comply with input text:

```
$$("list_input").attachEvent("onTimedKeyPress",function(){
    var value = this.getValue().toLowerCase(); //here it gets user input
value
    $$("list").filter(function(obj){   //here it filters all titles from
the dataset
    return obj.title.toLowerCase().indexOf(value)==;
})
```

RELATED SAMPLE: List: Filtering

Peculiarities of Tree Filtering

```
Tree Filtering

//html input field
<input
    type='text'
    placeholder="type filter criteria here"
    style='width:250px; margin-left:20px'
    onkeypress="filter_tree()"
>

//filtering function
function filter_tree(){
    tree.filter("#value#", this.value);
}

// tree object
webix.ui({
    view:"tree",
    select:true,
    data: smalltreedata
});
```

The function performs filtering according to **#value#** values. It takes into account all the tree levels and displays results with the children of the filtered branches even if the children's values don't comply with the filtering criterion.

At the same time, you can change the filtering pattern with the help of the tree **filterMode** property with an object value.

```
{
    view:"tree",
    filterMode:{
        showSubItems:false,
        openParents:false,
        level:3},
    ...
}
```

- **showSubItems** (boolean) - defines whether the component should display children of the filtered items.
- **openParents** (boolean) - defines whether the component should expand branches to show the found items.
- **level** (number) - sets the hierarchy to filter tree nodes (one-based numbering). The '0' value says to filter all nodes.

RELATED SAMPLE: Filtering

RELATED SAMPLE: Filtering in TreeTable

Filtering by Multiple Properties

It's common knowledge that data management components can features more than one data template item per record. For instance, datatable record consists of several cells in different columns, dataview and list records contain several items, which is reflected in their data template.

You can filter data in the component referring to either of its template values using one and the same input field.

```
webix.ui({
    rows:[
        {view:"list", id:"dlist",
        template: "#rank#. #title#. #year#<br/>#category#",
        ..//list config
        },
        {view:"text", id:"filter-list" //input for filtering
        }
    ]
})


$$("filter-list").attachEvent("onTimedKeypress", function(){
    var text = this.getValue().toString().toLowerCase();
    //after text entering - filter related grid
    $$("dlist").filter(function(obj){
        //filter by multiple properties
        var filter = [obj.title, obj.category, obj.year].join("|");
        filter = filter.toString().toLowerCase();
        return (filter.indexOf(text) != -1);
    });
});
```

RELATED SAMPLE: Multiple Filter

Sorting

Sorting is enabled by **sort()** function that takes data item, sorting direction and sorting mode as parameters:

- Sorting as component property. Data will be during component initialization:

```
webix.ui({
    view: "list",
    template:"#title# - #year#"
    sort:{
        by:"#title#",
        dir:"asc",
        as:"string"
    }
});
```

- Sorting as a separate function. Data sorting will be triggered by any event you associate it with:

```
$$('chart').sort('#year#','desc');
$$('tree').sort('#value#', 'asc');
```

Read about more detailed description of the **sort** method in the dedicated chapter of API Reference

Intermediate

Data Validation

All the data from input fields must be validated before being sent to server. Data validation is implemented with dataprocessor, htmlform and form as well as with all the data management componens (datatable, dataview, tree, treetable, list).

Data validation is activated:

- Generally, validation is started on calling the **validate()** method. You can call it:
  - on **form submitting**;
  - on any **inner event** (*onChange* for form, *onAfterEditStop* for data component);
- With the **DataProcessor** object data is validated each time you save it to server. If validation fails, data isn't sent to server;
- With **data components** data is validated each time changes are made - editing, adding and updating (the moment *add()* and *update()* methods are called);
- By means of on-the-go HTML5 validation that works for form and htmlform.

While the first three are features of this library and work in conjunction with validation rules, the latter is a built-in HTML5 feature.

Validation by API

Data validation can be triggered on any event of any form element and validate any number of controls as well as the whole form:

Normally, form and htmlform, data validation is bound to a **submit** button with the help of **click** event handler:

```
{ view:"button", value: "Submit", click:function(){
    form.validate();
}}
```

At the same time, you can trigger validation whenever you like, e.g. on **onChange** event of a control:

```
{ view:"text", name: "fname", on:{
    "onChange":function(){
        //either validate the whole form
        this.getParentForm().validate();
        //or validate this element only
        this.validate();
```

```
    }}
}
```

Live Validation in Forms

Validation event can not only be attached to the control in a common way (as above) but by specific validateEvent property that allows for automatic attaching of two events:

- **"blur"** - to start validation on **onBlur** event, when focus is moved from the control;
- **"key"** - to start validation on **onKeyPress** event when a users enters a character into the field.

Note, that this method allows for defining validation event only for this control.

In case of data components, validation can be set on data loading to ensure that data came with right values:

```
webix.ui({
    view:"datatable",
    ..// config
    ready:function(){
        this.validate();}
});
```

RELATED SAMPLE: Datatable Validation

Partial Validation

With the **validate()** method you can validate:

- the whole form/component:

```
form.validate(); //all fields are validated
list.validate(); // data of the whole list is validated
```

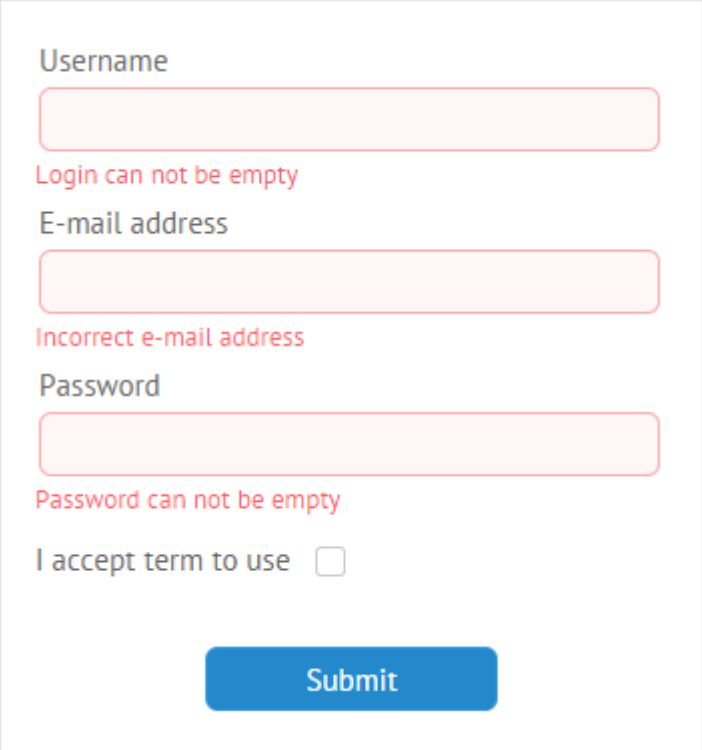
- specific component item;

```
list.validate(2); //data item with id=2 will be validated
text.validate(); //validate only this text field
```

Validation Message

If any record or field fails validation, it is marked with embedded **red highlighting** (true for both in-library and HTML5 validation).

It is also very easy to apply built-in validation messages for form fields.



All you need to do is to specify the *invalidMessage* property and set the necessary text of the message:

```
{ view:"text", label:'Username', name:"login", invalidMessage: "Login can not be empty" },
```

Validation messages require some space under form fields. You should specify the *bottomPadding* property and set the appropriate value to define the bottom offset of the input:

```
var formElements = [
    { view:"text", label:'Username', name:"login",
        invalidMessage: "Login can not be empty" },
    ...
];

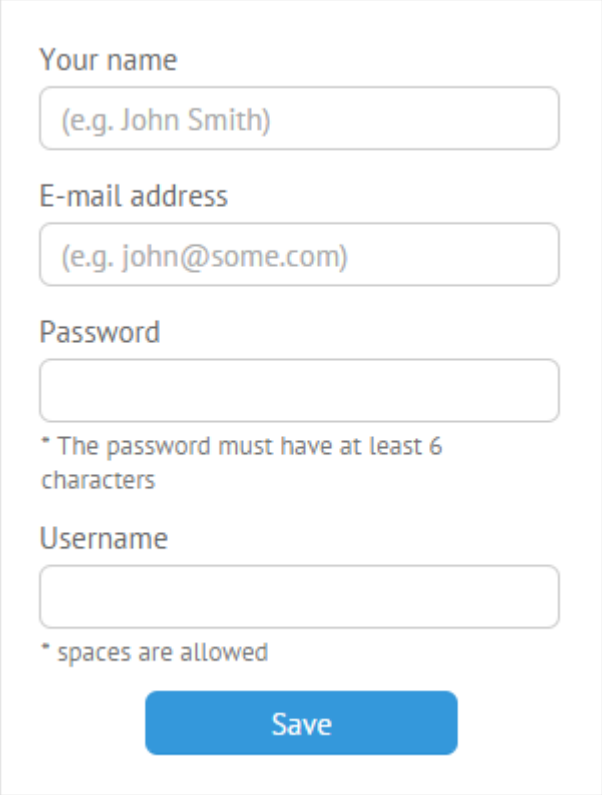
webix.ui({
    view:"form",
    elementsConfig:{
        labelPosition: "top",
        labelWidth: 140,
        bottomPadding: 18
    },
    elements: formElements
});
```

RELATED SAMPLE: Validation Message

The property is 18 px by default. If the value isn't set, space for validation messages will be automatically specified as 18px.

Adding input description

There's also a possibility to put some hints on correct filling of form fields for users.



The *bottomLabel* property will be helpful for this purpose. Don't forget to set the bottomPadding property to specify the size of labels.

```
{ view:"text", type:"password", label:"Password", bottomPadding: 35,
    bottomLabel: "* The password must have at least 6 characters"},
```

RELATED SAMPLE: Input Description at the Bottom

Validation Handling

Additionally, a **custom message** can be set to handle validation process. It's done in two ways:

1 . Webix message can be used together with the **validate()** method within the **sumbit_form** custom function;

```
webix.ui({
    view:"form",
    id:"myform",
    elements:[
        {},  //input fields
        { view:"button", inputWidth:300, label:"Submit",
click:"submit_form"}]
});
```

```
submit_form = function(){
    if (this.getParentView().validate()) //on success
        webix.message("All is correct");
    else
        webix.message({ type:"error", text:"Form data is invalid" });
};
```

2 . Webix message can be attached to validation events with the help of its **on** property. Possible events here are:

- **onAfterValidation** - triggers on validation process complete;
- **onValidationSuccess** - triggers if data passes validation successfully;
- **onValidationError** - triggers in case an error during validation happens.

Here you can see how message can be formed in case of a validation error:

```
on:{
webix.ui({
    view:"form",
    on:{
        onValidationError:function(key, obj){
            text = "Login can't be empty";
            webix.message({ type:"error", text:text });
    }
});
```

RELATED SAMPLE: 'onValidationError' Event

Message boxes of all types are described here.

Validation Rules

The **validate()** function checks if data complies to certain rules defined in the **rules** object that contains a collection of **property-rule** pairs. Predefined rules are accessed through **webix.rules** class.

Form validation

Here you attach rules to inputs by their **names**. **Name** is read-only property that can be assigned to any input of the form/ htmlform.

You can do this either within **rules** object property of the form component:

```
view:"form", elements:[
    {view:"text", name:"field1"},
    {view:"text", name:"field2"}
],
rules:{
    field1:rule1,
```

```
    field2:rule2
}
```

RELATED SAMPLE: Basic Validation

Or, you can attach a rule right in the **input constructor** as value of its **validate** property:

```
view:"form", elements:[
    { view:"text", label:'Is a Number', validate:webix.rules.isNumber,
name:"text2" },
    { view:"text", label:'Is an Email', validate:webix.rules.isEmail,
name:"text3" }
],
elementsConfig:{
    labelAlign:"right",
    on:{
        'onChange':function(newv, oldv){
            this.validate();
        }
    }
}
```

RELATED SAMPLE: Per Item Validation with Predefined Rules

Component Data Validation

Here you need to define which data item you'd like to validate, namely specify its template #value# or, in case of datatable, column ID.

```
rules:{
    "template/ID": rule
}
```

Validation rules can be of three types:

- Predefined Rules
- Custom Rules
- Rules using Special Validation Keys

Predefined Rules

There are four of them:

- **isNotEmpty** - checks whether a value is empty;
- **isNumber** - checks whether a value is number;
- **isEmail** - checks whether a value is e-mail (looks for an @-sign, fullstop, text after it);
- **isChecked** - checks whether a checkbox is checked.

```
webix.ui({
    view:"form1",
    elements:[
        { view:"text", label:'Username', name:"login" },
        { view:"text", label:'E-mail address', name:"email"},
        { view:"text", label:'Password', name:"password"},
        { view:"checkbox", labelRight:'I accept terms of use',
name:"accept"}

    ],
    rules:{
        login: webix.rules.isNotEmpty,
        email: webix.rules.isNumber,
        phone: webix.rules.isEmail,
        accept: webix.rules.isChecked
    }
});
```

RELATED SAMPLE: Validation with Predefined Rules

Note that in case you define rules within input constructor, the **isNotEmpty** rule can be substituted by **required** property.

```
//it's better to use
{ view:"text", label:'Not empty', required:true }
//instead of
{ view:"text", label:'Not Empty', validate:webix.rules.isNotEmpty }
```

RELATED SAMPLE: Required inputs in a form

Conditional usage of built-in rules

Predefined validation rules can be wrapped by another function to apply them selectively:

```
Ignoring empty values during validation

rules:{
   city:function(value){
        return !value || webix.rules.isNumber(value)
  }
}
```

Remember that a validation function must return *true* for success and *false* for failure.

Custom Rules

Basically, any custom function can define a rule. Such function takes the validated **value** as parameter.

Validation rules may define the minimal and maximum value for the form field or check whether the input data complies to a certain criterion.

```
Validation Criteria for Numbers

webix.ui({
    view:"form",
    rules:{
        text1:function(value){ return value > 100; },  // over 1400
        text2:function(value){ return value < 100; }, // below 100
        text3:function((value){ return value > ;} //positive
    }
});
```

RELATED SAMPLE: Validation with Custom Rules

Special Validation Keys

The keys are used with both predefined and custom rules for special cases, for instance:

- if you need apply the same rule to each property of data object, use **$all** key;
- if you need data from several form fields for validation, use **$obj** key.

$all key

```
All the fields must be filled

rules:{
    $all:webix.rules.isNotEmpty
}
```

RELATED SAMPLE: Validation with a Common Rule for All Fields

$obj key

The whole data object is passed into the rule while field data is accessed as **obj.field_name** or **obj.template/ID**.

```
Both votes and rank values must be positive

rules: {
    $obj:function(obj){
        return obj.votes >  && obj.rank > ;}
}
```

RELATED SAMPLE: Datatable Validation. Complex Rule.

It is as well used in defining complex rules, e.g. when password confirmation is required.

```
Passwords must be equal

rules:{
    $obj:function(data){ //data = obj
        if (data.pass1 != data.pass2){
            webix.message("Passwords are not the same");
            return false;
        }
    return true;
    }
}
```

RELATED SAMPLE: Password Confirmation

Note, that even if you don't pass data object into the function, you still can work with all its values deried with form **getValues()** method.

```
Complex Rule

webix.ui({
    view:"form",
    rules:{
        $obj:function(){
            var data = this.getValues(); //!getting data object
            if (!webix.rules.isEmail( data.email ))  //ckecking email
                return false;
        if (data.name == "") //checking name
            return false;
        return true; //success!
        }
    }
});
```

RELATED SAMPLE: Validation with a Complex Rule

What should be taken into account:

- If there are fields without **name** property, the function returns *false*;
- In case all rules are observed, the function returns *true* and the form is treated as valid.

Data Validation for Components

As with form, rules are set within the component constructor. With validation enabled, you'll be warned about invalid data in your component on client side while avoiding loading wrong data to server.

The validated data item is defined by its template, rules being the same as with form elements:

```
List Data Validation

webix.ui({
```

```
    view:"list",
    template:"#rank#. #title#",
    rules:{
        title: webix.rules.isNotEmpty
    },
    ..//list config
})
```

As a result, you cannot add an empty record to list data.

RELATED SAMPLE: List: Validation

Read more about data templates here.

With datatable component we need a column ID, since it correlates with data items.

```
Datatable Data Validation

webix.ui({
    view:"datatable",
    columns:[
        { id:"rank", header:"", ..},
        { id:"title", header:"", ..}
    ],
    rules:{
        "rank": positiveNumber, //custom rule
        "votes": positiveNumber
    }
})
```

RELATED SAMPLE: Datatable Validation

Invalid data is highlighted within the component and cannot be sent to server (in case you work with dataprocessor).

Validation during in_Component editing

By default, with rules defined, validation is started each time we finish editing and close the editor for this or that data item.

However, **editing events** allow validation with an editor opened. The obvious bonus of such method is that until new data is valid, it can't be changed within the component.

Here the **onBeforeEditStop** event comes to our help. It's attached to the component you work with:

```
webix.ui({
    view:"datatable",
    ..//config
    on:{
        onBeforeEditStop:function(state, editor, ignore){
            var check = ( editor.type.getValue() != "" );
```

```
            if (!ignore && !check){
                webix.message(editor.column+" must not be empty");
                return false;
            }
        }
    }
});
```

RELATED SAMPLE: Datatable. Editing Validation.

The function takes three arguments here:

- **state** - the object that stores old and new value of the text editor;
- **editor** - the editor being used;
- **ignore** - the flag to indicate user action after editing, whether they save the new data or escape the saving thus restoring the old data.

More information about text editors you can find in the dedicated article.

HTML5 Data Validation

Here we speak about clientside, in-browser data validation powered by HTML5 means.

In HTML5 layout input tags come in a plenty of types and feature a number of attributes that define validation pattern.

Input types conside with a **type** property of the view **text** while other input attributes are listed within the **attributes** object property:

HTML

```
<input type="text" id="text1"
    title="First name is required!"
    maxlength="25"
    placeholder="Your name"
required>
<label for="text1">First
name</label>
```

Javascript

```
{ view:"text", type:"text",
attributes:{
    maxlength:25,
    required:"true",
    title:"First name is required!"
}, value:'', label:"First Name"}
```

Here validation is defined by **required** attribute.

The validation process starts as soon as you type the first character into the field. When the necessary pattern is observed, a red highlighting disappears. Furthermore, if you try submitting the form, a validation error message appears for each field that has been incorrectly filled:

- in case of the wrong filling: "Please enter an email address/ a URL !" - depending on the *type* property value;

- in case of an empty field: "Please fill out this field!"

HTML5 input types for the library's form component are as follows::

- **text** (default type for this lib);
- **password** - hides typed characters;
- **email** - requires **@** symbol for validation;
- **url** - requires **http://** prefix for validation.

RELATED SAMPLE: HTML5 Data Validation

Related Articles

- Creating Message Boxes

Intermediate

Editing Data Items

Library components have similar editing patterns yet some peculiarities should be taken into account. Hence, there exist two groups:

- Editing Datatable and Treetable
- Editing Other UI-Related Components

Generally, a set of properties in the component's constructor enable on-the-go editing with its items. At the same time, programmatic editing is as well possible.

For these needs take the following steps during component initialization:

- set the **editable** property to the true state;
- define **editor** type.
- define **editAction** (single or double mouseclick, or keyboard action).

With **datatable** you should include the **editor** property into a column object.

Data **editors** and **edit actions** are described here;

Datatable and Treetable Editing

By default, only datatable and treetable feature built-in editing ability.

Editable and **editAction** are included into the component object while **editor** is set for each column separately.

```
webix.ready(function(){
    webix.ui({
    view:"datatable",
    columns:[
        { id:"rank", editor:"text", header:""},
        { ...},
        { ...}
    ],
    editable:true,
    editaction:"click",
    data: small_wide_film_set //data from external js-file
});
```

RELATED SAMPLE: Opening multiple editors

Editing Other UI-Related Components

To enable edit mode with other ui-related components, you should create a **prototype object** on their base and extend them with editability. Then they will get the above mentioned functionality.

For instance, to edit ui-related list, create an editable prototype for it first and give it name to your taste:

```
webix.protoUI({
    name:"editlist" // or "edittree", "dataview-edit" in case you work
with them
}, webix.EditAbility, webix.ui.list);
```

And proceed with the list object initialization. Note that view is called "editlist" value instead of "list," since it's the name of the newly create object from the code above:

```
webix.ui({
    container:"listA",
    view:"editlist",
    template:"#rank#. #title#",
    editable:true,
    editor:"text",
    editValue:"title",
    data:big_film_set
});
```

RELATED SAMPLE: List: Editing

Take notice of the **editValue** property here. It defines the dataset item from the template since each record in list and similar components has a template with several values from the initial data source. Study the nature of template-making here.

Note that editing allows for on-page changes only and doesn't presuppose automatic data saving into the dataset. To actually save data you should init webix.DataProcessor for the edited component.

Editing Methods

There's a collection of methods to enable and control editing.

Generally, you can switch any data item to the editable state with the **edit()** method that requires item ID as parameter:

```
editlist.edit(7);
```

To open editor in the datatable cell you need to specify row and column ID-s in casse of However, it's recommended to use *editRow*, *editColumn* and *editCell* functions. See the datatable API for details.

```
datatable.edit({
    row:2,
    column:"title"
});
```

In addition, the following methods are applicable:

- **editNext**() - closes the currently opened editor and opens editor in the next editable item.
- **editStop()** - closes all opened editors. Data is automatically saved;
- **editCancel()** - cancel editing and closes all opened editors. Data isn't saved.
- **editFocus()** - enables editing in the editor in focus (the method is helpful for several active editors on the page).

Check editing methods in the API reference.

Editing via a Form

Form and Component Binding

Webix components can be bound to each other to ensure select-based **synchronous changing** of their **data**. For instance, a simple function can be used to bind a form to a grid, which allows to edit datatable data:

- clicking the component item will trigger form filling;
- then you edit data in the form;
- form [saving] will send changed data back to the component.

```
$$('form1').bind('datatable1');
```

RELATED SAMPLE: Editors. Bind Form

Note that **name** attributes of form fields coincide with **properties (fields)** of the data used for the edited component while the component can be non-editable itself. You can study data binding separately.

Related Form for an Editable Component

Webix editing API allows to attach a form without data binding as well. Unlike binding, it works only for editable components.

First of all, you need to define the related form inside a master component with the help of form property:

```
    view:"datatable",
    editable:true,
```

```
    form:'myform' //ID or object of related form/popup
```

A form should be defined separately, either in layout or in popup.

```
view:"form",
id:"myform",
elements:[...]

//or
view:"popup",
id:"myform",
body:{
    view:"form", elements:[...]
   }
```

Note that:

- ID or object attached to a component by the **form** property should point to either a form, or a popup in which the form lies;
- In case of a **popup**, it can be initialized hidden so that it will be shown only when editing starts (when you click data item in master component).

Related Articles

- Data Update - the alternative method of item editing
- Data Editors

Intermediate

Data editors

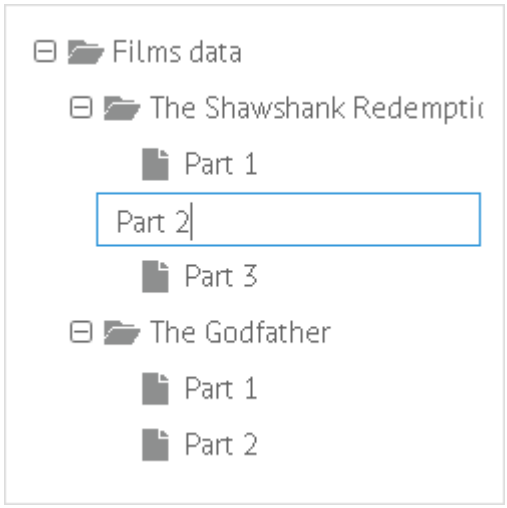
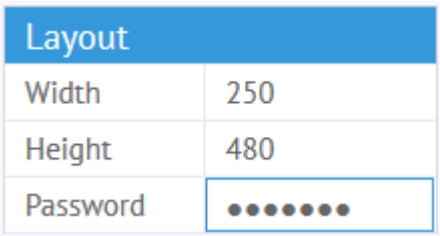
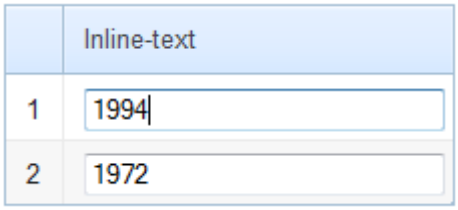
In this article build-in component editors and edit actions for them are discussed. To learn about Webix editing pattern, go to the main article.

Opening and Closing Editors

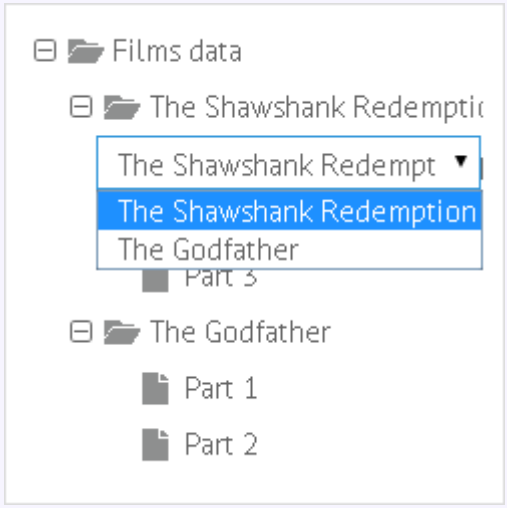
- an editor for the component is enabed, or opened, by the **editAction** that is defined for the whole component;
- an editor is closed by **moving focus outside the editor** (e.g. clicking another component part) or by hotkeys **'Enter'** for saving the new value and **'Esc'** for exiting without saving.

Editor Types

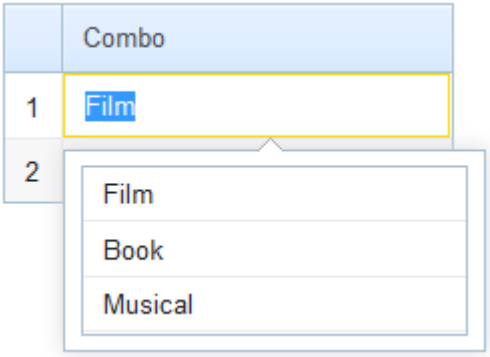
Table Editor Types

`editor:"text"` Learn more	
`editor:"password"` Learn more	
`editor:"inline-text"` Learn more	

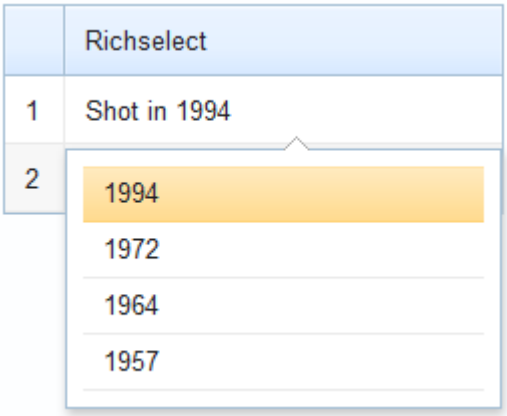
`editor:"select"` Learn more



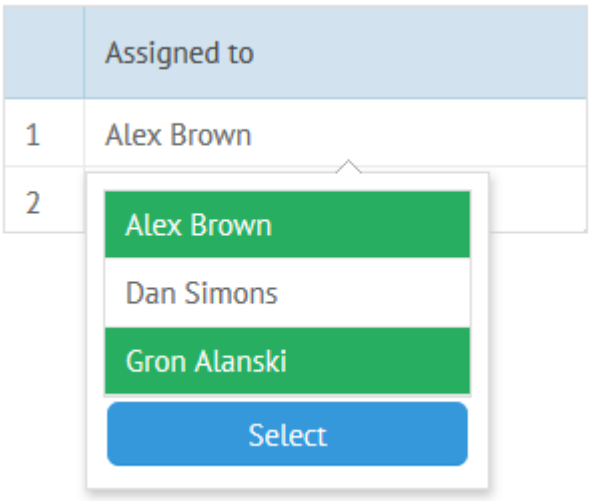
`editor:"combo"` Learn more



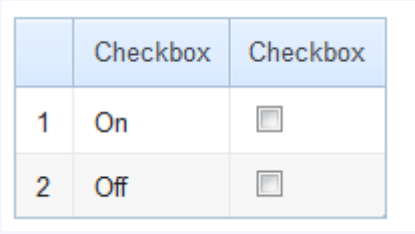
`editor:"richselect"` Learn more



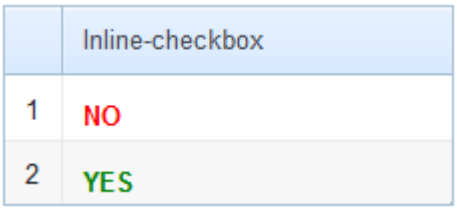
`editor:"multiselect"` Available in **Webix Pro** edition only Learn more



editor:"checkbox" Learn more



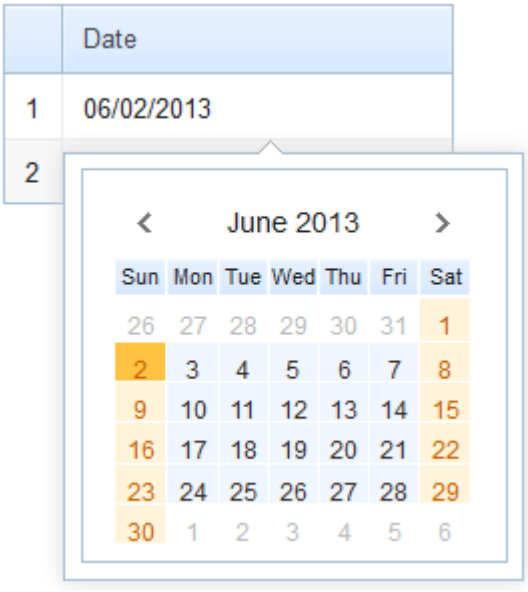
editor:"inline-checkbox" Learn more



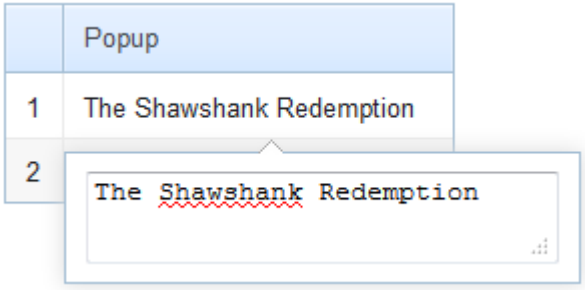
editor:"color" Learn more



editor:"date" Learn more



editor:"popup" Learn more



RELATED SAMPLE: Editor Collection

Text

A base editor for text values of datatable and dataview cells, list rows. By default is enabled on clicking the needed item;

```
{ id:"title", header:"Film title", editor:"text",
```

RELATED SAMPLE: Basic use of editors

Password

A base editor for passwords that masks symbols in the input.

```
//in the property sheet editor is defined by "type"
{ id:"pass", label:"Passoword", type:"password"}
```

RELATED SAMPLE: Property Sheet: Basic Initialization

Inline-text

A customizable text editor. Here you should set a template for the editing area, type and dimensions of an input area;

```
{ id:"year",template:"<input type='text' value='#value#'
style='width:130px;'>",
    editor:"inline-text"}
```

RELATED SAMPLE: Using 'inline' editors

- If you specify the **editor:"inline-text"** attribute the component will provide a special processing treat for the editor while editing: invoking edit-related events.
- If you don't specify the **editor** attribute you should provide the processing logic for the editor on your own.

Select

A customizable editor that allow for choosing one of the offered values. It looks like a standard HTML select control.

```
{ id:"type", header:"Category",  editor:"select", options:[...]}
```

RELATED SAMPLE: Using the 'select' editor

The options can be defined by either a simple or an associatibe array as well as by url. Look here for details.

Combo

Webix combo control with the possibility to filter a popup list by entering symbols into the dedicated input;

```
{ id:"title", header:"Film title",  editor:"combo",  options:[...]}
```

RELATED SAMPLE: Datatable: Combo Editor

- The options can be defined by either a simple or an associatibe array as well as by url. Look here for details;
- The editor is subject to extensive customization.

Pro Extensions for Combo Editor

In the **Webix Pro** edition the editor can be extended to show either a **dataview** or **datatable** in the popup:

- Dataview Suggest
- Grid Suggest

Richselect

Webix richselect control that is a non-editable variation of a combo editor.

```
{ id:"title", header:"Film title",  editor:"richselect",  options:[...]}
```

RELATED SAMPLE: Datatable: RichSelect Editor

- The options can be defined by either a simple or an associative array as well as by url. Look here for details;
- The editor is subject to extensive customization.

Pro Extensions for Combo Editor

In the **Webix Pro** edition the editor can be extended to show either a **dataview** or **datatable** in the popup:

- Dataview Suggest
- Grid Suggest

Multiselect



The editor is available is **Webix Pro** edition only

The editor is based on Webix multiselect control and allows selecting multiple options at a time.

```
{ id:"assigned", editor:"multiselect", optionslist:true, options:[...]}
```

RELATED SAMPLE: Multi-select editor in the DataTable

- The **optionlist** property is defined for the editable element (here:column) so that it can parse multiple values from the editor into a single cell value.
- The options can be defined by either a simple or an associative array as well as by url. Look here for details.
- The editor is subject to extensive customization.

Checkbox

A checkbox editor presupposes the choice between two values (true or false). In general, yon can assign any value to "true" and "false" checkbox states.

```
{ id:"ch3", header:"CheckBox",template:"{common.checkbox()}",
editor:"checkbox"}
```

The notation above does nothing with checkbox value. If you want to derive its value as data item property, you should either:

1) Define checkbox options with values for each state:

```
{ id:"ch3", template:"{common.checkbox()}", editor:"checkbox", options:{
    "true":"On","false":"Off","undefined":"Off"}
```

RELATED SAMPLE: Using 'inline' editors

2) Set **checkValue** and **unCheckValue** pair:

```
{ id:"ch1", checkValue:'on', uncheckValue:'off',
template:"{common.checkbox()}",
    editor:"checkbox"}
```

RELATED SAMPLE: Checkbox and radio in DataTable

Inline-checkbox

A customizable checkbox. You can do without a checkbox icon and define any template for boolean values, even a styled text.

```
function custom_checkbox(obj, common, value){
    if (value)
        return "<div class='webix_table_checkbox checked'> YES </div>";
    else
        return "<div class='webix_table_checkbox notchecked'> NO </div>";
}
    ...
{ id:"ch1", header:"", template:custom_checkbox, width:40, editor:"inline-
checkbox"},
```

RELATED SAMPLE: Custom radio and checkbox in DataTable

Color

It is used for editing color value in the dataset with the help of a colorboard. Colors are stored as hex codes; hence, you should set an appropriate template for a component item to display the chosen result as color, for instance, make a colored background.

```
var tpl="<span style='background:#value#; border-radius:4px;'> </span>
#value#";

{ id:"start", editor:"color", template:tpl, header:"Color A",width:120}
```

RELATED SAMPLE: Color Editor

Date

In its essence it is a datepicker that initializes a calendar to pick the necessary date.

```
{ map:"(date)#enddate#",    editor:"date",  header:"End date",
width:120},
```

RELATED SAMPLE: Date Editor

Popup

A popup window with body contents depending on data you edit.

- **textarea** - a multi-row input for editing long texts. The popup features *width: 250* and *height:50* by default.

```
{ id:"title",   header:"Film title", editor:"popup"}
```

RELATED SAMPLE: Pop-up editors

- **colorboard** and **caledar** - come in popup windows by default and are set as **date** and **color** editor types.

$Popup

configuration for popup editors.

Popup Configuration

You can configure editing controls inside popup windows, e.g. define other dimensions, alter properties of textarea, colorboard and calendar (check the corresponding articles.)

Popup editors can be configured in a scope with the following code. All the same-name editors in the application will be redefined:

```
//config may as well include only text, color and date hash
webix.editors.$popup = {
    text:{
        view:"popup", width:250, height:200,
        body:{view:"textarea"}
    },
    color:{
        view:"popup",
        body:{ view:"colorboard", width:500, height:500, rows:50, cols:50
}
    },
    date:{
        view:"popup",
        body:{ view:"calendar", weekNumber:true }
    }
};
```

Any instance of a popup editor can be configured separately, with the help of a **popup** property of an item in question:

```
Adding a weekHeader to the Date editor

{ id:"date",editor:"date", popup:{
    view:"popup",
    body:{
        view:"calendar", borderless:true, weekNumber:true
    }
}
```

Note that **select editors** (select, richselect, multiselect, combo) offer a wide range of customization possibilities that are described later.

Defining options for select editors

Options for select editors (**select**, **combo** and **richselect**) are defined under the common pattern width the help of '**options**' attribute of the dedicated column. There exist several possibilities:

1) Options are set **directly** in the "options" attribute either in an **index** or **associative array** as well as in a **JS object**:

```
Options defined by a simple array in the column config

columns:[
    { id:"cat_id", editor:"select", options:["Crime", "Thriller",
"Western"]}
]
```

RELATED SAMPLE: Using the 'select' editor

```
Options defined by an associative array in the column config

columns:[
    { id:"cat_id", editor:"select", options:[
        {id:1, value: "Crime"},
        {id:2, value:"Thriller"},
        {id:3, value:"Western"}
    ]}
]
```

```
Options defined by a JS object

columns:[
    { id:"cat_id", editor:"select", options:[
        "1" : "Thriller",
        "2" : "Crime",
        "3" : "Western"
    ]}
]
```

RELATED SAMPLE: Using the 'select' editor. Specifying select options in a variable

2) Options are defined by a **separate variable** containing option collection. In this case the "options" attribute must be set to the name of this variable.

```
Setting select options in a variable

var film_options = [
    {id:1, value: "Thriller"},
    {id:2, value: "Crime"},
    {id:3, value: "Western"}
];
//or
var film_options =["Crime", "Thriller", "Western"];
...
columns:[
    { id:"cat_id", editor:"select", options:film_options,
header:"Category"}
]
```

RELATED SAMPLE: Datatable: Combo Editor

3) Options are taken from **file** or loaded by **url** defined by "string" value of options attribute.

```
Options loaded from an external file

columns:[
    { id:"cat_id",  editor:"select", options:"data/options.json"}
]
```

RELATED SAMPLE: Loading data in the 'select' editor

"Options" have an alias attribute **"collection"** that features the same functionality.

```
Using "collection" to define options

columns:[
    { id:"cat_id", editor:"combo",  collection:film_options,
header:"Category"}
]
```

Advanced configuration of select editors

Webix select controls, **combo**, **richselect** and **multiselect**, are highly customizable, which means that the same-name editors
can be customized too. The following things can be changed:

- template of list items;
- template of the input field (combo editor only);
- filtering pattern (combo only);
- text of "Select" button (multiselect only);
- the number of list items in the popup;
- functions can be attached to a list.

Two column attributes are mandatory - **options** or **collection** to interprete options into text and **suggest** to define the confuguration of related popup list:

```
{ id:"year", editor:"combo", options:years, suggest:{
     template:'#value#', //template of the input when editor is opened,
default
     filter:function(item,value){ //redefines default webix combo filter
        if
(item.value.toString().toLowerCase().indexOf(value.toLowerCase())===)
            return true;
        return false;
     },
     body:{
        template:'Year #value#', //template of list items
        yCount:7, //10 by default
        on:{
           'onItemClick':function(id){
                webix.message(this.getItem(id).value);
            }
        }
     }}
```

```
}
```

RELATED SAMPLE: Datatable: Advanced Combo Editor

Multiselect specificity

To configure a suggest list for a multiselect editor, one should specify its suggest type - **multisuggest**:

```
{ id:"year", editor:"multiselect", options:years, optionlist:true,
suggest:{
    view:"multisuggest",
    buttonText:"Select items"
}}
```

Datatable and treetable specificity - live editors

Webix datatable, treetable and property sheet can use any of the above mentioned editors in **live** mode and update the same data property is case it's used in another column other than the edited one.

```
//"rating" is used in both columns while it can be editable ony in the
first one
columns:[
    { id:"rating",  header:"Count", editor:"inline-text", liveEdit:true},
    { template:"#rating#", width:150 }
]
```

RELATED SAMPLE: Using 'live' editors

The moment the second column (the one that is automatically edited) changes its value, onLiveEdit event takes place.

Edit Actions

- **'click'** - opens the editor on a single click (used by default);
- **'dblclick'** - opens the editor on a double click;
- **'custom'** - allow setting the necessary way of editing, e.g a keyboard event.

When setting an editaction to "custom", you need to refer to UIManager and write a function to enable the needed editing pattern:

```
webix.UIManager.addHotKey("enter", function(view){
    var pos = view.getSelectedId();
    view.edit(pos);
}, gridc);
```

Here editing is enabled on pressing the "Enter" key on the selected item.

Custom Editors

The library allows creating fully custom editors.

```
webix.editors = {
    "mytext":{
        focus:function(){...}
        getValue:function(){...},
        setValue:function(value){...},
        render:function(){...}
     }
  };
```

To define a new editor, you should specify at least 5 methods for it:

- **focus()** - sets the focus to the input with the editor;
- **getValue()** - gets the value of the editor;
- **setValue()** - sets the value of the editor;
- **render()** - renders the editor;
- **getInputNode()** - gets the input area object.

For instance, this is how a biult-in text editor is made:

```
webix.editors.myeditor = {
    focus:function(){
        this.getInputNode(this.node).focus();
        this.getInputNode(this.node).select();
    },
    getValue:function(){
        return this.getInputNode(this.node).value;
    },
    setValue:function(value){
        this.getInputNode(this.node).value = value;
    },
    getInputNode:function(){
        return this.node.firstChild;
    },
    render:function(){
        return webix.html.create("div", {
            "class":"webix_dt_editor"
        }, "<input type='text'>");
    }
}
```

Inner properies

- **this.node** - HTML tag of the editor. Appears after *render()* execution;
- **this.value** - the initial value of an input. Appears after *setValue();* execution;

- **this.config** - editor configuration;
- **this.popup** - popup ID (is used for editors with popup windows)

Extending Functionality of an Editor

In addition to creating an editor "from scratch", you can extend any of the existing ones. It becomes helpful when slight changes are needed.

For instance you have a datatable column with e-mails. Input type **email** from HTML5 package will warn you in case some of e-mail address features is missing. The cell will be highlighted red.

	Name	Email
1	Alex	alex.foo.com
2	Jim	jimmy@foo.com

In this case you extend the existing **text editor** and change render pattern in it:

```
webix.editors.myeditor = webix.extend({
        render:function(){
        return webix.html.create("div", {
            "class":"webix_dt_editor"
        }, "<input type='email'>");
    }}, webix.editors.text);
```

RELATED SAMPLE: Custom Text Editor

Intermediate

Paging

Paging helps present numerous items from the dataset dividing them into groups to display on separate pages. It is used in conjunction with data-presenting components like dataview, datatable and list.

Pager can be initialized as

- separate ui-related component (should be linked to a component to enable paging);
- as property of the component it is used for.

In both cases pager can be placed into its own HTML container different from that of the associated component.

Main properties of a pager object:

- size - the number of records per page;
- group - the number of pages (numbered squares for paging);
- page - sets the page that will be shown intially (zero-based numbering).

If a component container is too small to display the number of items specified by size, a scrollbar appears.



#	Film title	Released	Votes
21	The Matrix	1999	492.325
22	Rear Window	1954	148.162
23	Raiders of the Lost Ark	1981	300.252
24	The Silence of the Lambs	1991	324.419
25	The Usual Suspects	1995	331.99
26	Psycho	1960	182.319
27	Se7en	1995	380.479

RELATED SAMPLE: Paging animation

Paging and Dynamic Loading

With dynamic loading, only part of the stored records are loaded into the component during init. Each time you page through the component, a data request is sent to server to load the next portion of data.

Read more about dynamic loading.

Pager as separate view

In this case pager is initialized as separate component in layout and is linked to the datatable by ID.

```
webix.ui({
    rows:[
        {
            view:"datatable",
            columns:[ ...],
            pager:"pagerA", //linking to a pager
        },
        {
            view:"pager", id:"pagerA",
            size:50,
            group:5
        }]
})
```

RELATED SAMPLE: Visible Pager in Layout

In this case when pager is a standalone view size property can be combined with either limit or count property to set the total values for the pager, namely:

- limit - the total number of pages in pager. The number of records will be calculated automatically.
- count - the total number of records the pager will process. The number of pages will be calculated automatically.

You can as well use Webix pager as a separate entity without master component and provide fully custom login for it - go to the end of the article for details.

Pager as component property

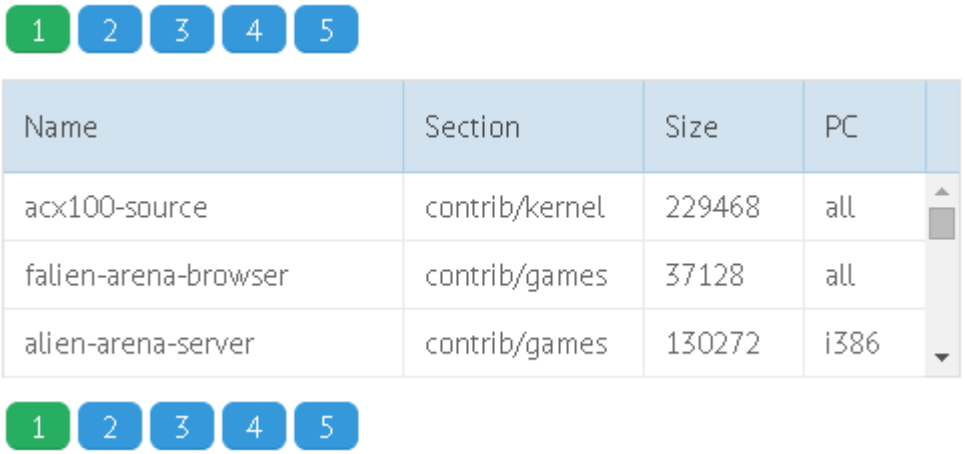
Here pager object appears to be the property of a datatable and includes two parameters of its own.

```
webix.ui({
    view:"datatable",
    columns:[...],
    pager:{
        size:50,
        group:5
    }
});
```

RELATED SAMPLE: Visible pager

Several Pagers per Component

If you want more than one pager for the component, for instance the ones above and below the view, there should be two containers in HTML layout, or two views with pager instances. The two pagers must be linked to each other with the help of the clone method.



You can clone both standalone pagers and those included into the component constructor.

```
Cloning a standalone pager

//an identical pager
$$("pagerA").clone($$("pagerB"));

//a pager with some properties of its own
$$('pagerA').clone({
    container:"pagingB_here",
    size:50,
    group:5
});
```

RELATED SAMPLE: Two-area pager

With a component pager you need to get the pager object with the **getPager()** method:

```
Cloning a component pager


grida.getPager().clone({
    container:"paging_here_too",
    size:10,
    group:5
});
```

RELATED SAMPLE: Defining several pagers on a page

Cloned pagers will change their values sumultaneously!

Invisible pager

Paging can be implemented through an invisible pager, which gives you an ability to control the process by custom functions and create your own "Next/Previous Page", "Jump to Start/End" buttons, etc. instead of a standard pager.

For these needs you should include **apiOnly** parameter with *true* value into the pager constructor, whether it is inited as a separate view or as a component's property.

```
Datatable with Invisible Pager

view:"datatable",
...
pager:{
    id:"pagerA",
    apiOnly:true, //makes pager invisible
    size:50,
    group:5
}
```

In this case, dedicated buttons will help us page:

```
Next Page Button

<input type='button' value='next page' onclick='next_page()'>
...
function next_page(){
    $$("pagerA").select("next");
}
```

RELATED SAMPLE: Not-visible pager

Select() function can take the **index** of the desired pager (zero-based numbering,) or its **order* ("next", "prev", "first", "last").

Custom Templates for Paging

By default, pager looks as numbered squares accompanying the component . But at the same time you can change the pattern with the help of a custom template:



Template elements may include the following elements:

- *{common.prev()}* - 'less-than' sign, directs to previous page;
- *{common.first()}* - double 'less-than' sign, directs to the first page;
- *{common.next()}* - 'greater-than' sign, directs to the next page;
- *{common.last()}* - double 'greater-than' sign, directs to the last page;
- *{common.page()}* - displays the number of the current page (starting from 1);
- *{common.pages()}* - displays a set of numbered squares;
- pager data elements (e.g. '#limit#' to display the total number of pages);
- standard text.

```
pager: {
    template: "{common.prev()}{common.next()}Page {common.page()} from
#limit#"
}
```

RELATED SAMPLE: Custom Pager Template

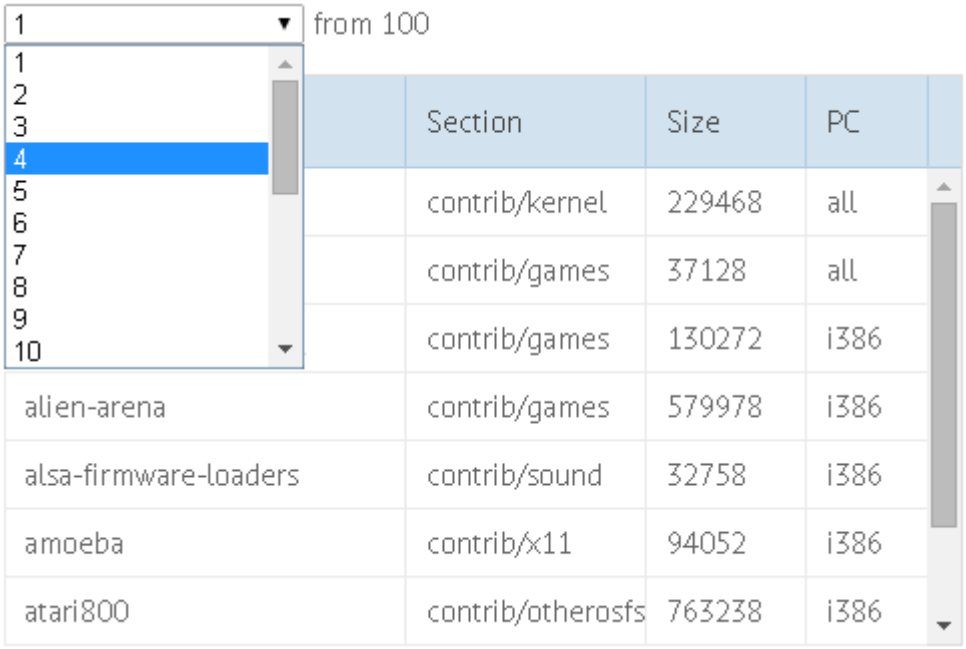
HTML Custom Pager

You can use HTML elements as a base for a custom pager that includes both **HTML tags** and **pager properties** in its **template**.

Properties of the component pager are stored in the **data** object, pager templates reside in the **common** object:

- **data.page** - the index of the current page(zero-based numbering);
- **data.size** - the number of data records per page (either set manually, or calculated on the base of component size)
- **data.limit** - the number of record processed. Equals to the number of records in the connected component.
- **common.next(), common.prev(), common.first(), common.last()** - page order signs.

Custom 'select' pager




```
pager:{
    template:function(data, common){
        var current = data.page + 1;
        var html = "<select onchange='grida.setPage(this.value)'
style='text-align:center; width:150px'>";
        for (var i=1; i<=data.limit; i++)
            html+="<option "+(i ==
current?"selected='true'":"")+">"+i+"</option>";
        html+="</select> from "+data.limit;
        return html;
    },
    container:"paging_here",
    size:10,
    group:5
}
```

RELATED SAMPLE: Custom pagers

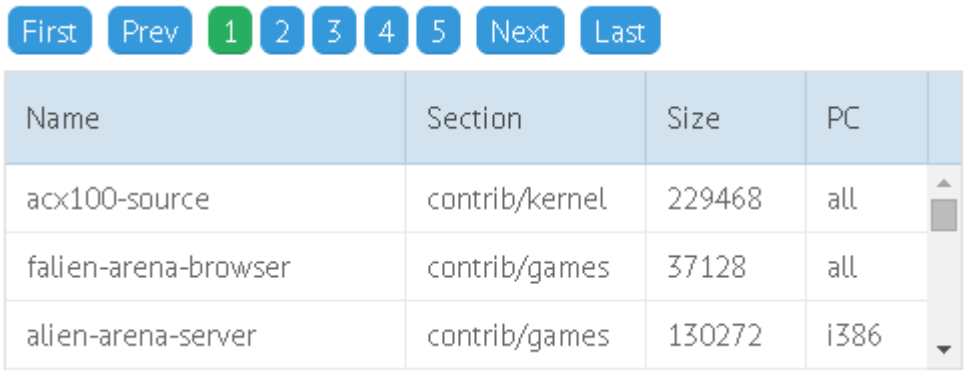
Pager Localization

By default, pager uses recognizable signes to display **order buttons** (first, last, prev, next).

```
Default Locale

webix.locale.pager = {
    first: " &lt;&lt; ",
    last: " &gt;&gt; ",
    next: " &gt; ",
    prev: " &lt; "
};
```

Pager localization allows using words of any language for these buttons:




```
Custom Locale

webix.locale.pager = {
    first: "First",
    last: "Last",
    next: "Next",
    prev: "Prev"
};
```

RELATED SAMPLE: Pager localization

Animation during Paging

Animation it switched on by **animate** property. Set to *true*, it allows scrolling through the view in X or Y mode, depending on the scroll value of the main component.

Animation can take various effects when they are stated in the **animate** constructor through type, subtype and direction parameters.

```
pager: {
    animate:{
        direction:"top",
        type:"flip"
    }
}
```

RELATED SAMPLE: Paging animation types

Learn more about animation types and subtypes in the corresponding chapter of the manual.

Page and Component Size

Here we treat two tasks:

1 . How to ensure that the number of records per page equals to the number of records the component can display?

The number of records per page is stated by the **size**:(number) parameter of a pager.

At the same time, the number of records can be limited in the component itself by the following parameters:

- **yCount** (number) - the number of items in a vertical mode;
- **xCount**(number) - the number of items in the horizontal mode (typical of dataview and x-list).

All you need is to make these two parameters the same:

```
rows:[
    {
        view:"pager",
        id:"pagerA",
        size:10
    },
    {
        view:"datatable",
        columns:[...],
        pager:"pagerA",
        yCount:10
    }
]
```

RELATED SAMPLE: Adjusting Component Size to Page

2 . How to **adjust component size** to page size?

To adjust the number of records shown per page to the size of parent HTML container and, hence, to the component's height, you should let pager autosize.

```
<div id="testA" style='height:600px; width:300px;'></div>
...
webix.ui({
    rows:[
        {
            view:"pager",
            id:"pagerB",
            autosize:true,
            group:5
        },
        {
            view:"list",
            pager:"pagerB",
            container:"testA"
        }
    ]
});
```

RELATED SAMPLE: Adjusting Page Size to Component Size

Paging Methods

Getting Pager Object

```
$$("datatable1").getPager();
```

Getting Current Page

Note that you'll get page index (zero-based numbering), so add 1 to get the actual page number.

```
$$("datatable1").getPage();

//or, using the pager object
$$("pagerA").config.page;
```

Setting (Opening) Page

To programmatically switch to the necessary page, specify the page's index within the dedicated method:

```
$$("datatable1").setPage(2); // makes the third page visible
```

..or select the needed page:

```
$$("datatable1").getPager().select(2); // -> selects the third page
```

Standalone Pager

Pager can be used without master component for creating custom logic. To render such pager, the following code should be used:

```
webix.ui({
  view: 'pager',
  template: '{common.prev()} {common.pages()} {common.next()}',
  master:false,
  size: 50,
  group: 5,
  count: 1000,
  on: {
      onItemClick: function(id, e, node) {
          //..some custom logic..
      }
  }
});
```

Template parameter can be omitted in favour of default template.

RELATED SAMPLE: Standalone pager

Note that count and size parameters should be used anyway, **count** divided by **size** equals to the **total number** of pages.

Intermediate

Drag-n-Drop within Webix Views

Basics of DnD

Here we speak about **simple drag and drop** of items within one and the same component as well as between different views and their instances. The information about how to make any app element draggable is to be found in the dedicated article of the documentation.

Drag-and-Drop Modes

All the components you'd like to work with must have **drag** property, both source and target components. In case of several instances of one and the same component, it would be enough to enable DnD. For adequate DnD between different components study external data functionality.

The property has several values to define DnD mode:

- drag:*true* - enables drag-n-drop within the component, allows taking items into its other instances and different components on the page;
- drag:"*order*" - enables DnD in the **reorder mode**, allows dragging component item within one component thus changing their order;
- drag:"*source*" - enables dragging items from the component without an ability to drop data from other components;
- drag:"*target*" - enables dropping data to the component. It can take data from outside, but you cannot drag anything from it.

Mutlidrag mode (dragging of several items at a time is possible) is enabled by setting multi selection ability within the component.

Multi Selection:

- multiselect: true - selection of several items regardless of their heirarchy;
- multiselect:"level" - selection of several items belonging to one and the same hierarchical level within the same tree branch.

```
Treetable Multidragging

webix.ui({
    view:"treetable",
    ..//treetable config
    multiselect:true,
    drag:true
})
```

RELATED SAMPLE: Multi Drag-and-Drop between Trees

Reorder mode works only with non-hierarchical components like datatable, list and x-list.

RELATED SAMPLE: DnD with List Items

Drag-and-Drop Context and Events

In essence, DnD is a a set of sequential events: first you hook on the necessary component item, then drag it to the desired position and drop the item releasing the mouse button.

Therefore, the component with draggable items as well as the one with a dropping ability, gets the following events:

- **onBeforeDrag** - fires before DnD is started;
- **onBeforeDragIn** -fires before a dragged item is moved over the dropping area;
- **onBeforeDrop** - fires when the dragged item is moved over the dropping area;
- **onAfterDrop** -fires after the dropping has happened;
- **onDragOut** - fires when a dragged item moves out of the dropping area.

They are used to control the DnD process and customize it on different stages since any event can trigger any custom function you'd like to associate this event with.

Functions attached to these events have **context** and **native event** as arguments.

Native event is a DOM event that happens during drag-and-drop while **context** is an object with the following properties:

- **from** - the source object;
- **to** - the target object;
- **source** - the ID of the dragged item(s);
- **target** - the ID of the drop target, null for drop on empty space;
- **start** - the ID from which DnD was started.

For instance, the **onBeforeDrop** can be used to make a copy a dragged item the moment it's dropped while not changing its place at all:

```
view:"datatable",
drag:true,
on:{
    onBeforeDrop:function(context, e){
        this.getItem(context.target).title = context.source.innerHTML;
//copying
        this.refresh(context.target);
        return false; //block the default behavior of event (cancels
```

```
dropping)
    }
}
```

RELATED SAMPLE: Drag-and-Drop from HTML

More about the possibilities of DnD event system

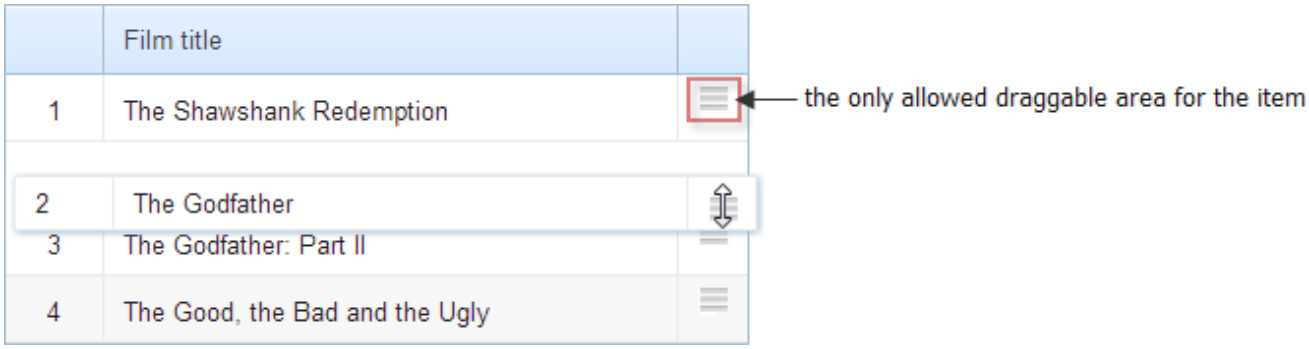
Drag-and-Drop - Advanced Level

- Defining Drag Handle - learn how to set a dragging handle to a component item;
- Defining Drag Marker - how to set a DnD message and mark by color the dropping area;
- DnD with Tree and TreeTable Components - how to observe hierarchy during DnD;
- DnD Between Different Components - how to use the **externalData** property while dragging an item from one component to another;
- Drag-and-Drop Events - how to use DnD event system for custom DnD, e.g. copying items with drag-and-drop;
- On-Page Drag-n-Drop (Advanced Level) - how to make any Webix view or HTML node on the page draggable and control every aspect of DnD;
- HTML5 Drag-n-Drop with Webix Views - how to use HTML5 DnD within Webix lib.

Advanced

Defining Drag Area

Predefined drag area is usedful when using drag-n-drop on **touch devices** as slide movements over drag area will be interpreted as dragging rather than scrolling.



RELATED SAMPLE: Drag and Drop with Drag Handle

By default any draggable item or component can be dragged by clicking and holding left mouse on any of its areas, but sometimes it looks more convenient to define a separate area that will be the only place you can click to drag the item.

Such area may be defined by icon on the item included into the item data template.

```
Create an icon

<style type="text/css">
    .webix_drag_handle{
    background-image:url(./handle.png); // path to the image icon
    background-repeat: no-repeat;
    cursor:n-resize; //type of the cursor to signify DnD ability
            }
</style>
```

```
Define drag area

webix.ui({
    view:"list",
    template: "#data#<div class='webix_drag_handle'></div>"
    on:{
        onBeforeDrag:function(data, e)
        return (e.target||e.srcElement).className == "webix_drag_handle";
        }
 });
```

OnBeforeDrag event triggers a function that sets 'webix_drag_handle' class for both target and source

elements.

Drag Area with Datatable

To define a drag area for Webix Datatable you should either add template to some column or add a separate column with drag area CSS:

```
view:"datatable",
drag:true,
columns:[
    { id:"title",   fillspace:true, header:"Film title" },
    { id:"drag", header:"", template:"<div
class='webix_drag_handle'></div>", width:35}
]
```

RELATED SAMPLE: Limiting the draggable area for items (dragging rows)

In case of **column drag-n-drop**, drag area can be defined as well:

```
view:"datatable",
dragColumn:true,
columns:[
    { id:"votes", header:"<div class='webix_drag_handle'></div>Votes",
width:120},
```

RELATED SAMPLE: Limiting the draggable area for items (dragging columns)

Related Articles

- Defining Drag Marker
- DnD with Tree and TreeTable Components
- DnD Between Different Component

Advanced

Defining Drag Marker

Drag Marker is a graphical element that appears in the dropping area and helps clearly see the position where the dragged element needs to be dropped.



RELATED SAMPLE: Custom Target Drag Marker

The look-and-feel of the drag marker is set via a div element with a **CSS class** where you set all its properties (positioning, borders, background color, marker text (if any), etc..).

```
.my_line_mark{
    border-top:1px dashed orange;
    width:100%; height:1px;
    position:absolute;
    left:0px;
    display:none;
}
```

The div element is placed into the document body:

```
var marker = webix.html.create("div", {
        "class":"my_line_mark"
}," ");
document.body.appendChild(marker);
```

The library features a set of functions for working with HTML. One of them, **create()**, makes an HTML div container with Javascript means, assigns a CSS to it while the content of the new HTML block is empty, which is signified by * *.

After that, you should extend the already inited component with the drag marker functionality. Here the **$dragMark** property comes to your help. Its value is a function that makes the marker visible as soon as the dragged element reaches the target area. Marker dimensions depend on the dimensions of an element being dragged:

```
webix.extend(treea, {
```

```
    $dragMark:function(context, ev){
        marker.style.display = 'none';
        if (context.target){
            var pos = webix.html.offset(ev.target); // position of the
target view
            var box = webix.html.offset(this.$view); // current position
of the dragged element

            marker.style.display = 'block'; // shows the marker
            marker.style.top = pos.y+"px"; // sets top offset of the
marker
            marker.style.left = box.x+"px"; //sets left offset of the
marker
            marker.style.width = this.$view.offsetWidth+"px"; //set the
marker width
        }
    }
}, true);
```

Comments:

- the function takes **DnD context** (described above) and native DOM event as parameters;
- initially the drag mark is not displayed and becomes visible only in the target area;
- firstly, properties of a dragged element are found out; then they define properties of the drag marker;
- **webix.html.offset** method gets the absolute top and left postioning of an HTML element specified as an argument;
- **offsetWidth** DOM method returns width of an element, including borders and padding.

Adding and Removing Marks

addCss() and **removeCss()** methods can help you apply any style for your drag marker defined as a CSS class. They are also used within a **$dragMark** function, but here you heedn't create a separate HTML container for the marker.

```
webix.extend(treeb, {
    $dragMark:function(context, ev){
        if (this.my_marked && this.my_marked != context.target){
            this.removeCss(this.my_marked, "my_custom_mark", true);
            this.my_marked = null;
        }
        if (context.target){
            this.my_marked = context.target;
            this.addCss(context.target, "my_custom_mark", true);
        }
    }
}, true);
```

- **my_custom_mark** is a CSS class for drag marker;
- a marker is added only to a target component (the mark becomes visible as you drag an item to the target);
- a marker is hidden the moment a dragged item leaves the target area as well as the moment you drop an item there.

Related Articles

- Defining Drag Area
- DnD with Tree and TreeTable Components
- DnD Between Different Component

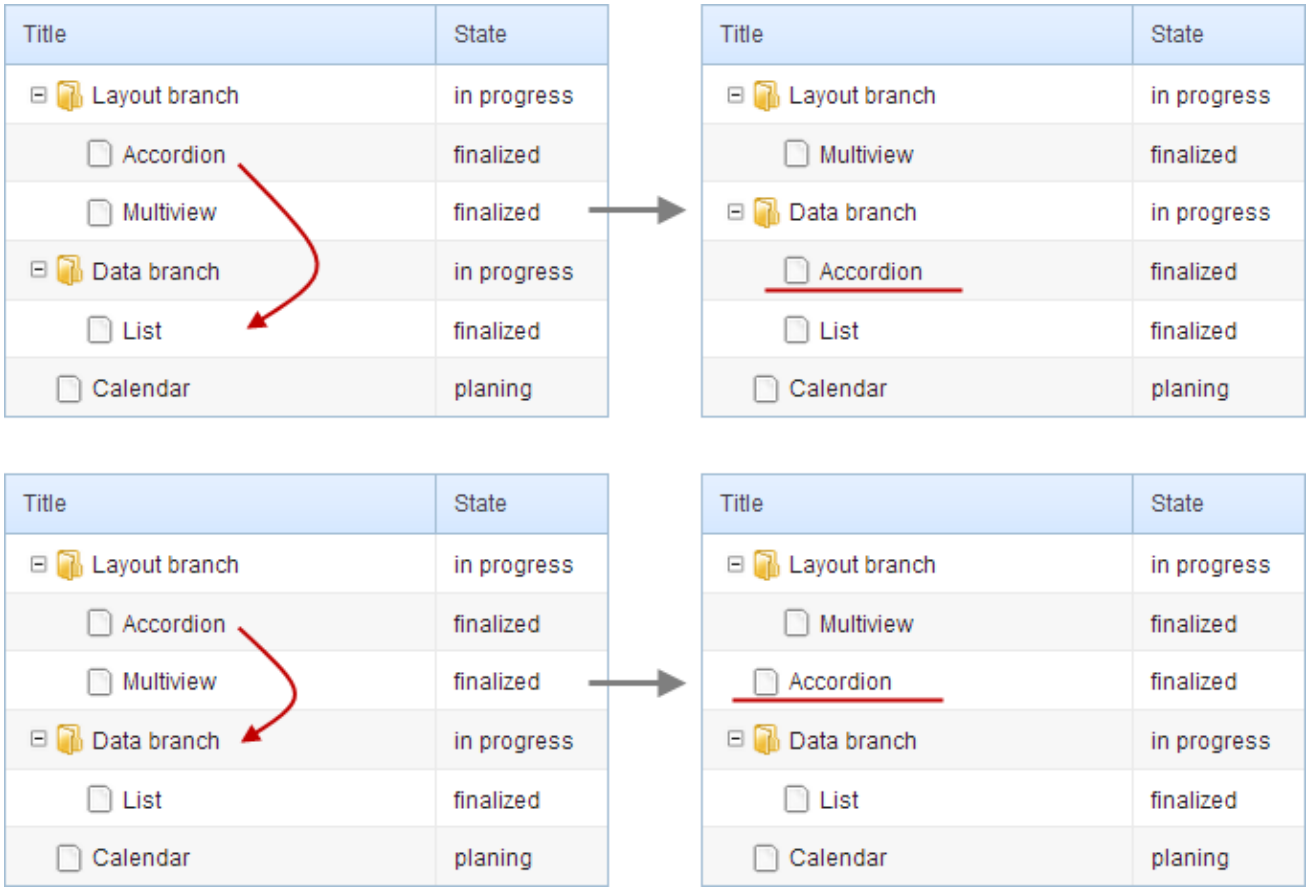
Advanced

Drag-and-Drop with Tree and TreeTable

In case of tree and treetable where data is presented in a hierarchical way, it's vital to specify how tree nodes would be dragged between different levels.

By default, if a tree node is dragged into the branch of the same level as its parent, it becomes the first child of this branch. At the same time, if you drop lower-level nodes to the higher-level branches (the mouse is released over this higher-level node) the dragged node takes its level and become the previous child of this node.)

In addition, if you drag-and-drop the tree branch it is moved together with its leaves.



RELATED SAMPLE: Basic Drag-and-Drop between Trees

Event handling within DnD can help you change dragging as well as dropping pattern.

For instance, if you want to prevent DnD on a top level items as well in leaf items, you should block it before the dragged element reaches the dropping area:

```
on:{
    onBeforeDragIn:function(config, id){
        if (!id) return false;                          //block dnd on top
level
        if (!this.getItem(id).$count) return false; //block dnd in leaf
```

```
items
}}
```

Dropping process can be as well customized. To enable dropping of the nodes into a tree branch as its children you should state that the dropping target is a parent branch.

```
on:{
    onBeforeDrop:function(context){
        context.parent = context.target;    //drop as child
        context.index = -1;                 //as last child
}}
```

RELATED SAMPLE: Drag-and-Drop between Tree Nodes - Drop as Child

The posiition of a dropped item is set in the following way: the item becomes the first child if it is dropped over an opened tree branch; in other cases, e.g. the item is dropped over the leaves of a tree branch is takes the next position relative to the child it is dropped over:

```
on:{
    onBeforeDrop:function(context){
        if (this.getItem(context.target).$count &&
this.getItem(context.target).open){
            //drop as first child if the item is dropped into an opened
tree branch
            context.parent = context.target;
            context.index = ;
        } else {
            //in other cases, the dropped item takes the next next
position
            context.index++;
        }
    }
}
```

RELATED SAMPLE: Drag-and-Drop between Trees - Drop Next

Related Articles

- Defining Drag Area
- Defining Drag Marker
- DnD Between Different Component

Advanced

Drag-and-Drop between Different Components

External Data Functionality

During drad-n-drop operations between different components as well as different instances of a component it's vital to ensure that data items from the source component receive adequate representation in the target one.

externalData property features a function responsible for dragging as value (in case you work with different components).

```
{   view:"tree",
    drag:true,
    select:true,
    data:[],
    externalData:grid2tree
};
```

The property function takes two parameters - **data** and **id:**

- **data** (object) - the data item to move;
- **id** (string) - the item id in the source element.

When moved to the target component the item retains the ID it had in the source component unless the target already has an item with this ID. In this case, the ID will be set randomly while the initial ID will be stored in the function.

Take for instance, an app containing two trees, a dataview and datatable. The tree dataset includes items with **data.value** items while dataview and datatable have **data.title** and **data.rank**. To enable drag-n-drop within the whole group, you need two functions:

1 . The function that will populate dataview and datatable.

```
function tree2grid(data, id){
    data.rank = data.rank || -1;
    data.title = data.title || data.value;
    return data;
}
```

Notice the "or" logic here:

- If you drag data from datatable to dataview and vice versa, native data and external data have the same parameters, so *data.title* value from the datatable becomes *data.title* of the dataview. The same is true about *data.rank*.
- If you drag data from any of the trees to datatable and dataview, you need to redefine *data.title* parameter and take its value to the **data.value** of the tree. Then, since in the sample data for both trees has no *data.rank* parameter, its value will be displayed as "-1".

2 .The function that will populate tree with dragged data.

```
function grid2tree(data, id){
    data.value = data.value || data.title;
    return data;
}
```

Here only *data.value* matters. The tree will take the dragged-and-dropped *data.value* value from the other tree, or the *data.title* value from either fo the datatables.

RELATED SAMPLE: Basic Drag-and-Drop

Related Articles

- Defining Drag Area
- Defining Drag Marker
- DnD with Tree and TreeTable Components

Advanced

Drag-and-Drop Events

DnD event system allows you to customize the process of drag-and-drop and adjust it to your needs.



Copying with DnD

Basically, drag-and-drop of component items means that you remove an item from one component and place it to the other one. Still if needed, you can **copy items with DnD** by actually leaving a dragged item in the source component and making its copy in the target one.

For these needs, we refer to the **onBeforeDrop** event and associate a copying function with it. Also, we make use of **source** and **from** properties of the DnD context object.

```
gridb.attachEvent("onBeforeDrop", function(context, ev){
    for (var i=; i< context.source.length; i++){
context.from.copy(context.source[i],context.start,this,webix.uid());
    }
    return false;
});
```

Here we **copy** each item (that is dragged) from the source object with the help of the dedicated function. It takes the following arguments:

- **context.source[i]** - the ID of the copied object;
- **context.start** - the index of the copy;
- **this** - the object that an item is copied to (since the fucntion is attached to **gridb**, **this** refers to it)
- **webix.uid()** -the ID of the copy-item, here it's generated automatically with the tool for getting an unique (in frames of a session) ID.

Displaying Titles of Dragged Items

In case of datatable, the whole rows can be dragged to other components as well as moved between

datatable instances.

By default, the whole row is displayed while being dragged, which is not always convenient and pleasant to see. Sometimes, it would be enough to display the value of the main column during DnD. The same happends with dataview and list, the components that feature several template values in each item.

Here, the **onBeforeDrag** event is needed as the functionality should be enabled as soon as we hook on the necessary item. Also, as above, we take **source** and **from** properties from the DnD context object.

```
grida.attachEvent("onBeforeDrag", function(context, ev){
    context.html = "<div style='padding:8px;'>";
    for (var i=; i< context.source.length; i++){
        context.html += context.from.getItem(context.source[i]).title +
"<br>" ;
    }
    context.html += "</div>";
});
```

Here we create a **div** container for the DnD context item and define 8-px padding for it. Then, with each of the dragged items from the source we take a **title** value from the dataset and close a *div* container.

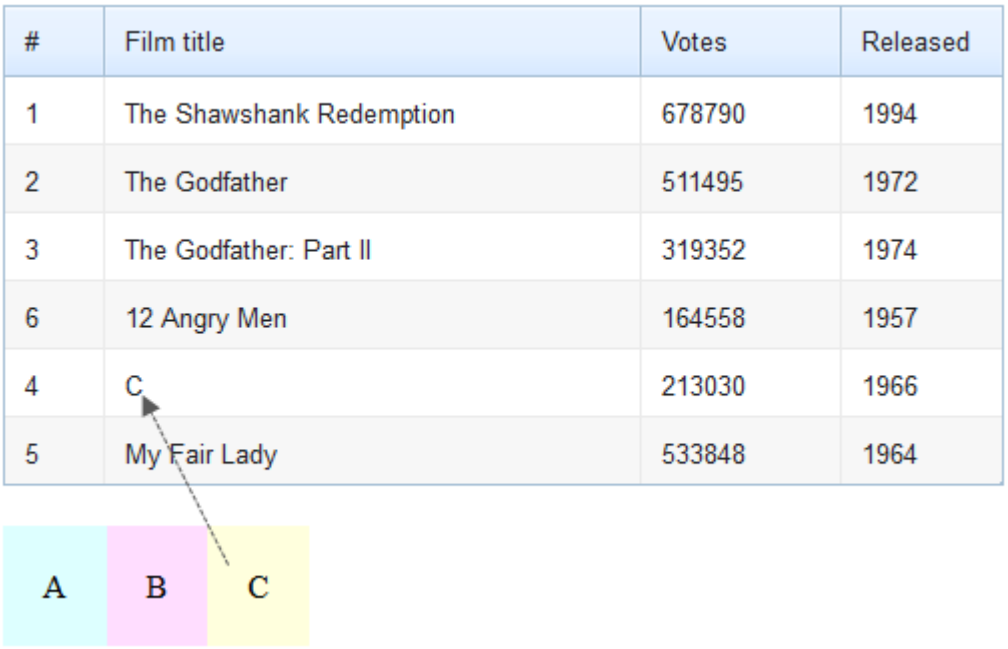
RELATED SAMPLE: Using events for customizing dnd

Related Articles

- Defining Drag Area - learn how to set a dragging handle to a component item;
- Defining Drag Marker - how to set a DnD message and mark by color the dropping area;
- DnD with Tree and TreeTable Components - how to observe hierarchy during DnD;
- DnD Between Different Component - how to use the **externalData** property while dragging an item from one component to another;

HTML5 DnD with Webix Views

HTML5 brought native drag-n-drop techniques that can be used as well for Webix views. Here we study how to drag html nodes to a Webix datatable using HTML5 dnd.



RELATED SAMPLE: HTML5 Drag-and-Drop handling

Firstly, we make the needed HTML nodes (here - colored divs below grid) **draggable** and assign a function to fire when you start dragging them:

```
<div id="divA" draggable="true" ondragstart="drag(event)" > A </div>
```

And **enable drag-n-drop** in both directions (drag from here and drop here) within a Webix grid:

```
view:"datatable",
..config..,
drag:true
```

Secondly, we need to set which data matters during DnD. In HTML5 data is transferred with the help of (surprisingly) **dataTransfer** object. We need the following of its methods:

- **setData()** - to set transferred data as *{key1:value1, key2:value2}*;
- **getData(key)** - to get the needed data item by its key;

From HTML nodes we take their text values (innerHTML);

```
function drag(ev){
    ev.dataTransfer.setData("title",ev.target.innerHTML);
}
```

When the dragged node is dropped on the target grid item, we **get the data from DataTransfer** and parse it into the item.

HTML5 events are handled with event method that takes **node** or its **ID**, **event type**, **event handler** and **"this" object** as parameters. Event names are passed without **on** prefix:

```
webix.event(mygrid.$view, "drop", function(e){
    var grid = $$(e); //e - id of drop target
    var id = grid.locate(e); // return ID of target grid item
    grid.getItem(id).title = e.dataTransfer.getData("title"); //setting
new title
    grid.refresh(id); //make the changes visible
});
```

At the same time, we need to prevent default drag-n-drop behavior, if we want HTML nodes to keep their position (in fact, copying of their innerHTML takes place);

```
webix.event(mygrid.$view, "dragover", function(e){ e.preventDefault(); });
```

HTML5 Drag-n-Drop can be studied here (Mozilla Developer Network).

Intermediate

Working with the clipboard

Webix provides support for the clipboard operations in all data-containing components. For this purpose, the library supplies the CopyPaste mixin.

By default, the support is disabled. To enable it, set the clipboard property to *true* :

```
Enabling the clipboard support

webix.ui({
    view:"list",
    ...
    clipboard: true,
});
```

RELATED SAMPLE: List: Work with Clipboard

In 3 components this mixin is already included. These components are:

- DataTable (*has extended support. See details*);
- Tree (*has extended support. See details*);
- List.

If you want to work with the clipboard in any other data-containing component, you should extend it with CopyPaste class. For example, to add the support to DataView use the technique shown below:

```
var webix.ui({
    view:"dataview",
    id:"myview",
    ...
    clipboard: true,
});

webix.extend($$("myview"),webix.CopyPaste);
```

Custom 'copy' function

To customize the way that the data will be stored in the clipboard (when the user presses CTRL+C) you should use the templateCopy property. In that case, before transfering to the clipboard the copied data will be modified according to the specified template.

To use the the templateCopy property you need to set the clipboard property to a value 'custom'.

Let's take a simple example: you have a list that displays the names of your favourite films. Assume, in the clipboard besides the film title you want to store its year and rating.

Then:

- Set the clipboard property to the 'custom' value;
- Specify the templateCopy property like this (as a parameter, the template function takes the data item object):

```
var films_set = [
    { id:1, title:"The Shawshank Redemption", year:1994, votes:678790,
rating:9.2},
    { id:2, title:"The Godfather",           year:1972, votes:511495,
rating:9.2},
    { id:3, title:"The Godfather: Part II",   year:1974, votes:319352,
rating:9.0},
    { id:5, title:"My Fair Lady",            year:1964, votes:533848,
rating:8.9}
];

webix.ui({
    container:"box",
    view:"list",
    data:films_set,
    template:"#title#",
    clipboard: "custom",
    templateCopy: function(item) {
        return item.title+" (year: "+ item.year+", rating:
"+item.rating+")";
    }
});
```



RELATED SAMPLE: List: Work with Clipboard

Custom 'paste' function

To customize the way that the data will be retrieved from the clipboard to a Webix component you should use the onPaste event.

The event fires when the user presses Ctrl+V keys combination and as a parameter, the handler function takes the text stored in the clipboard.

To use the the onPaste event you need to set the clipboard property to a value 'custom'.

```
var films_set = [
    { id:1, title:"The Shawshank Redemption", year:1994, votes:678790,
rating:9.2},
    { id:2, title:"The Godfather",           year:1972, votes:511495,
rating:9.2},
    { id:3, title:"The Godfather: Part II",   year:1974, votes:319352,
rating:9.0},
    { id:5, title:"My Fair Lady",            year:1964, votes:533848,
rating:8.9}
];

webix.ui({
    view:"list",
    id:"mylist",
    ...
    clipboard: "custom"
});
$$("mylist").attachEvent("onPaste", function(text) {
    text = text.split(" , ");

    var sel = this.getSelectedId(true);
    for (var i = ; i < sel.length; i++) {
        var item = this.getItem(sel[i]);
        item.title = text[];
        item.year = text[1];
        item.votes = text[2];
        item.rating = text[3];
        this.refresh(sel[i]);
    }

});
```

RELATED SAMPLE: List: Work with Clipboard

Clipboard Events

- **onPaste** - fires when Ctrl+V shortcut is pressed:

```
tree.attachEvent("onPaste", function(text) {
        webix.message("Node is pasted: " + text);
});
```

RELATED SAMPLE: Custom Clipboard Operations

Grouping Data Items

Grouping Types

Grouping data in Webix components can be obligatory and optional, static and dynamic.

Obligatory grouping

Grouping is a must-have feature for components that present data hierarchically yet are populated with non-hierarchical (inline) dataset.

Such components are subject to initial static grouping. Each group features a group header(parent branch) that can be opened to show the group members.

- grouplist;
- tree;
- treetable.

When they are fed with hierarchical data grouping is not necessary. Read more about data types.

Optinal grouping

Data can be groupped if you need to present data united by some criterion.

- chart;
- datatable.

With these componnents you can see only the result of grouping.

In case of inline dataset:

- the initial grouping for list makes it grouplist;
- the initial grouping for datatable makes it treetable. Still, datatable can be dynamically grouped.

However, they are separate components - **groulist** and **treetable** - and should be created with their own names.

Static/Initial grouping

Grouping criteria are set **during component init** by the **$group** key of the **scheme** object. Necessary for the components with obligatory grouping.

```
webix.ui({
```

```
    view:"tree",
    scheme:{
        $group:"year"
    }
});
```

RELATED SAMPLE: Tree: Plain JSON Data with Client-side Grouping

Dynamic grouping

Grouping is defined by a function applied to the component.

```
$$("mytree").group("year");
```

To **cancel grouping**, make use of the opposide method.

```
$$("mychart").ungroup();
```

Setting Grouping Pattern

Syntax for **group()** method and **$group** object property is the same - you need to specify here the grouping critetion as:

- **data item;**
- **function;**
- **object.**

Data Item

String that points to the property from the initial dataset;

```
//static mode
tree = webix.ui({
    view:"tree",
    scheme:{
        $group:"year"
    }
});

//dynamic mode
tree.group("title");
```

Function

The function is applied to data object and returns grouping criterion(-a);

```
scheme:{
    $group:function(obj){
```

```
        return obj.year + ..;
    }
}
```

RELATED SAMPLE: List: Grouping

Object

An object with properties defining grouping criterion and grouping pattern:

- **by** (must-have) - set the grouping criterion either as *string* or as *function*.
- **map** - an object that redefines values to display in the current template. Here math **functors** are applied to the specified properties of group members.
- **footer** (for treetable) - defines template for extra table line that displays common property of group members.
- **row** (for datatable and treetable) - defines template for the whole row regardless of initial columns.

```
//static mode
webix.ui({
    view:"chart",
    scheme:{
        $group:{
            by:"year",
            map:{...}
        }
    }
});

//dynamic mode
grida.group({
    by:"year",
    map:{...}
});
```

If you use only **by** property in the group object, it is the same as if you define criterion with a data item or function. Hence, the following definitions are **equal**:

```
$group:"year"

$group: function(obj){ return obj.year;}

$group:{ by:"year" }

$group:{
    by: function(obj){ return obj.year }
}
```

The function can be of any complexity:

```
by: function(obj){ return Math.floor(obj.year/10); }
```

RELATED SAMPLE: Static grouping

Grouping Functors

Grouping functors are used within **map** and **footer** that redefine templates.

```
$group:{
    by:"year",
    map:{
        property(template):['property', functor]
    }   //or property:[function (obj){...}]
}
```

Grouping functors:

- **any** to show all values for this property; the same effect will be set if you **omit functor** at all.
- **sum** to calculate and show total sum of the values;
- **max** to pick the maximum value of the property;
- **min** to pick the minimum value of the property;
- **count** to calculate the number of items in each group;
- **string** to show property name as a string rather than its value. For *{year:1994}*, the word "year" witll be displayed in the component.
- **custom functor** for other math operations.

Learn more about grouping functors.

Templates for Groups and Items

By default, component items display object **value** which is:

- **value** property of the datasource;
- value from the dataset defined by template;
- value of an array element (if data comes in JSArray);

For items that display a **group headers** (parent branches), value is the one defined by **by** criterion:




```
webix.ui({
    view:"tree",
    scheme:{
        $group:{
            by:"year" // group header value
        }
    },
    data: [  //default value
    { id:1, value:"The Shawshank Redemption", year:1994, votes:678790,
rating:9.2, rank:1}, {...}]
});
```

RELATED SAMPLE: Tree: Plain JSON Data with Client-side Grouping

If a component has a template that sets other property as value for its items, you need to define **map** to redefine this value during grouping:

```
webix.ui({
    view:"tree",
    template:'{common.icon()} {common.folder()}#title#',
    scheme:{
        $group:{
            by:"year",
            map:{
                title:['year', 'any']
            }
        }
    }
});
```

Initially, template allows showing only titles. Map tells that group **title** is **year** and makes year the value to show.

Each component has its own peculiarities concerning templates in a grouped mode. Study them

below.

In addition, you can chanage data to display by defining a function for the **"onGroupCreated"** event.

Datatable and Treetable Specificity

For datatable and treetable data templates are set by column IDs or column template or both.

Yet, in grouped mode, the grids won't show data for those columns which templates are not set either by template function or by map.

Map can redefine more than one property.

```
map:{
    votes:["votes", "sum"], //"votes" values will be summed
    title:["year"] //title will take year values
}
```

RELATED SAMPLE: Aggregating data while grouping

Function templates allow for setting different presentation ways for group items and group headers.

- **obj.$group** - stores item state (whether it is a group item or group header)
- **obj.$level** - stores hierarchy level with upper being 1.

The two snippets are equal:

```
template: function(){
    if(obj.$group) return ...; //for group headers
    return ... //for group items
}


template: function(){
    if(obj.$level==1) return ...; //for group headers
    return ... //for group items
}
```

Treetable with map defined:

```
columns:[
    { id:"title", template:function(obj, common){
            if (obj.$group) return common.treetable(obj, common) +
obj.title;
            return obj.title;
            }
    }
],
```

```
scheme: {
    $group:{
        by: "year",
        map:{
            title:["year"] //year is 'obj.title' for group header
        }
    }
}
```

RELATED SAMPLE: Aggregating data while grouping

Treetable with unset map:

```
columns:[
    { id:"title", template:function(obj, common){
        if (obj.$level == 1) return common.treetable(obj, common) +
obj.value;
        return obj.title;
        }
    }
],
scheme:{
    $group:"year" //year is obj.value for group header
}
```

RELATED SAMPLE: Static grouping

Row Templates and Additional Rows

Row template

Row property within grouping object sets group header template for the whole row regardless of grid columns.

```
grida.group({
    by:"year",
    map:{
        "year":["year",  "any"],
        "votes":["votes", "sum"]
    },
    row:function(obj){
        return "Year "+obj.year+", total votes:
"+webix.i18n.numberFormat(obj.votes);
    }
});
```

RELATED SAMPLE: Data aggregation in Custom Format

Group Footer

Group footer display common value for all the members of this group, e.g. total sum, max or min value, etc.



Footers are defined in the same way as **map** and includes the chosen **property** treated with a functor and a template for the **footer row**.

```
grida.group({
    by:...,
    map:{...},
    footer:{
        votes:["votes", "sum"],
        row:function(obj ){ return "<span style='float:right;'>Total:
"+webix.i18n.numberFormat(obj.votes)+"</span>"; }
    }
});
```

RELATED SAMPLE: Dynamic grouping: Extra lines

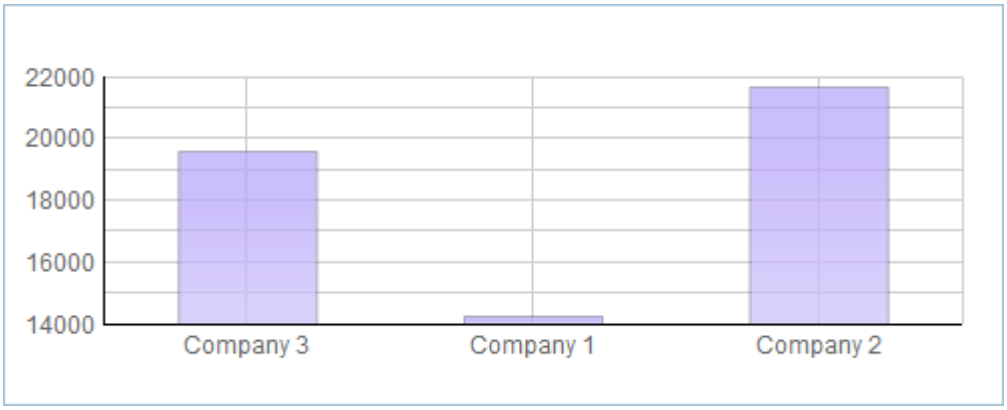
Chart Specificity

For grouped chart grouped items and group headers correspond to each of its axes

- X-axis values display grouping criterion (introduced by **by**);
- Y-axis values display the chosen values of group items treated with a functor.

For instance, if you need to display sells of each company, you should group all sales values by company while calculating the sum for each company.

```
webix.ui({
    view:"chart",
    value:"#sales#", //default template for Y axis
    scheme:{
        $group({
            by:"company", //template for X axis
            map:{
                sales:["sales","sum"]
            } //Y axis template now shows sum of all sales for each year
        })
    }
})
```



RELATED SAMPLE: Chart: Grouping

The first element in the map is a data item property while the second one is a **functor** applied to all the values of this property.

Grouplist Specificity:

Grouplist features ites own properties:

- **templateItem** - to set template for items in a group;
- **templateGroup/templateBack** - to set template for group header.

See grouplist article for details.

Grouping Events

- **onGroupCreated** - fires when grouping is applied to the component and takes the following parameters:
  - **id** - ID of the group header (parent branch);
  - **value** - value of the grouping criterion;
  - **data** - component data object;
- **onGroupFooter** - fires when grouping is applied to the component and a footer for each group is provided.

```
webix.ui({
    view:"treetable",
    on:{
        "data->onGroupCreated":function(id, value, data){
            this.item(id).value = "Year "+value;
        }
    }
});

grida.group({
    by:"year" //obj.value = year
```

```
});
```



RELATED SAMPLE: Handling events while data grouping

Custom Functor for Data Grouping

Functor is a method that is applied to an item property in order to define the way it is presented in a group header.

```
[
    { id:1, title:"The Shawshank Redemption", year:1994, votes:678790},
    { id:2, title:"The Godfather", year:1972, votes:511495},
    { id:3, title:"The Godfather: Part II", year:1994, votes:319352},
    { id:4, title:"The Good, the Bad and the Ugly", year:1972,
votes:213030}
]
```

When data is grouped by year, we may want to learn total number of votes per year, or their maximim/minimal value, etc. It is **grouping functors** that are designed for this purpose. They are used within **map** object of the grouping function.

```
$group:{
    by:"year",
    map:{
        votes:['votes', 'sum']
    }   //'sum' is a functor to calculate votes total for each year
}
```

Initial Datatable:

	Film title	Released	Votes
1	The Shawshank Redemption	1994	678.79
2	The Godfather	1972	511.50
3	The Godfather: Part II	1974	319.35
4	The Good, the Bad and the Ugly	1966	213.03

Grouped Datatable:

	Film title	Released	Votes
	1974		606.57
	1975		587.52
	1976		387.80
	1981		385.44

RELATED SAMPLE: Data aggregation

Simplest operations are supplied by prebuilt grouping functors stored in the GroupMethods class.

However, you can a create custom functor to meet your needs.

Adding a New Functor

Any function can become a grouping functor provided that you've given it the **unique name**, described the logic and added to the GroupMethods class.

Such function takes two arguments:

- **prop** - the property to process;
- **data** - data object with items of this group.

There's no functor to find out the average value for item ptoperties, so let's call it **median** and add it.

```
webix.GroupMethods.median = function(prop, data){
                if (!data.length) return ;
                var summ = ;
                for (var i = data.length - 1; i >= ; i--) {
                    summ += prop(data[i])*1;
                };
                return summ/data.length;
            };
```

And apply the newly created functor referring to it by its name:

```
$group:{
    by:"year",
    map:{
        votes:['votes', 'median']
    }
}
```

RELATED SAMPLE: Custom aggregation while grouping

The functor can be as well defined **bypassing GroupMethods class** by setting a custom fucntion.

```
function getAverage(prop, data){
                //..the same as for 'median' above
}

$$("mychart").group({
        by:"year",
        map:{
            sales:["sales", getAverage]
        }
});
```

RELATED SAMPLE: Chart: Grouping

By the way, you can omit the functor yet set the way grouping criterion will be presented:

```
webix.ui({
    view:"treetable",
    columns:[ //default template for ungrouped items
        { id:"title", template:"{common.icon()} #title#" },
    ],
    scheme:{
        $group:{
            by:function(obj){ return Math.floor(obj.year/10); },
            map:{
                title:[function(obj){
                    var min = obj.year - obj.year%10;
                    return min + " - "+ (min+10)
                }]
            }
        }
});
```



RELATED SAMPLE: Static grouping

Styling and Animation

- Skins
- Component Styling
- Animation
- Icons with UI Components
- Setting Borders to the Components

Skins

Webix provides 11 skins that you can use to change the overall look of your application:

- Flat (default skin)
- Compact
- Material Design
- Touch

- Air
- Air Compact
- Clouds
- Glamour
- Light
- Metro
- Terrace
- Web

Each skin requires including its specific CSS file.

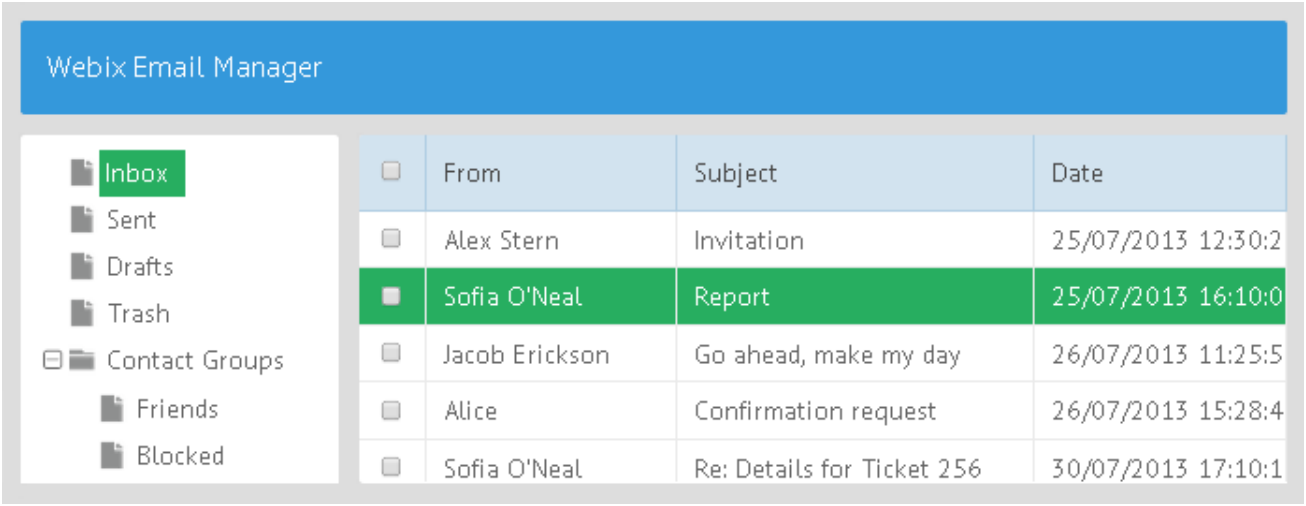
All CSS files for skins reside in the **codebase/skins** folder of the package.

Flat

To apply the **Flat** skin, include the default CSS file:

```
Applying the 'Flat' skin to the app

<script src="../codebase/webix.js"></script>
<link href="../codebase/skins/webix.css" rel="stylesheet" type="text/css">
```

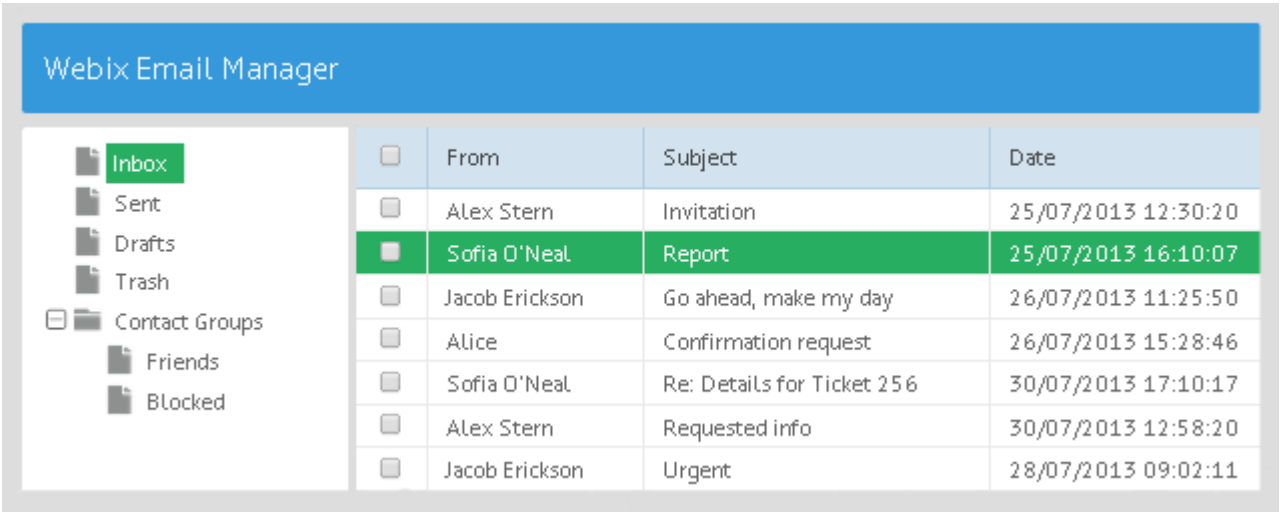


Compact

To apply the **Compact** skin, include the *compact.css* file:

```
Applying the 'Compact' skin to the app

<script src="../codebase/webix.js"></script>
<link href="../codebase/skins/compact.css" rel="stylesheet"
type="text/css">
```



Material Design

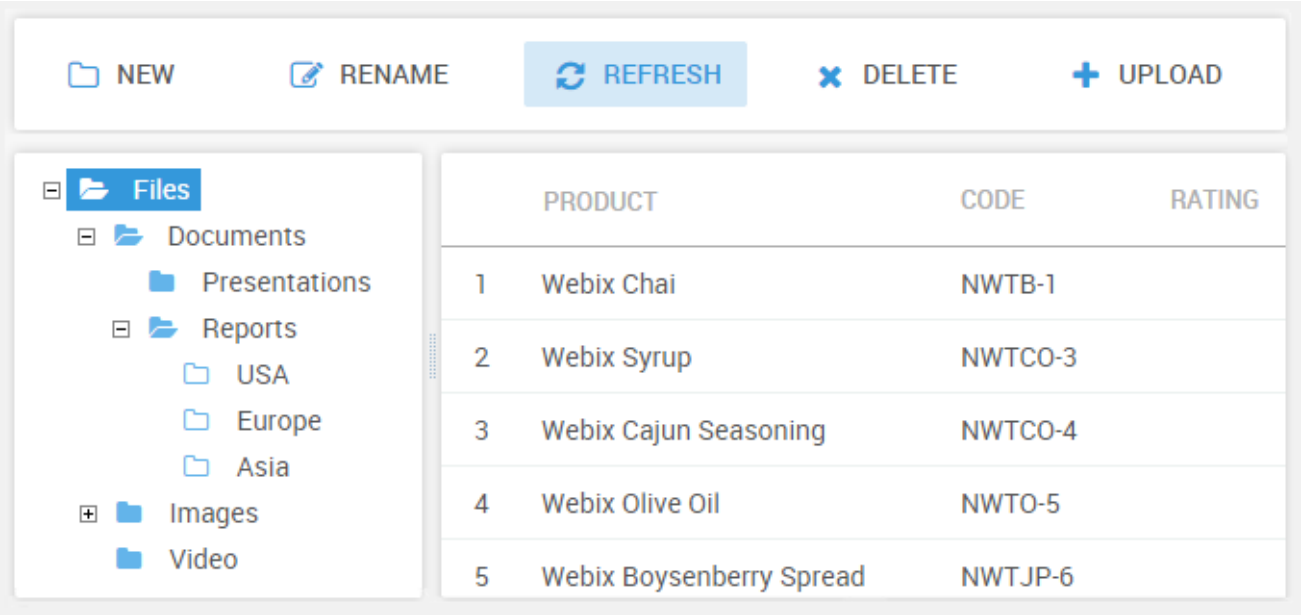
To apply the **Material Design** skin:

- download it from a separate repository at github.com
- copy files into *codebase* folder of Webix UI package

- include the *material.css* file:

```
Applying the 'Material Design' skin to the app

<script src="../codebase/webix.js"></script>
<link href=".../codebase/skins/material.css" rel="stylesheet"
type="text/css">
```



You can read a more detailed article about the usage of Material Design skin.

Touch

To apply the **Touch** skin, include the *touch.css* file:

```
Applying the 'Touch' skin to the app

<script src="../codebase/webix.js"></script>
<link href="../codebase/touch.css" rel="stylesheet" type="text/css">
```

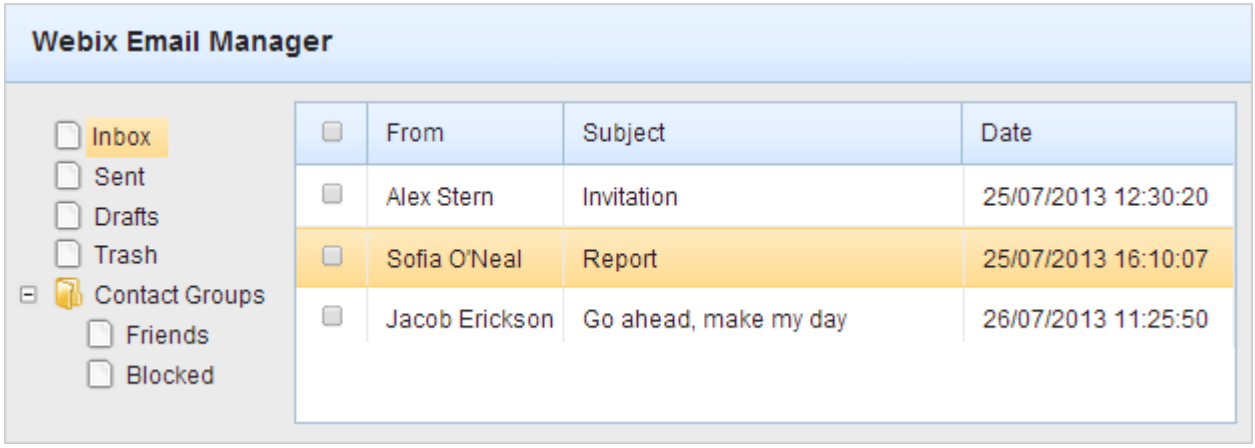


Air

To apply the **Air** skin, include the *air.css* file:

```
Applying the 'Air' skin to the app

<script src="../codebase/webix.js"></script>
<link href="../codebase/air.css" rel="stylesheet" type="text/css">
```

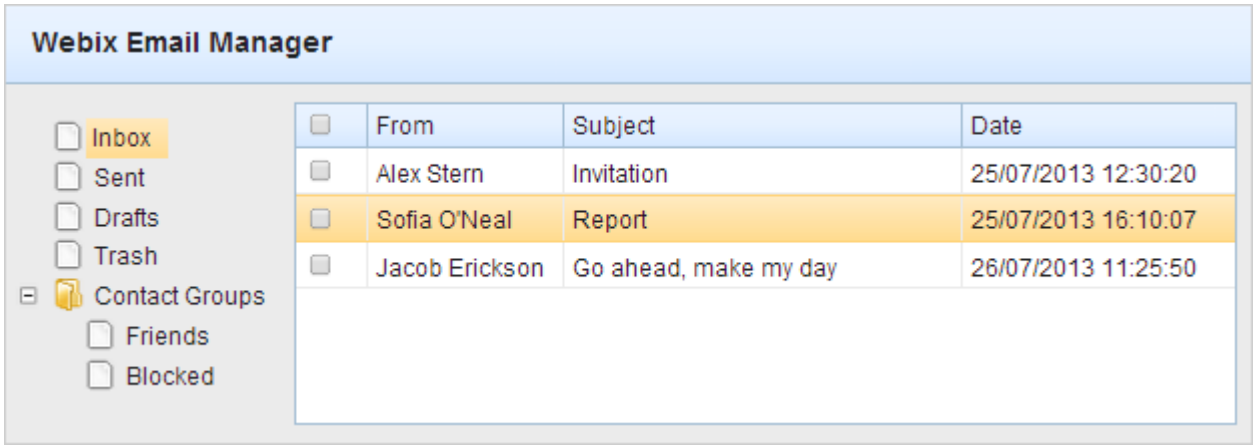


Air Compact

To apply the **AirCompact** skin, include the *aircompact.css* file:

```
Applying the 'AirCompact' skin to the app

<script src="../codebase/webix.js"></script>
<link href="../codebase/skins/aircompact.css" rel="stylesheet"
type="text/css">
```

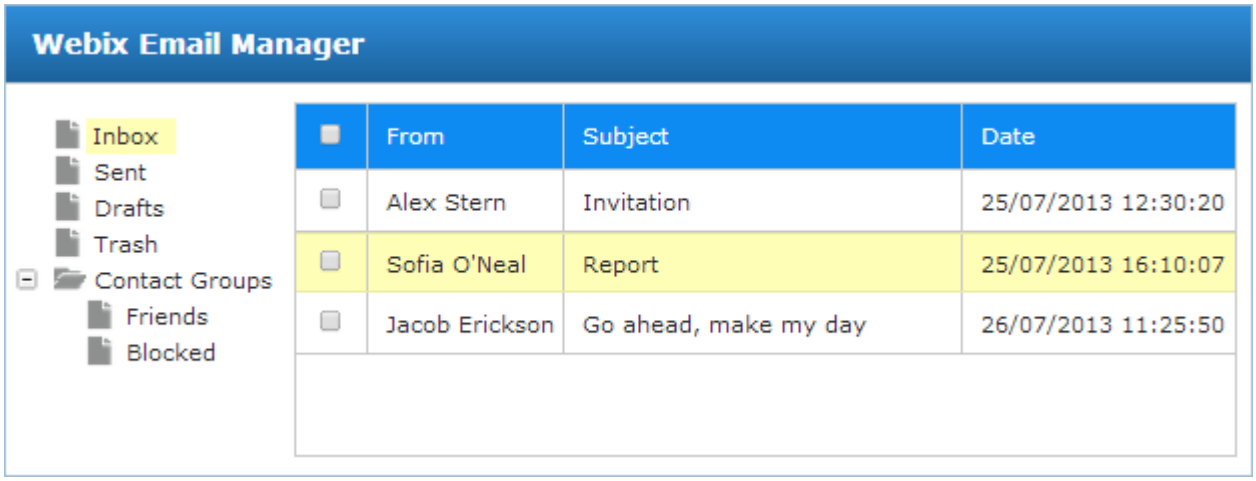


Clouds

To apply the **Clouds** skin, include the *clouds.css* file:

```
Applying the 'Clouds' skin to the app

<script src="../codebase/webix.js"></script>
<link href="../codebase/skins/clouds.css" rel="stylesheet"
type="text/css">
```

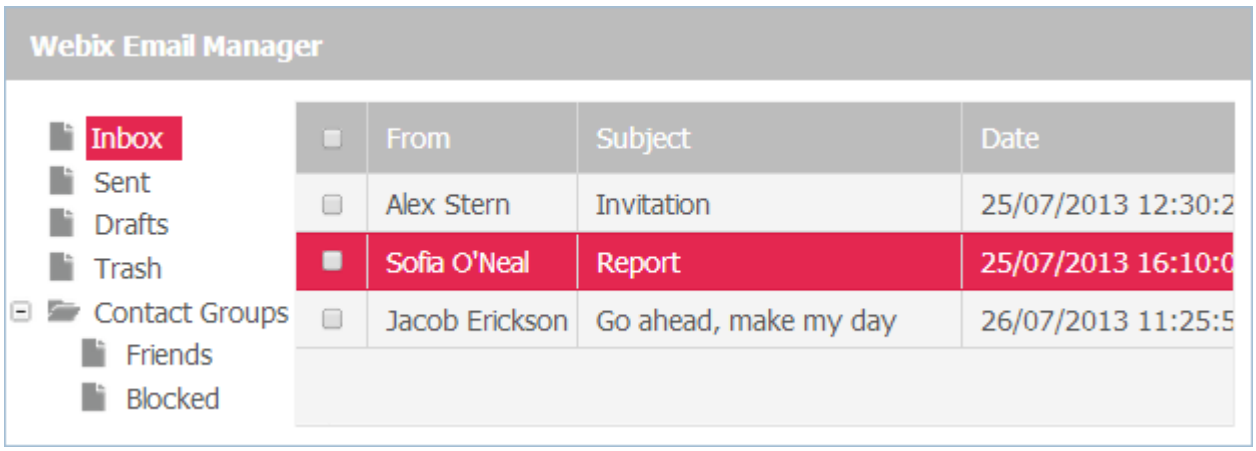


Glamour

To apply the **Glamour** skin, include the *glamour.css* file:

```
Applying the 'Glamour' skin to the app

<script src="../codebase/webix.js"></script>
<link href="../codebase/skins/glamour.css" rel="stylesheet"
type="text/css">
```

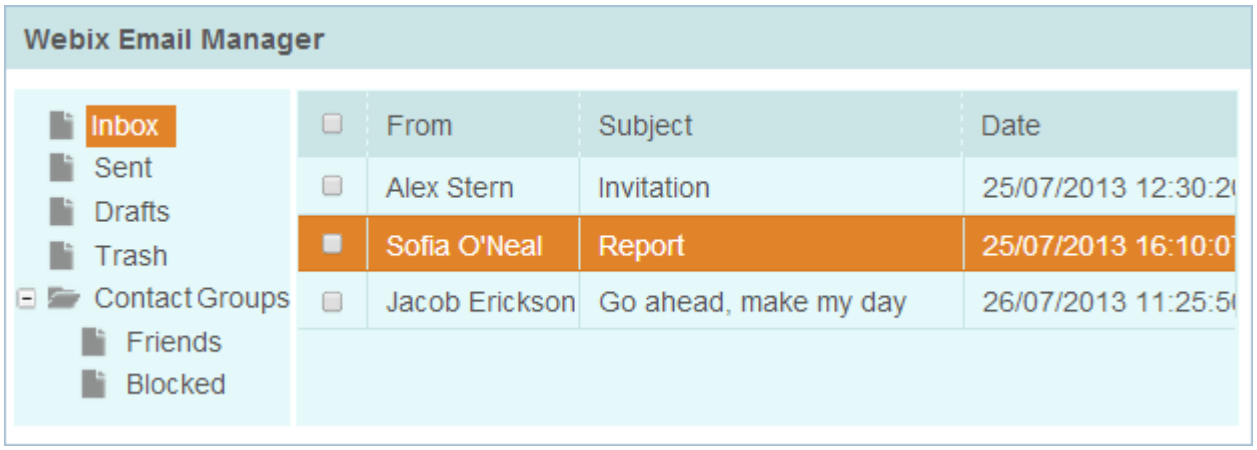


Light

To apply the **Light** skin, include the *light.css* file:

```
Applying the 'Light' skin to the app

<script src="../codebase/webix.js"></script>
<link href="../codebase/skins/light.css" rel="stylesheet" type="text/css">
```

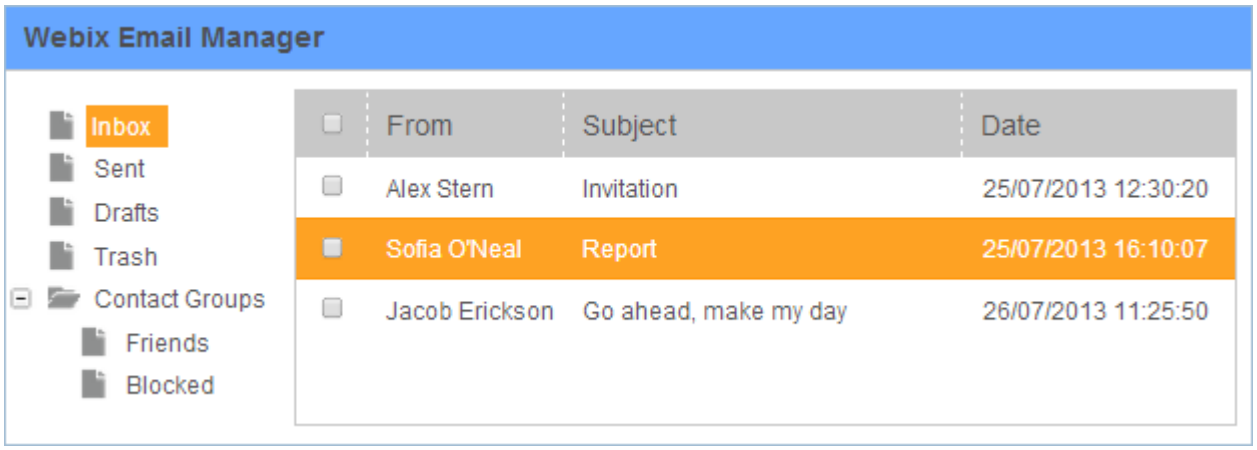


Metro

To apply the **Metro** skin, include the *metro.css* file:

```
Applying the 'Metro' skin to the app

<script src="../codebase/webix.js"></script>
<link href="../codebase/skins/metro.css" rel="stylesheet" type="text/css">
```

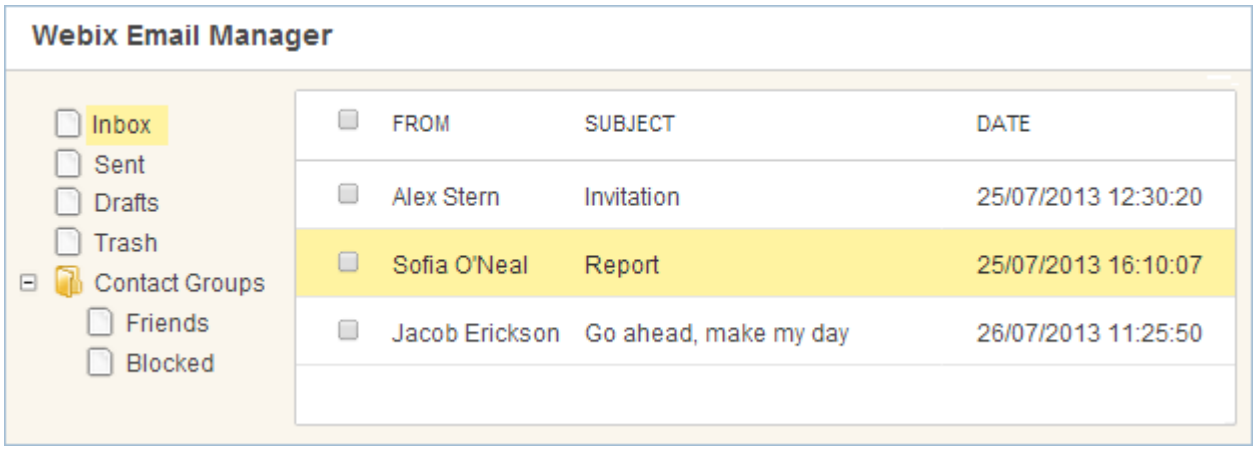


Terrace

To apply the **Terrace** skin, include the *terrace.css* file:

```
Applying the 'Terrace' skin to the app

<script src="../codebase/webix.js"></script>
<link href="../codebase/skins/terrace.css" rel="stylesheet"
type="text/css">
```

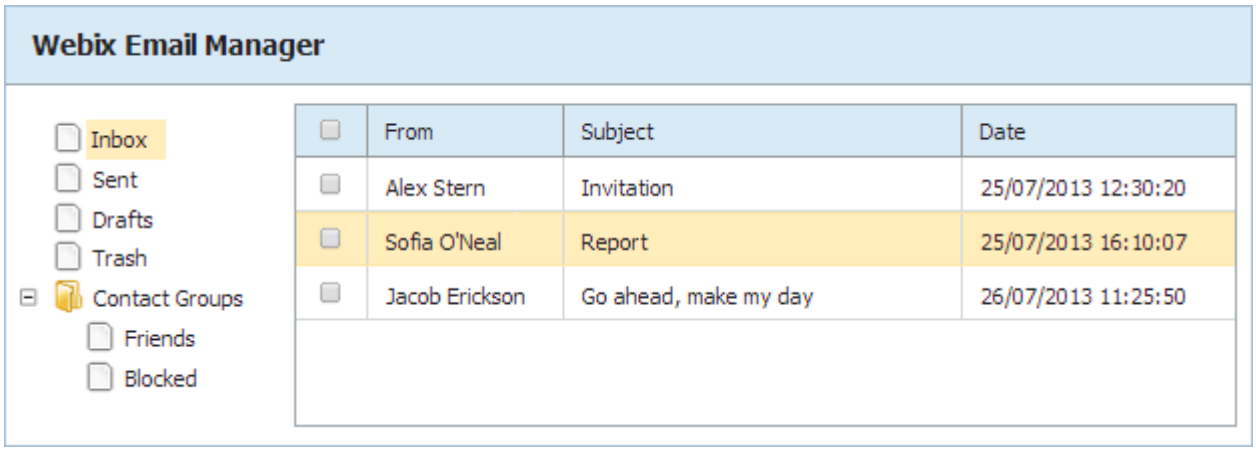


Web

To apply the **Web** skin, include the *web.css* file:

```
Applying the 'Web' skin to the app

<script src="../codebase/webix.js"></script>
<link href="../codebase/skins/web.css" rel="stylesheet" type="text/css">
```



Material Design Skin

Skin works with Webix 2.3+. It is not inluded in the package and should be downloaded from GitHub.

You can see live demo by this link.

How to use

- copy files into the *codebase* folder of Webix UI package
- use skin file on a page as any other skin

```
<html>
<head>
    <script type="codebase/webix.js"></script>
    <link rel="stylesheet" type="text/css"
href="codebase/skins/material.css">
</head>
<body>
webix.ready(function(){
    //some webix related UI here
});
</body>
</html>
```

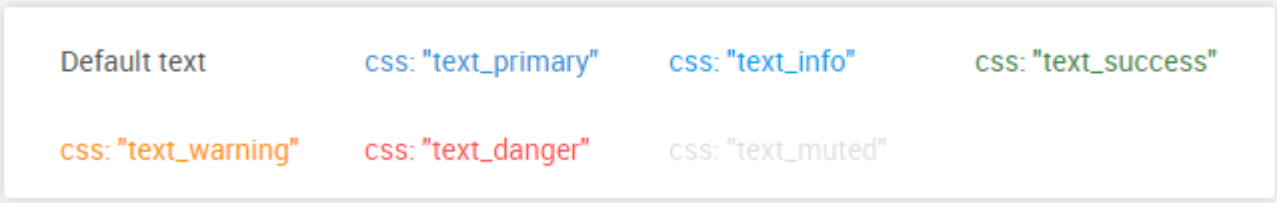
Shared CSS classes

An example of CSS classes usage is given by this link.

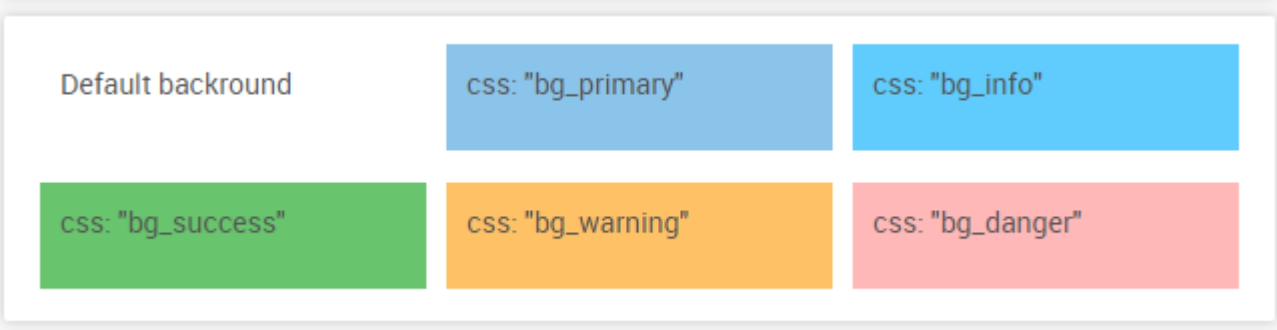
Text and Background

The following css classes are available for text:

- text_primary
- text_info
- text_success
- text_warning
- text_danger
- text_muted



The background css classes have similar names. Just replace *text_* with *bg_*:




```
webix.ui({
    rows:[{
        template:"Some text",
        css:"text_info"
    },{
        template:"Some background",
        css:"bg_danger"
    }]
});
```

Buttons

You can apply several classes for a button:

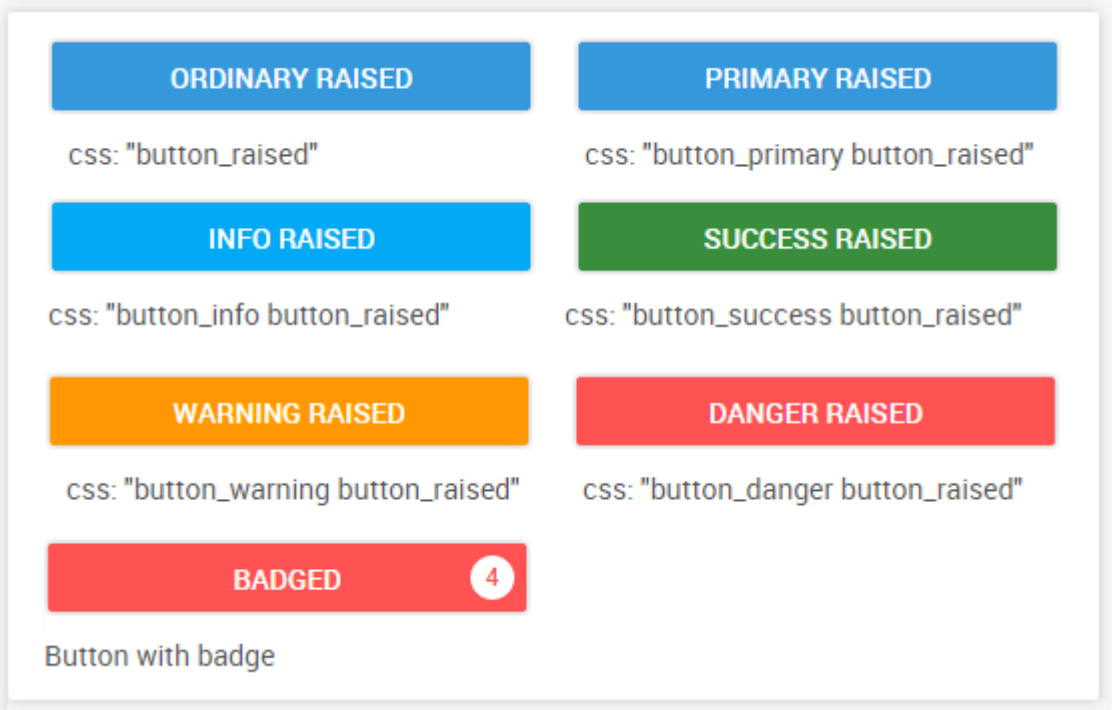
- button_primary
- button_info
- button_success
- button_warning
- button_danger
- button_transparent
- button_raised

The flat buttons look like this by default:



You can also combine flat and "raised" buttons by means of using several css rules:

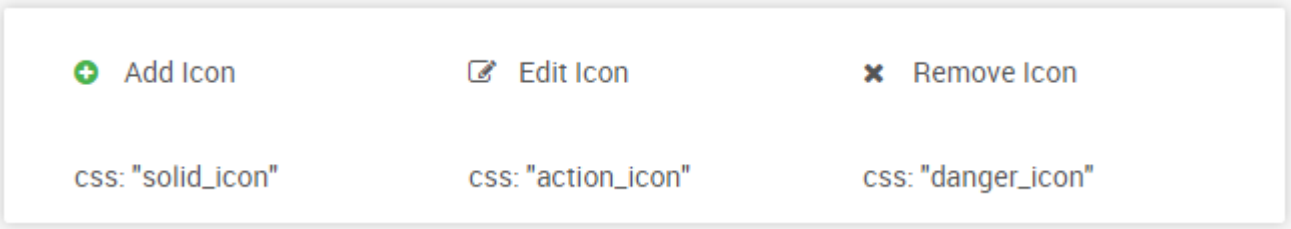
```
webix.ui({
    view:"button",
    css:"button_success button_raised"
});
```



Icons

There are 3 css icon classes:

- solid_icon
- action_icon
- danger_icon

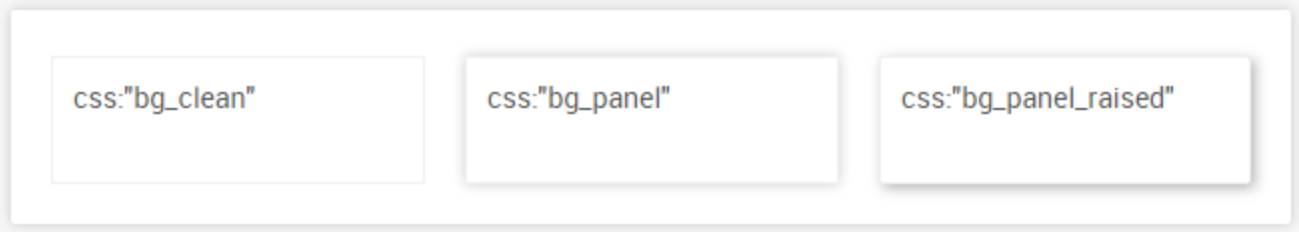



```
webix.ui({
    view:"icon",
    icon:"users",
    css:"solid_icon"
});
```

Panels

The following types of panels can be used:

- bg_clean
- bg_panel
- bg_panel_raised




```
webix.ui({
    rows:[
        { template:"A" },
        { template:"B", css:"bg_panel"},
        { template:"C" }
    ]
});
```

Material layout

The skin adds one more type of layout:

```
webix.ui({
    type:"material",
    rows:[
        { template:"A" },
        { template:"B" },
        { template:"C" }
    ]
});
```

This type of layout shows all content elements as material panels.

License

Code is released under the MIT License:

Copyright (c) 2015

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Intermediate

Component Styling

Basic Principles of Styling

All the library components are styled with the help of an external style sheet "webix.css" you specify in the document head area alongside with the "webix.js" lib file. It's a mandatory step.

After that, the styling can be customized in two modes:

1 . With **embedded styles** you define in the document head between < style > tags.

```
<style>
    .my_style  {
        font-style: italic;
    }
</style>
<script>
    webix.ui({
         view:"dataview",
         css:"my_style",
         template: "#title#<br/> Year: #year#, rank: #rank#",
    });
</script>
```

The Shawshank Redemption ***1994.*** *Rank: 1*	***The Godfather*** ***1972.*** *Rank: 2*
The Godfather: Part II ***1974.*** *Rank: 3*	***The Good, the Bad and the Ugly*** ***1966.*** *Rank: 4*
My Fair Lady ***1964.*** *Rank: 5*	***12 Angry Men*** ***1957.*** *Rank: 6*

RELATED SAMPLE: Dataview Italic

2 . By means of **inline styling** where you insert CSS into HTML tags during defining an appropriate template for data rendering.

Setting CSS class:

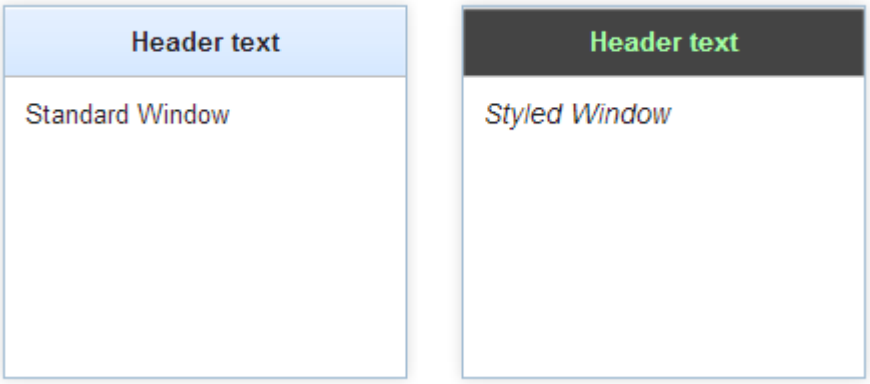
```
webix.ui({
```

```
    view:"dataview",
    template:"<div class='my_style'>#title#<br/> Year: #year#, rank:
#rank#</div>",
    data:"..."
});
```

The result will be the same as above.

Adding Style

```
view:"window",
body:{
    template:"<span style='font-size:14px;font-style:italic'>Styled
Window</span>"
}
```



RELATED SAMPLE: Styled Window

Dynamic Styles

A CSS class can be added to the component on the go, by either using the standard pattern of property redefinition:

```
.mycss{
   color:red;
}

//"datatable" - component ID
$$("datatable").define("css", "mycss");
```

or by using webix.html helper:

```
//getNode() returns the component's topmost node
webix.html.addCss( $$("datatable").getNode(), "mycss");
```

The advantage of the addCss method is that it is supplied with the opposite one, the removeCss that can be used to undefine the CSS class from the component:

```
webix.html.removeCss( $$("datatable").getNode(), "mycss");
```

Styling inside Type Parameter

When you set dimensions of component items, you use the **type** property that contains an object with an item's width and height. There you can as well put other parameters including template and css.

```
webix.ui ({
    view:"dataview",
    type:{
        height: 45,
        width: 300,
        template:"<span style='color:#67B802; font-
style:italic'>#name#</span>"
    }
    ...
})
```

Type can also be created separately through the **webix.type** class.

```
webix.type(webix.ui.dataview,{
    name:"custom",
    template:"<span style='color:#67B802; font-
style:italic'>#name#</span>",
    height: 45,
    width: 300
});

webix.ui ({
    view:"dataview",
    type: "custom",
    ...
})
```

The Shawshank Redemption *1994*	*The Godfather* *1972*
The Godfather: Part II *1974*	*The Good, the Bad and the Ugly* *1966*
My Fair Lady *1964*	*12 Angry Men* *1957*

RELATED SAMPLE: Dataview Type Styling

Adding Predefined Webix CSS Classes

You can either **apply predefined CSS** as it is:

```
webix.ui({
    view:"dataview",
    template:"<div class='webix_strong'>#title#</div> Year: #year#, rank:
#rank#",
    ...
});
```

RELATED SAMPLE: Controlling a Number of Displayed Items

Or **redefine default CSS** for this document:

```
.mywin .webix_win_head .webix_template{
    background: #444; color:#AFA;
}
```



RELATED SAMPLE: Styled Window

As far as you've noticed, **Webix default CSS classes** are marked by a **"webix_"** prefix. Moreover, each component part has its peculiar CSS class, you can redefine. Read below to learn more about it.

Styling separate items in the component

Instead of defining css class for the whole component you can define css class for single items. To do so you need to place $css property in the data object. This property can contain a name of css class or a hash of css rules.

```
Setting item's style directly in the dataset

<style>
    .my_style{
        background-color:#FFAAAA;
    }
</style>

<script>
webix.ui({
    view:"list",
    data:[
```

```
        { id:1,
          value:"The Shawshank Redemption",
          $css:"highlight"
        },
        { id:2,
          value:"The Godfather",
          $css:{ "text-align":"right" }
        }
    ]
})
</script>
```

If you want to change styles dynamically, after page loading you can use the addCss and removeCss api.

Styling of a Specific Part of a Component (CSS maps)

CSS maps help stylize a specific part of a component. You should refer to its CSS map, pick the css class to the needed part and define the style in the document head.

Style applies to the header of the unitlist:

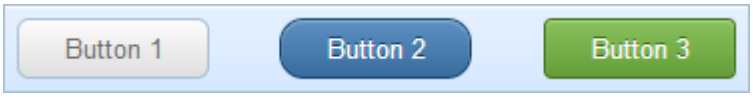



```
<style>
.webix_unit_header{
    color:#cb4c26;
    font-style:italic;
}
</style>
```

CSS classes for each component are described in CSS Image Maps Section

Styling Instances of One and the Same Object

Controls are most frequently used components so one and the same control type, e.g. button, occurs more than once on the page, as a rule.



Take, for instance, buttons. If you want to style each button differently, you should define a separate CSS class for each button, like this:

```
.bt_1 input{
    border-radius:12px; -moz-border-radius:12px; -webkit-border-
radius:12px;
    background: #396D9E;
    ...
}
.bt_2 input{
    border-radius: 4px; -webkit-border-radius: 4px;-moz-border-radius:
4px;
    border: 1px solid #3D7113;
    ...
}
```

Here **.bt1** and **bt2** are class names for two different buttons while **input** specifies that the button's input area should be styled as well.

RELATED SAMPLE: Buttons: Styling

Changing Default Icons

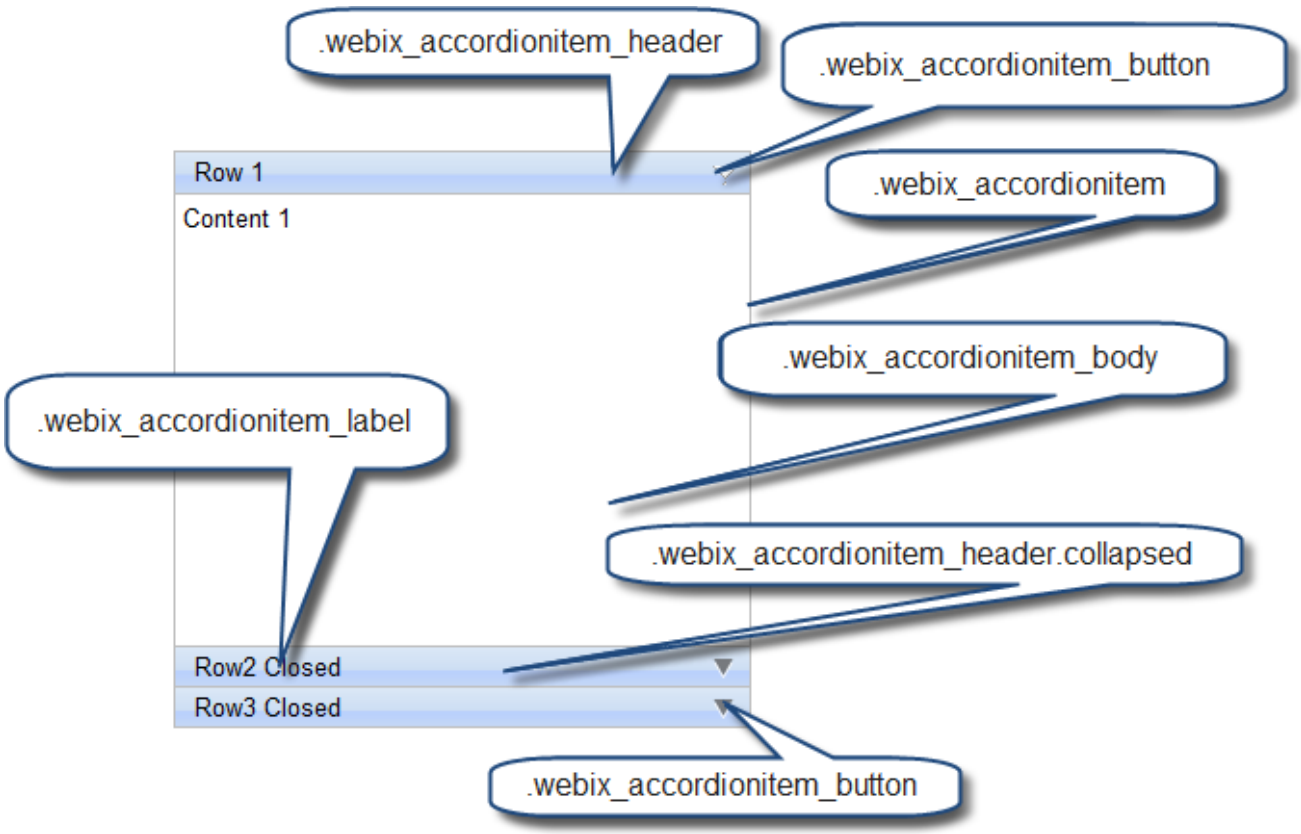
Default FontAwesome icons that are used for various controls can be changed to custom ones, which is described in detail in a separate documentation article.

CSS image maps

Components:	Controls:
◦ Accordion ◦ Calendar ◦ Carousel ◦ Chart ◦ Colorboard ◦ Dataview ◦ Datatable ◦ Tree ◦ Groulist ◦ Layout ◦ List ◦ Menu ◦ Message Box ◦ Popup ◦ Property Sheet ◦ Toolbar ◦ Treetable ◦ Unitlist ◦ Window ◦ XList ◦ Uploader	◦ Button ◦ Checkbox ◦ Combo ◦ Counter ◦ Colorpicker ◦ Datepicker ◦ Label ◦ Radio ◦ Select ◦ Segmented ◦ Tabbar ◦ Richselect ◦ Text ◦ Textarea ◦ Toggle

Accordion

CSS Image Map

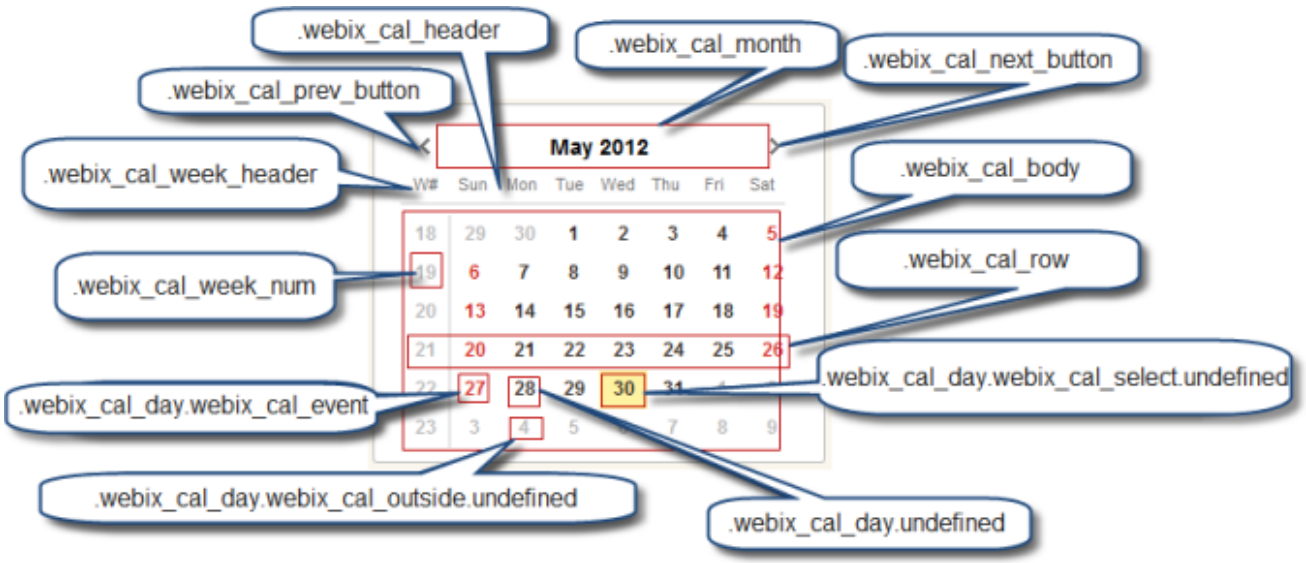


css class	Element
.webix_accordionitem	a single pane of the accordion
.webix_accordionitem_body	the body of the pane
.webix_accordionitem_header	the expanded header
.webix_accordionitem_header.collapsed	the collapsed header
.webix_accordionitem_label	the label of the header
.webix_accordionitem_button	'expanded' or 'collapsed' arrow depending on the pane type

Read more about Accordion

Calendar

CSS Image Map

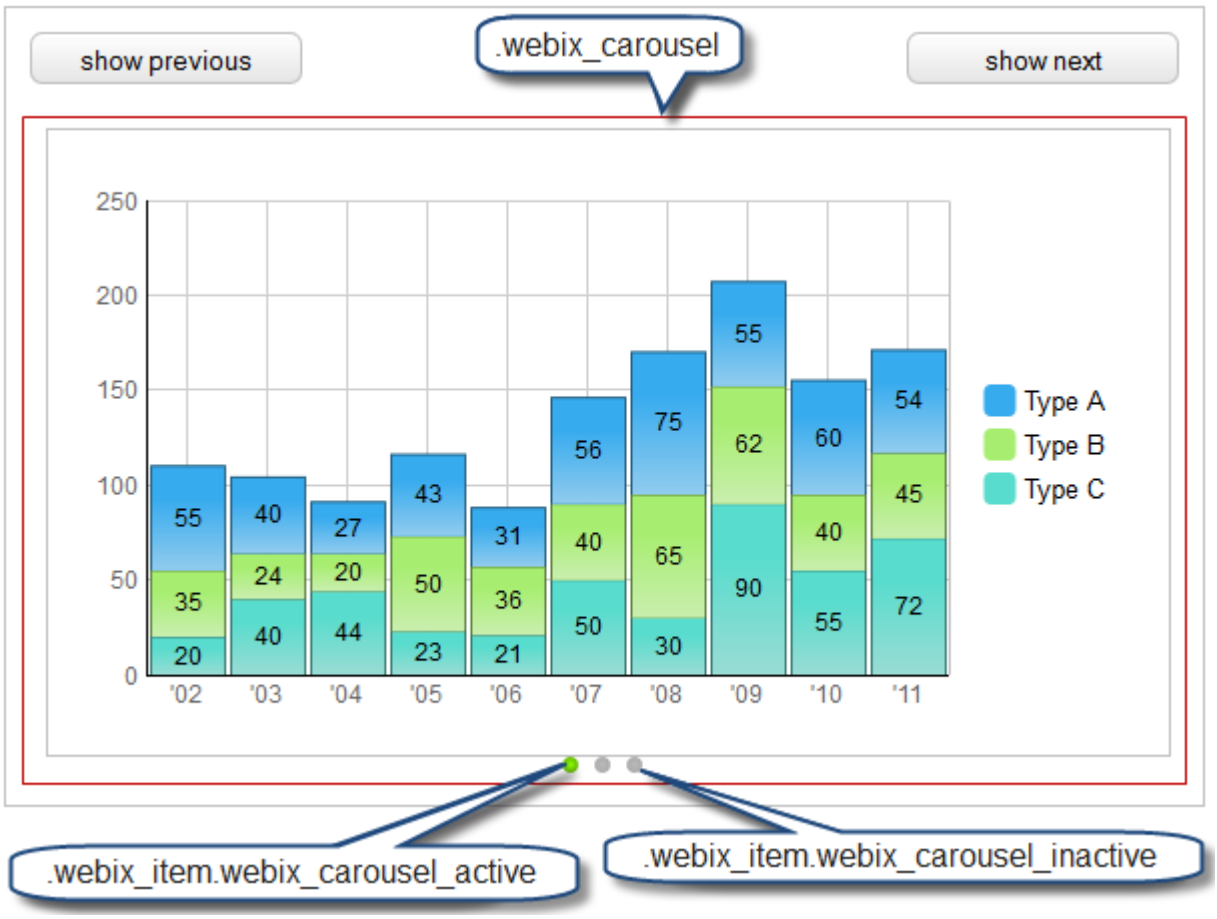


css class	Element
.webix_cal_month	the 'month' section of the calendar
.webix_cal_next_button	the 'next' button
.webix_cal_prev_button	the 'previous' button
.webix_cal_header	the header of the calendar
.webix_cal_week_header	the week header of the calendar
.webix_cal_body	the body of the calendar
.webix_cal_row	the row of the calendar (week)
.webix_cal_week_num	the week number of the calendar
.webix_cal_day.webix_cal_outside.webix_cal_event	the day of the neighbouring month in the calendar (weekend)
.webix_cal_day.webix_cal_outside.undefined	the day of the neighbouring month in the calendar (weekday)
.webix_cal_day.undefined	any weekday of the month
.webix_cal_day.webix_cal_event	weekend on the calendar
.webix_cal_day.webix_cal_select.undefined	the selected day

Read more about Calendar

Carousel

CSS Image Map



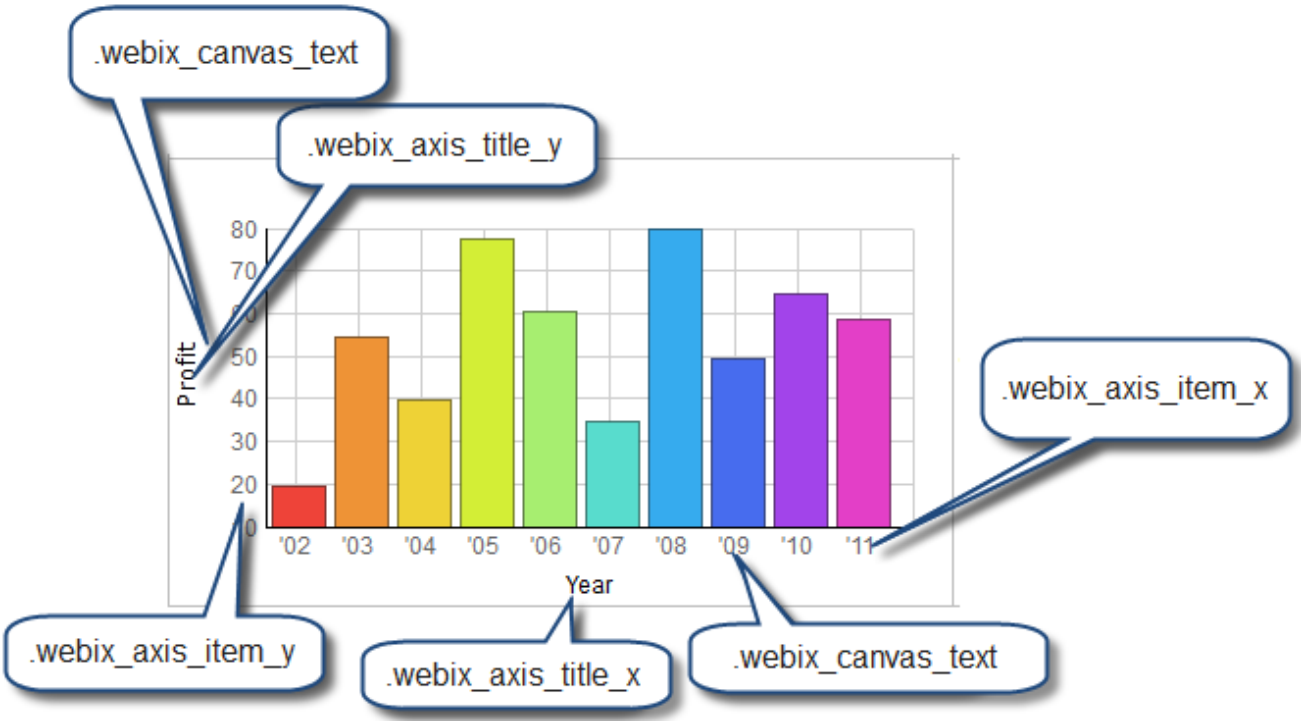
Carousel is a layout-building component that can house any of the data-presenting components while navigation between them is enabled with the help of standard controls. Check their CSS maps if you want to style carousel elements.

css class	Element
.webix_carousel	carousel object
.webix_item.webix_carousel_active	casousel item that is currently shown
.webix_item.webix_carousel_inactive	carousel item that is currently hidden

Read more about Carousel

Chart

CSS Image Map

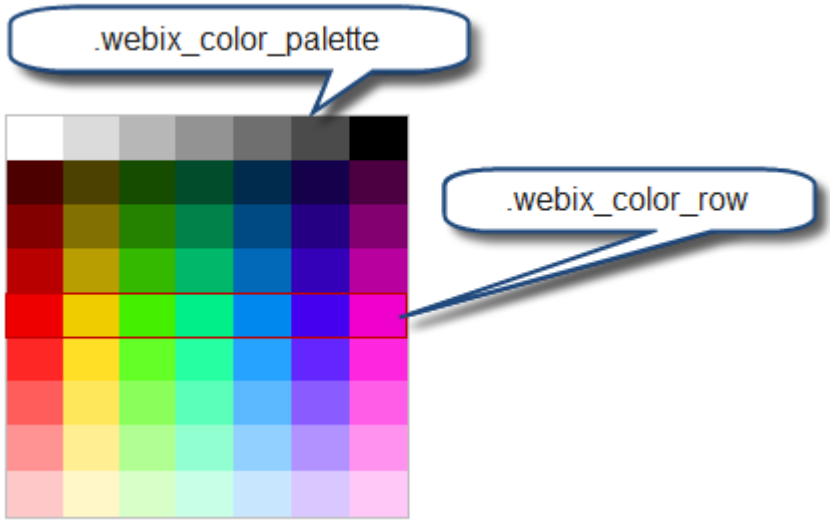


css class	Element
.webix_axis_item_y	the Y-axis item label
.webix_axis_title_y	the Y-axis title
.webix_axis_item_x	the X-axis item label
.webix_axis_title_x	the X-axis title
.webix_canvas_text	all text labels using in the chart (including Y-axis, X-axis items and title

Read more about Chart

Colorboard

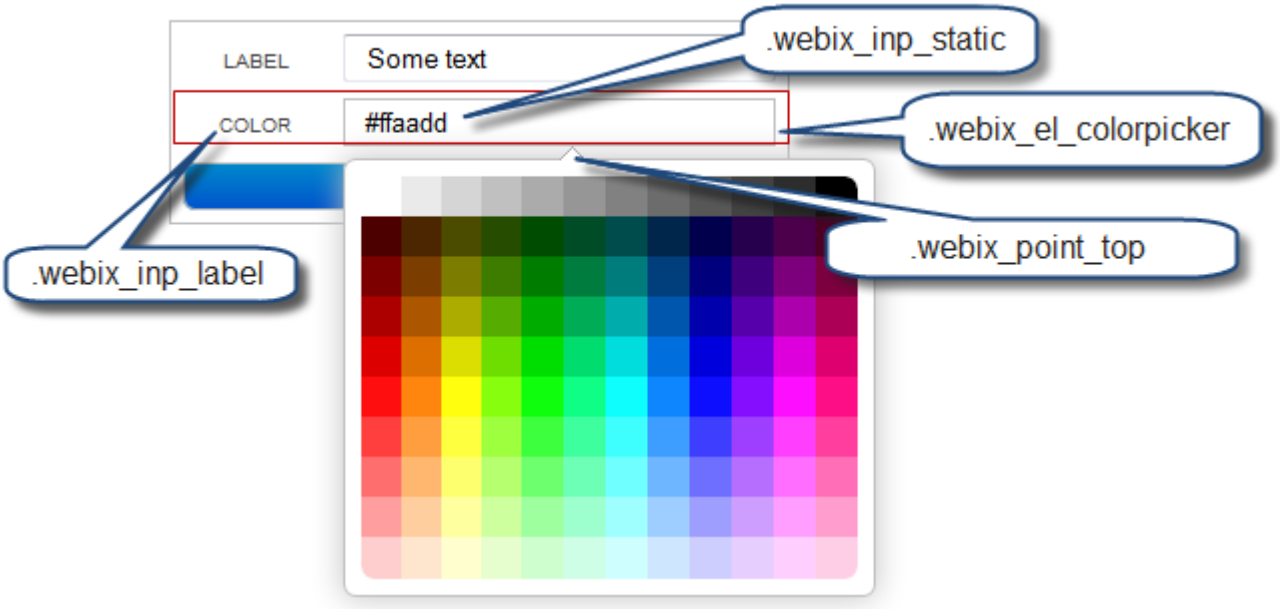
CSS Image Map



css class	Element
.webix_color_palette	the whole colorboard
.webix_color_row	a row in the colorboard

Colorpicker

CSS Image Map



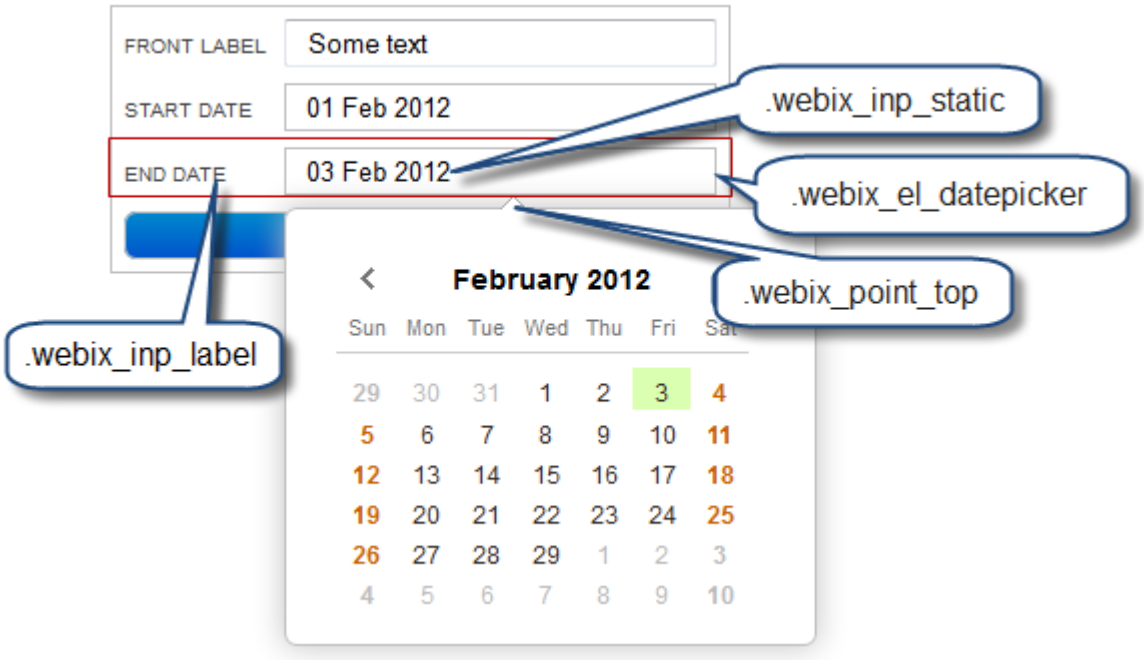
css class	Element
.webix_el_colorpciker	the whole 'colorpicker' control
.webix_inp_label	text label for the control
.webix_inp_static	input area of the control
.webix_point_top	a popup arrow that points to a control that initied its appearance

To study the CSS map of the popup colorboard, please, refer to its CSS Map.

Read more about Colorpicker

Datepicker

CSS Image Map



css class	Element
.webix_el_datepicker	the whole 'colorpicker' control
.webix_inp_label	text label for the control
.webix_inp_static	input area of the control
.webix_point_top	a popup arrow that points to a control that initied its appearance

To study the CSS map of the popup calendar, please, refer to its CSS Map.

Read more about Datepicker

Datatable

CSS Image Map

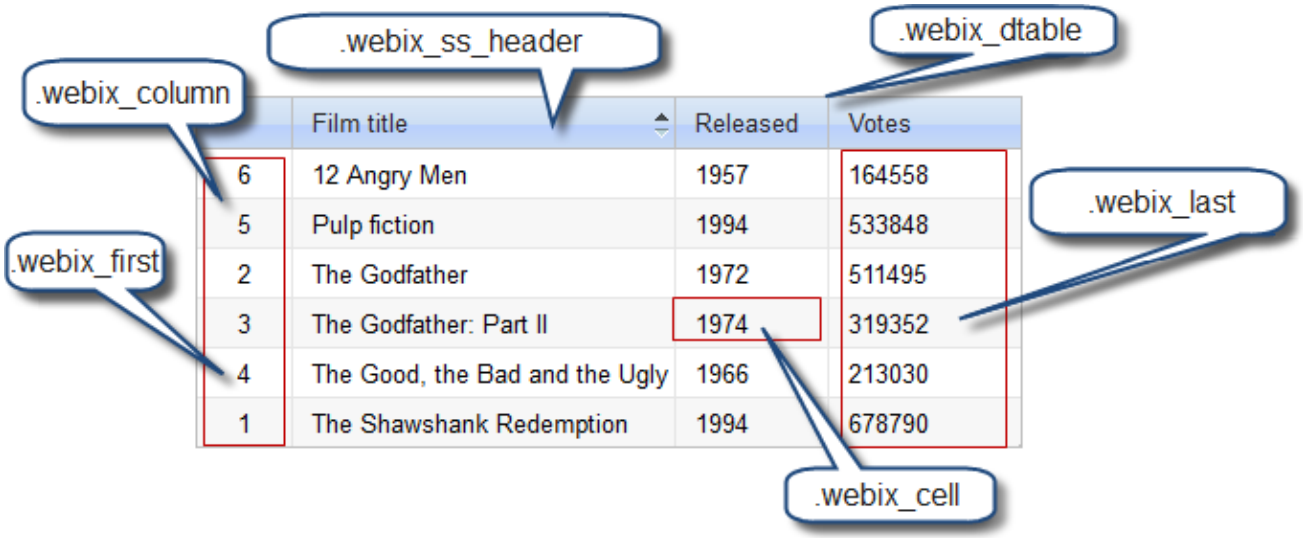


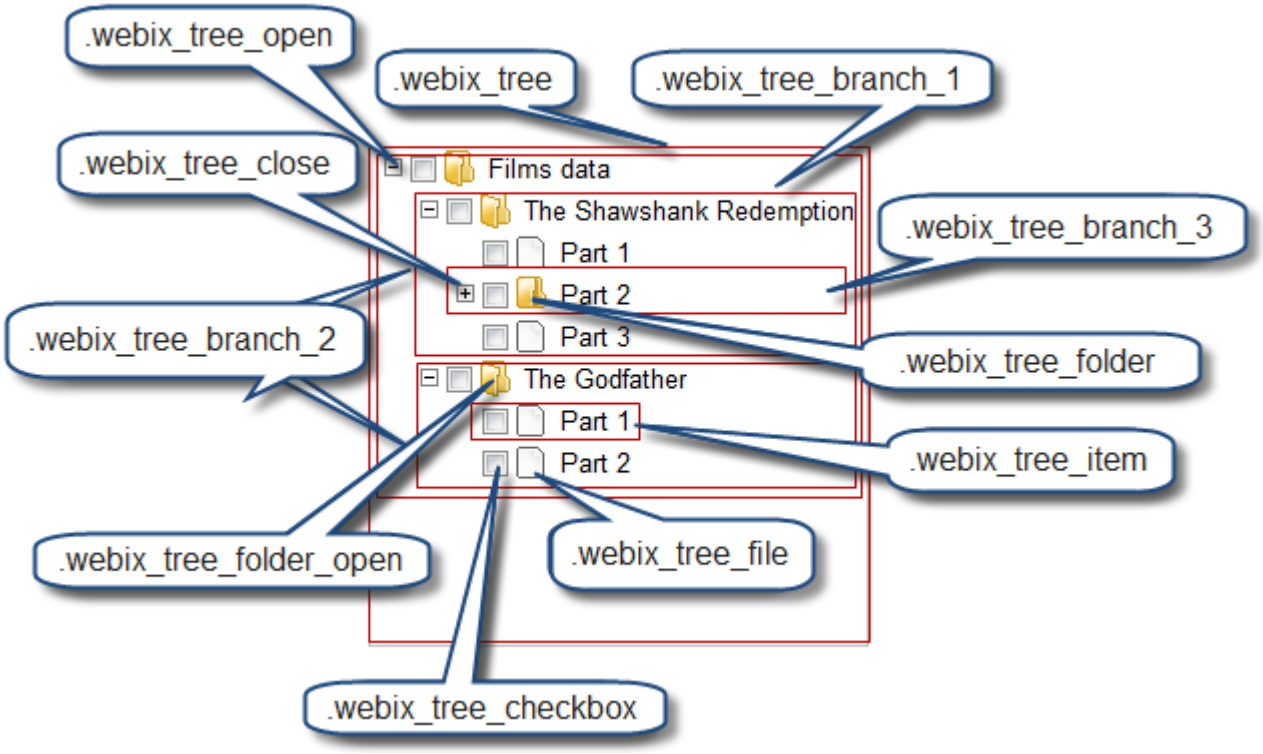
Table 1 CSS image maps

CSS class	Description
.webix_dtable	container with DataTable
.webix_ss_header	container with the header
.webix_ss_header .webix_hcell	a cell of the header
.webix_ss_header .webix_first	the first cell of the header
.webix_ss_header .webix_last	the last cell of the header
.webix_ss_footer	container with the footer
.webix_ss_footer .webix_hcell	a cell of the footer
.webix_ss_footer .webix_first	the first cell of the footer
.webix_ss_footer .webix_last	the last cell of the footer
.webix_ss_body	container with the table body
.webix_column	a column of the table
.webix_column .webix_last	the last column of the table
.webix_column .webix_first	the first column of the table
.webix_column .webix_cell	a cell of the column

Read more about Datatable component.

Tree

CSS Image Map

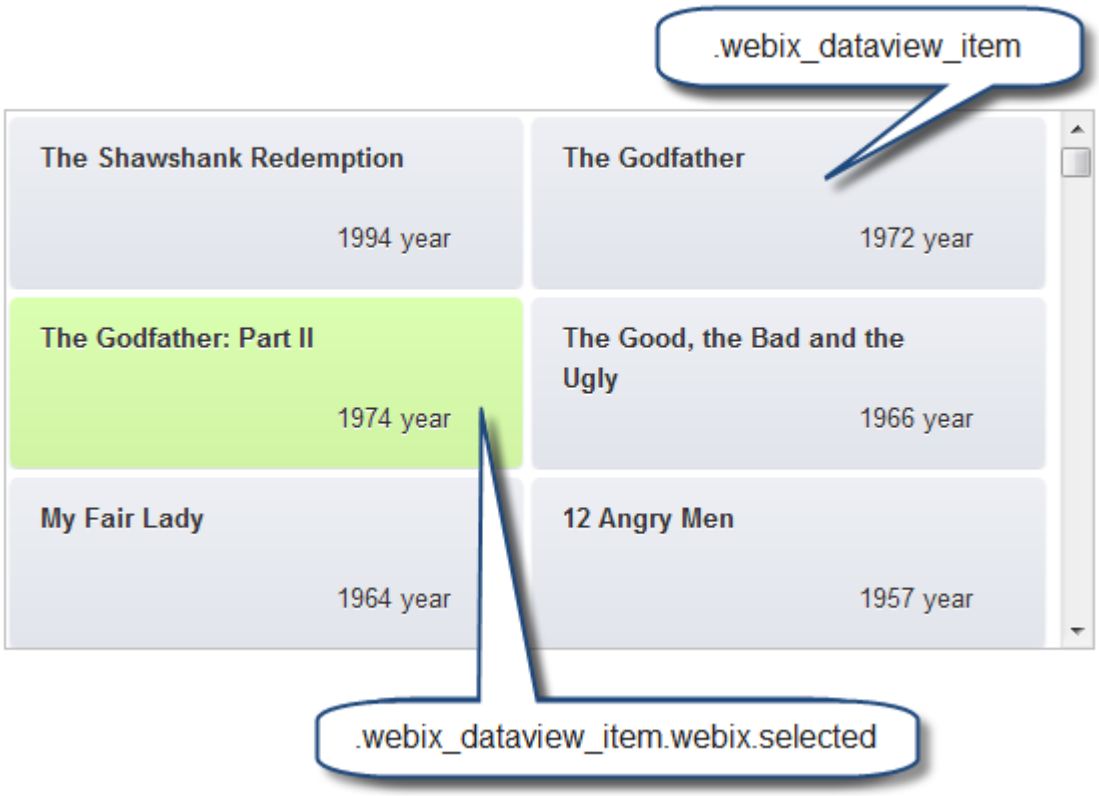


css class	Element
.webix_tree	the whole tree
.webix_tree_branch_1	the first-level items of the tree
.webix_tree_branch_2	the second-level items of the tree
.webix_tree_branch_3	the third-level branches of the tree
.webix_tree_item	an inline item within the tree
.webix_tree_folder	a 'closed' folder icon
.webix_tree_folder_open	an 'open' folder icon
.webix_tree_file	a file icon
.webix_tree_close	a 'closed branch' sign
.webix_tree_open	an 'open branch' sign
.webix_tree_checkbox	a checkbox (if any)

Read more about Tree.

Dataview

CSS Image Map

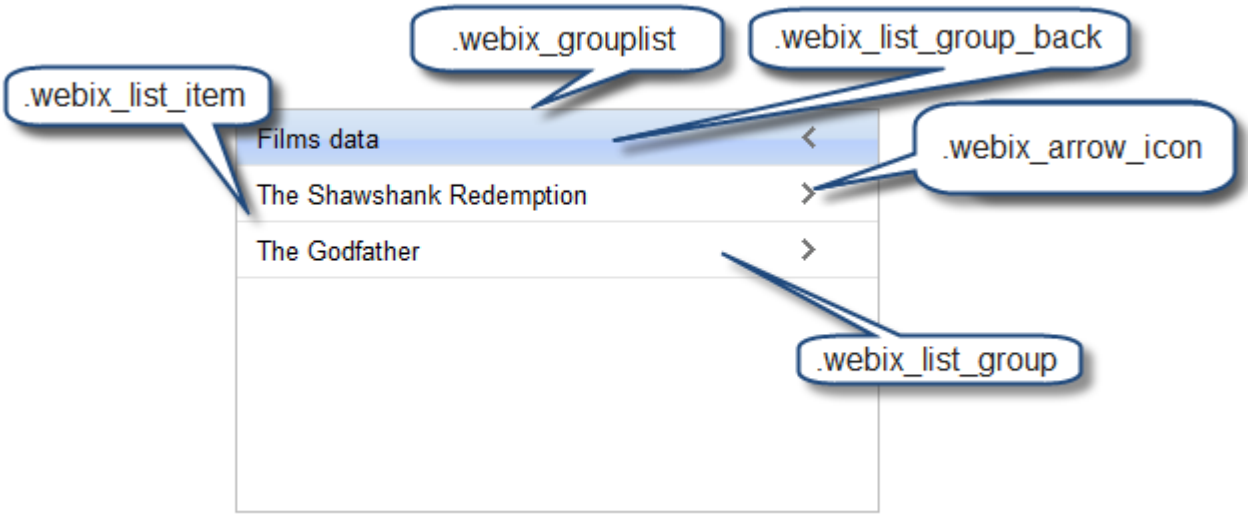


css class	Element
.webix_dataview_item	an item of the dataview
.webix_dataview_item.webix.selected	the selected item of the dataview

Read more about Dataview

Groulist

CSS Image Map

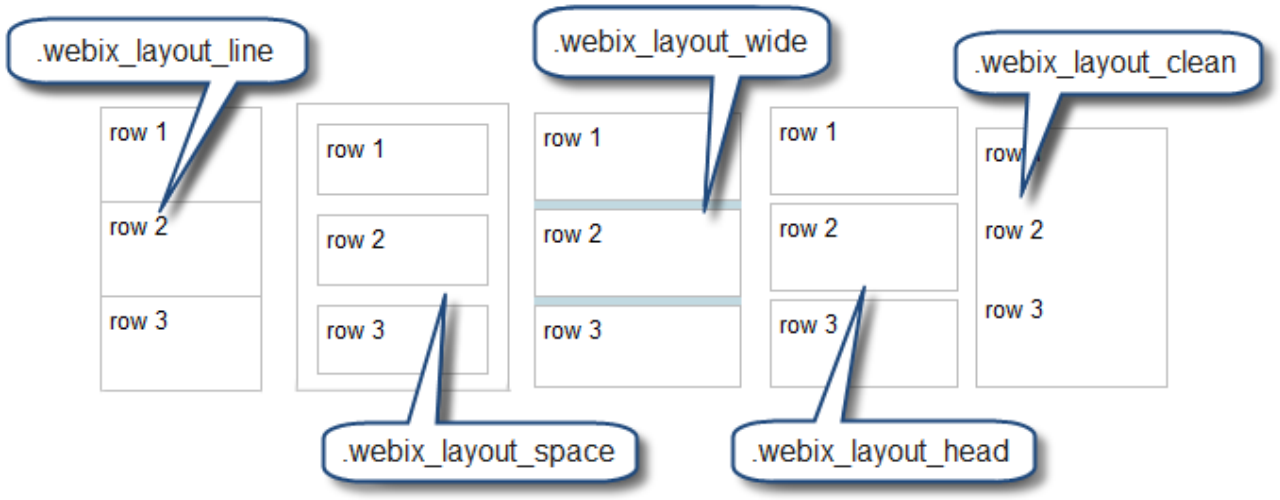


css class	Element
.webix_grouplist	the whole grouplist
.webix_list_item	an item in the grouplist
.webix_list_group_back	the group header item in the selected state
.webix_list_group	the group header item in the unselected state
.webix_arrow_icon	the arrow on the group header item

Read more about Grouplist

Layout

CSS Image Map

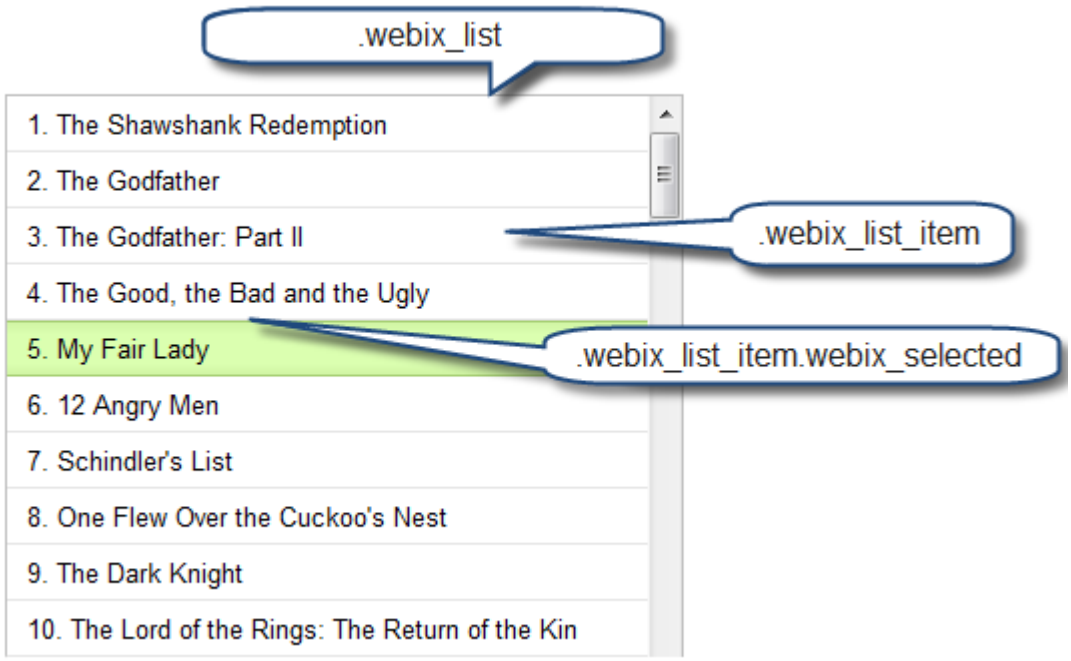


css class	Element
.webix_layout_space	the borders of the type 'space'
.webix_layout_head	the borders of the type 'head'
.webix_layout_wide	the borders of the type 'wide'
.webix_layout_line	the borders of the type 'line'
.webix_layout_clean	the borders of the type 'clean'

Read more about Layout

List

CSS Image Map

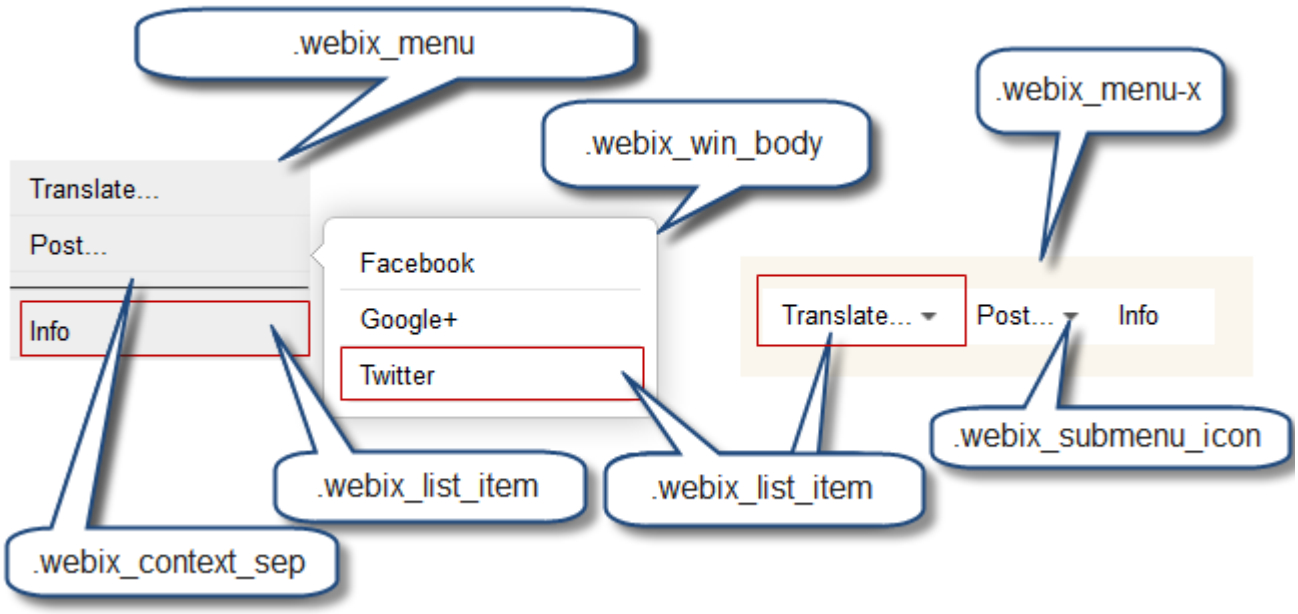


css class	Element
.webix_list	list view
.webix_list_item	an item of the list
.webix_list_item.webix_selected	the selected item of the list

Read more about List

Menu

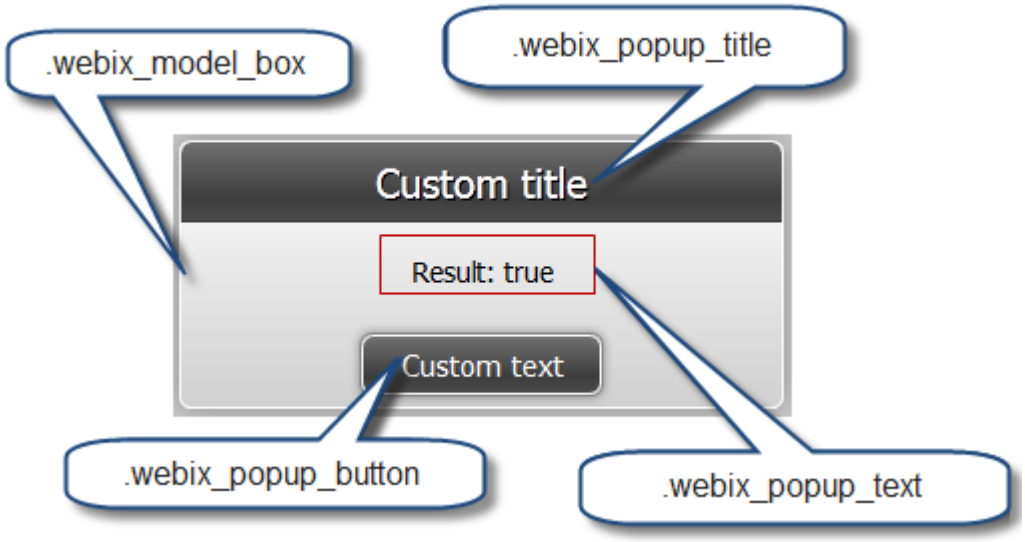
Css Image Map



css class	Element
.webix_menu	the whole menu
.webix_menu-x	the whole menu in case of horizontal arrangement
.webix_win_body	a drop-down list
.webix_list_item	an item of the drop-down list
.webix_context_sep	a line separator between menu groups
.webix_submenu_icon	a icon for an item with a submenu

Message Box

CSS Image Map

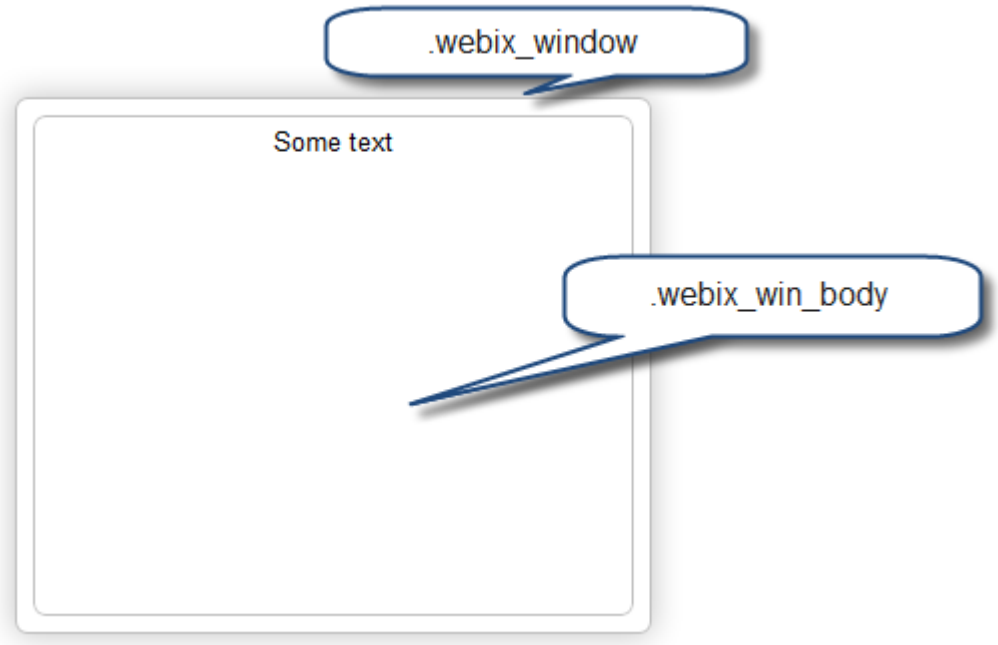


css class	Element
.webix_model_box	the whole message box
.webix_popup_text	text in the box body
.webix_popup_title	the header the box
.webix_popup_button	a button in the control area of the box

Read more about Message Boxes

Popup

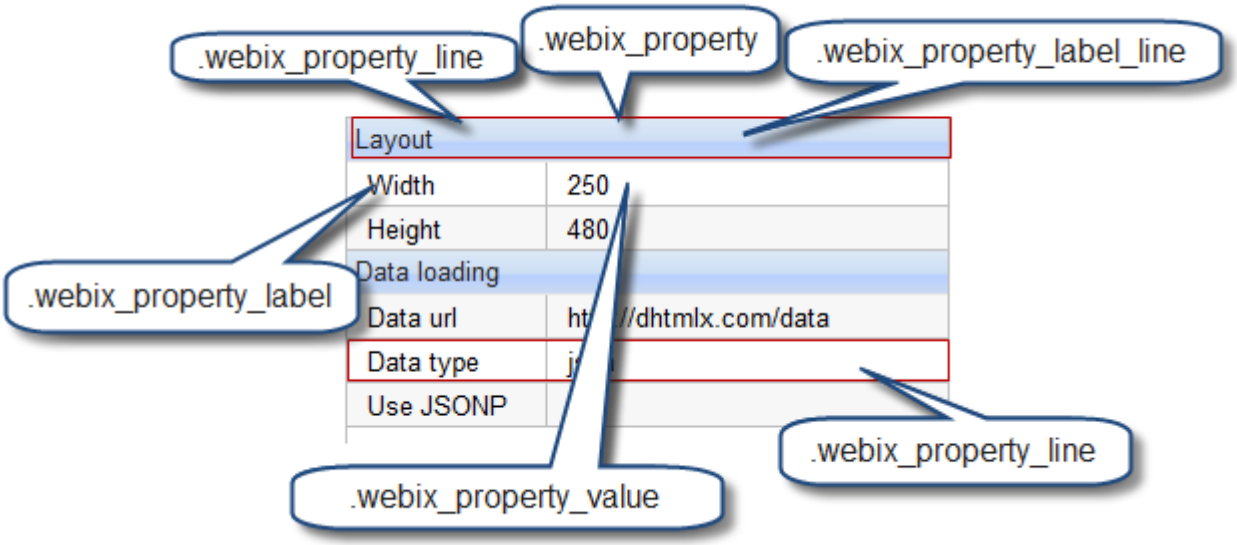
CSS Image Map



css class	Element
.webix_window	the whole popup window
.webix_win_body	the body of the window

Read more about Popup windows

Property Sheet



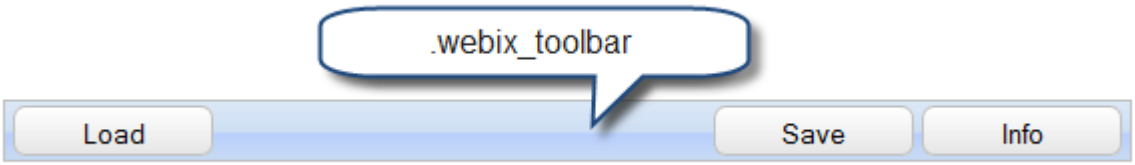
CSS Image Map

css class	Element
.webix_property	the whole xlist
.webix_property_line	an item of the list
.webix_property_label_line	an item of the list
.webix_property_label	an item of the list
.webix_property_value	an item of the list

Read more about Property Sheet

Toolbar

CSS Image Map



css class	Element
.webix_toolbar	the whole toolbar

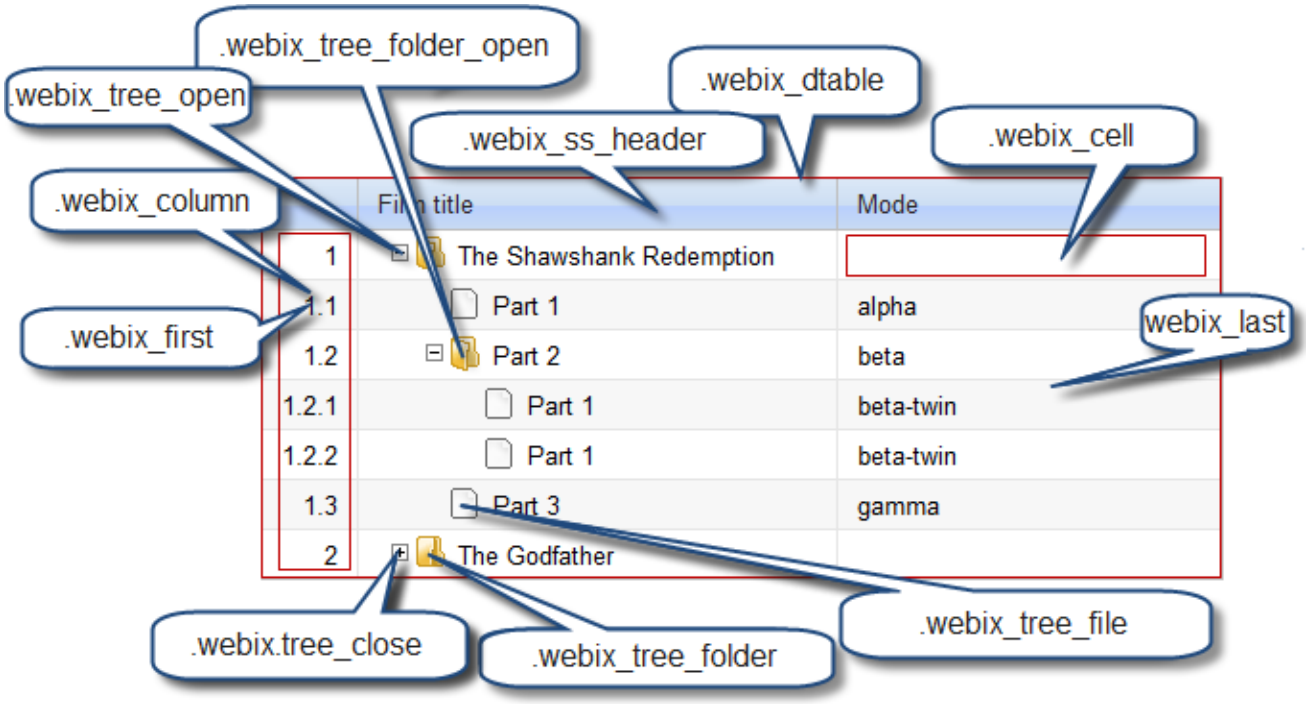
Toolbar is a container for multiple controls. See the corresponding image maps to learn about custom control styling.

Read more about Toolbar

Treetable

Treetable combines features of the Tree and Datatable. Hence, its CSS classes for it are inherited from both of the components.

CSS Image Map

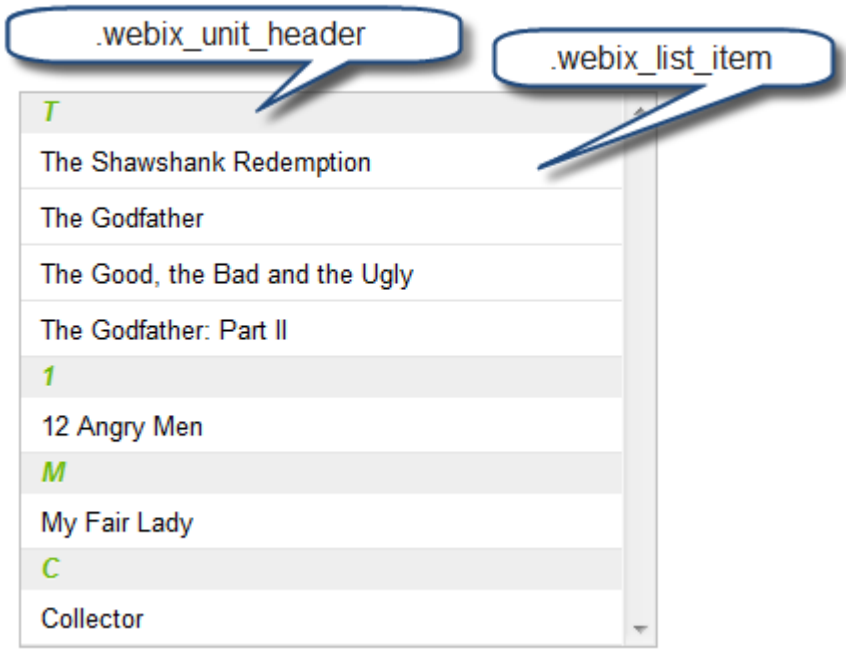


css class	Element
.webix_dtable	the whole datatable
.webix_ss_header	the header of a datatable
.webix_column	a column of a datatable
.webix_cell	a cell of a datatable
.webix_first	the first column of a datatatble
.webix_last	the last column of a datatable
.webix_tree_folder_open	an 'open' folder icon
.webix_tree_folder	a 'close' folder open
.webix_tree_file	a file icon
.webix_tree_open	an 'opened branch' sign
.webix_tree_close	a 'closed branch' sign
.webix_tree_checkbox	a 'checkbox' sign (if any)

Read more about Treetable

Unitlist

CSS Image Map

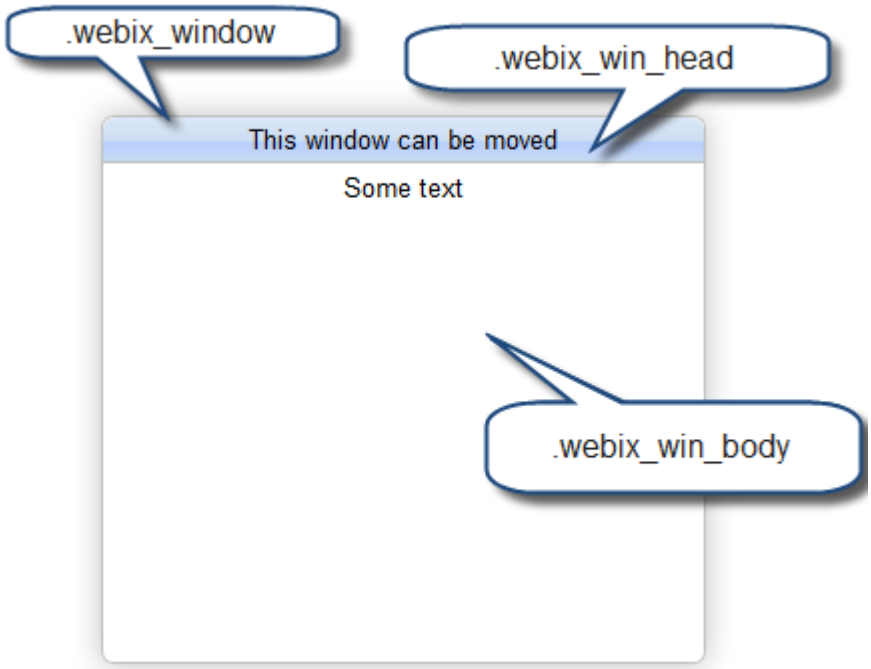


css class	Element
.webix_list_item	an item of the list
.webix_unit_header	the header of a unit

Read more about Unitlist

Window

CSS Image Map



css class	Element
.webix_win_head	the head of the window
.webix_win_body	the body of the window
.webix_window	the whole window

Read more about Window

XList

CSS Image Map

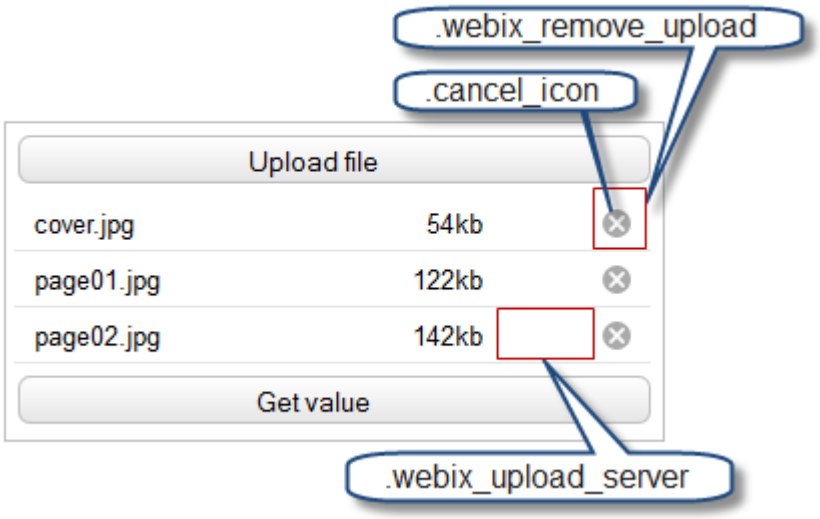


css class	Element
.webix_list-x	the whole xlist
.webix_list_item	an item of the list

Read more about XList

Uploader

CSS Image Map



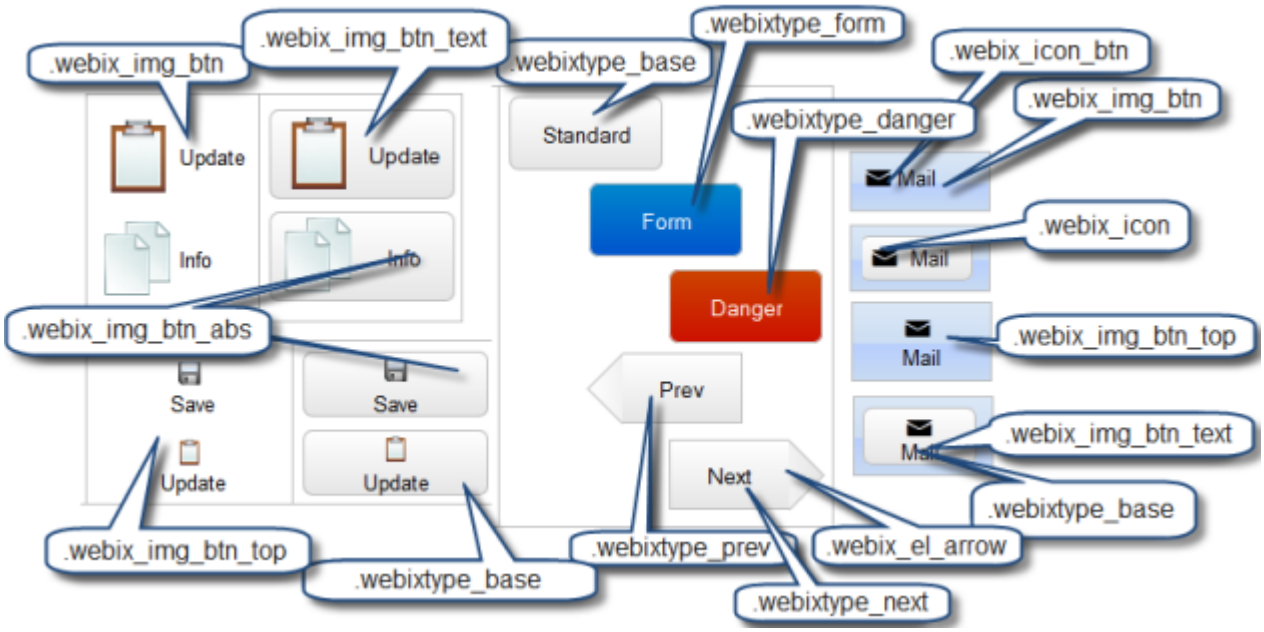
css class	Element
.webix_remove_upload	the "remove item" area
.cancel_icon	the "remove_item" icon
.webix_upload_server	the "upload_state" area
.icon-ok webix_icon	the "upload ready" icon

Basically, uploader is comprised of other Webix components such as button and list/template. See their CSS maps for details.

Read more about Uploader

Button

CSS Image Map



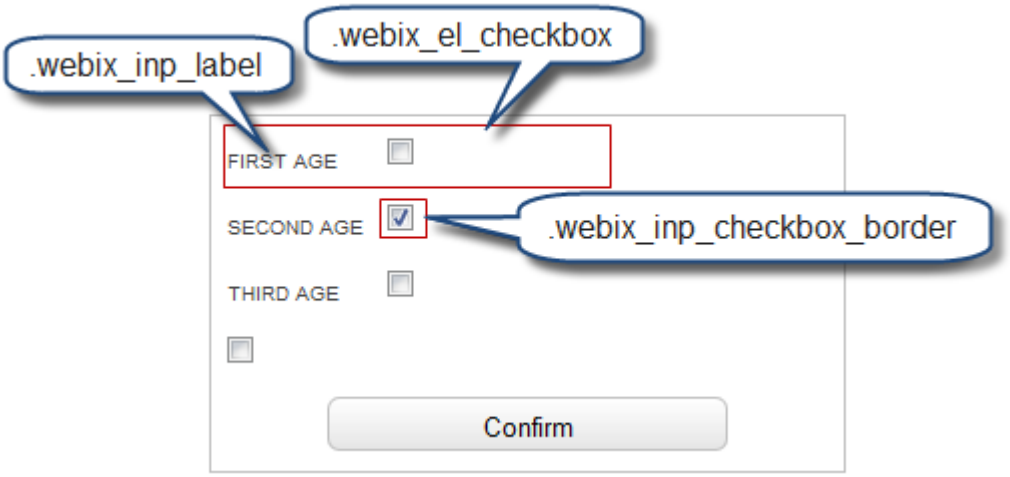
css class	Element
.webix_el_button	any button
.webixtype_base	default button
.webixtype_danger	a button with the "danger" type
.webixtype_form	a button with the "form" type
.webixtype_prev	a button with the "prev" type
.webixtype_next	a button with the "next" type
.webix_img_btn	a button with the "image" and "icon" type
.webix_img_btn_abs	a picture of any button with an image
.webix_img_btn_top	a button with the "imageTop" and "iconTop" type
.webix_img_btn_text	text value of the a button with an image
.webix_icon [icon-mail]	an icon for the "iconButton", "iconTop" and "iconButtonTop" button
.webix_icon_btn [icon-mail]	an icon for the "icon" button [icon-name]
.webix_el_arrow	an arrow for the 'prev', 'next' buttons

You can find the name of any icon on the Font Awesome site since icons are taken from there. Iconization in the library is described separately.

Read more about Buttons.

Checkbox

CSS Image Map

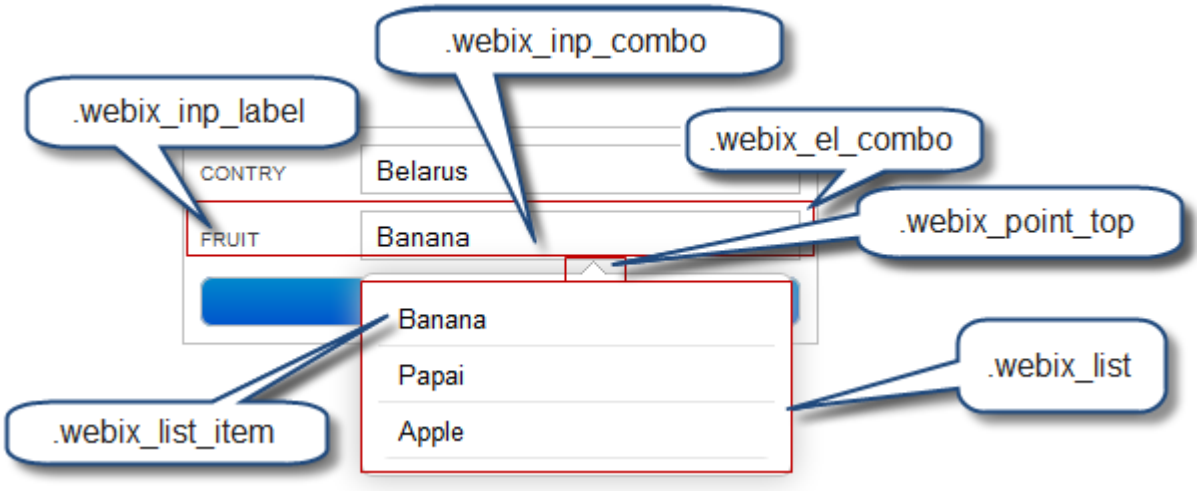


css class	Element
.webix_el_checkbox	the whole checkbox
.webix_inp_label	the label
.webix_inp_checkbox_border	the input area

Read more about Checkbox

Combo

CSS Image Map

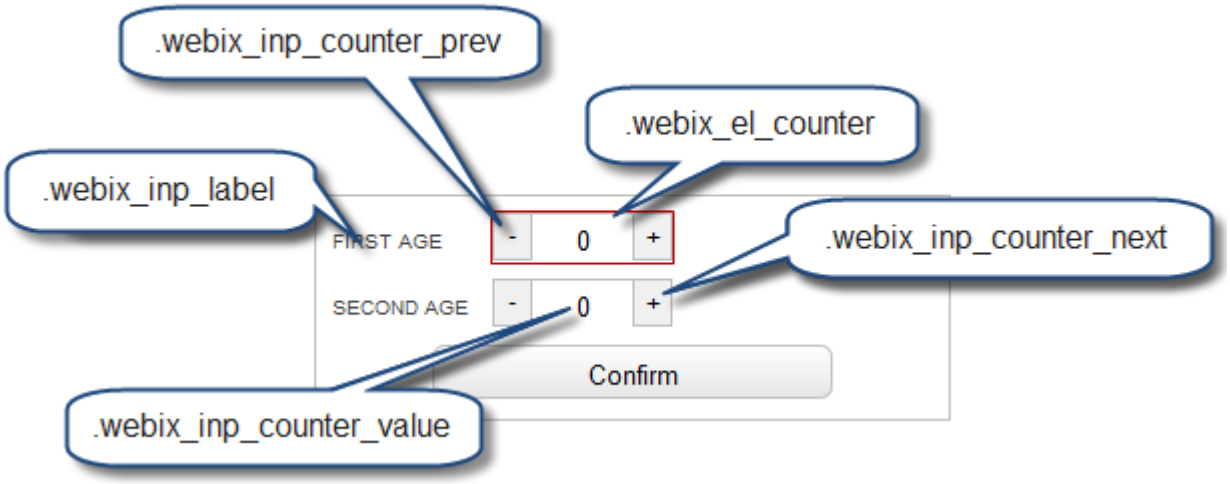


css class	Element
.webix_el_combo	the whole combo
.webix_inp_label	the label
.webix_inp_combo	the input area
.webix_list	the drop-down list
.webix_point_top	the top arrow of pull-down
.webix_list_item	an item of the list

Read more about Combo

Counter

CSS Image Map

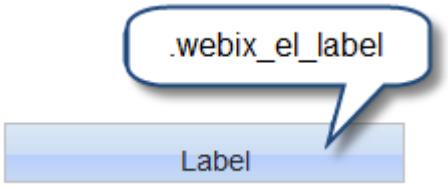


css class	Element
.webix_el_counter	the whole counter
.webix_inp_label	the label of the counter
.webix_inp_counter_prev	the left button of the input area
.webix_inp_counter_value	the input area displaying the chosen value
.webix_inp_counter_next	the right button of the input area

Read more about Counter

Label

CSS Image Map

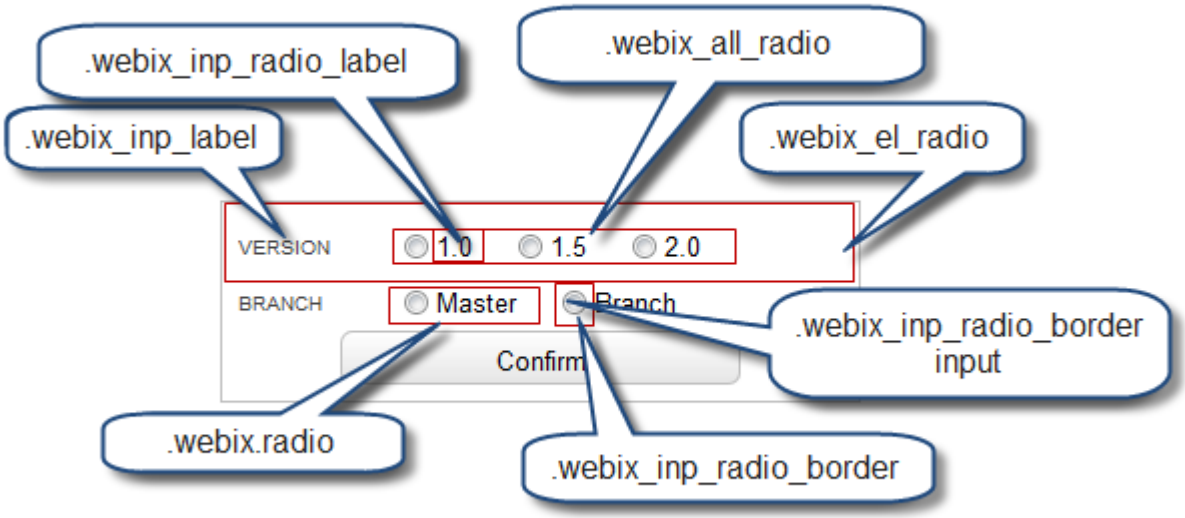


css class	Element
.webix_el_label	the whole label

Read more about Label

Radio

CSS Image Map

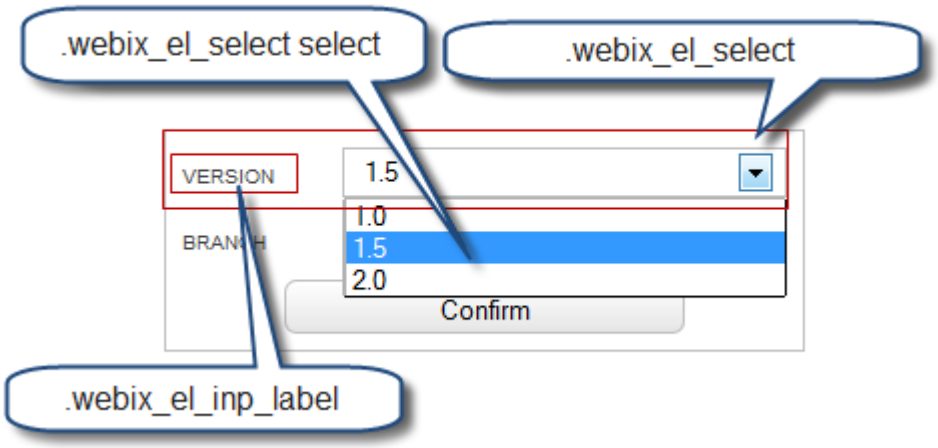


css class	Element
.webix_el_radio	the whole 'radio' control
.webix_inp_label	the label for the whole control
.webix_inp_radio_label	the label for a single radio button
.webix_all_radio	all radio buttons together
.webix_inp_radio_border	an area for a single radio button
.webix_inp_radio_border input	the input area

Read more about Radio

Select

CSS Image Map

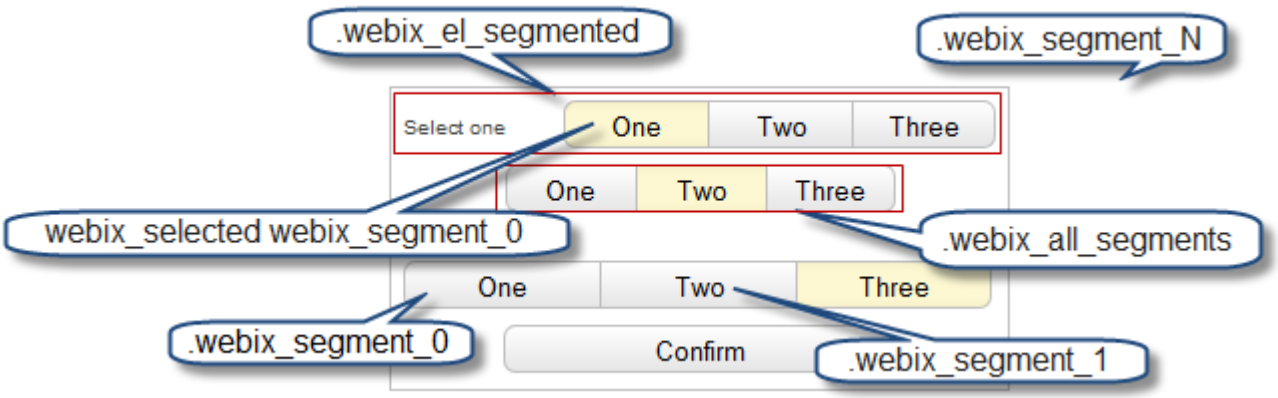


css class	Element
.webix_el_select	the whole 'select' control
.webix_el_select select	the dropdown list of the control
.webix_el_inp_label	text label for the control

Read more about Select

Segmented

CSS Image Map

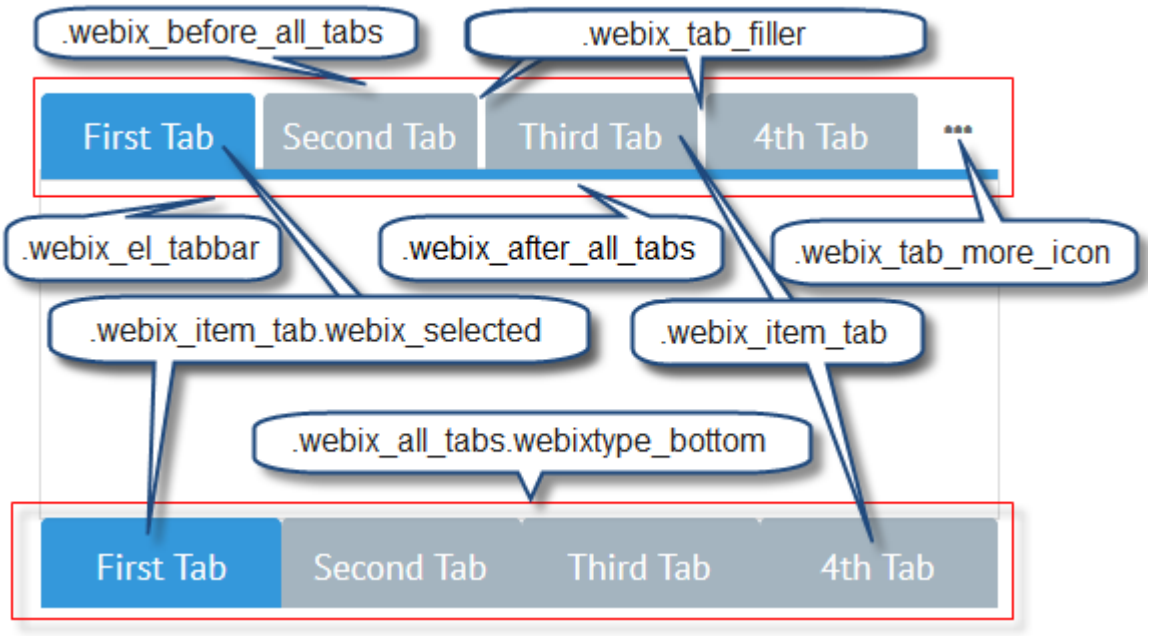


css class	Element
.webix_el_segmented	the whole 'segmented' control
.webix_segment_0	the first tab in the unselected state
.webix_segment_N	the last tab in the unselected state
.webix_segment_1	the tabs from the second to the penultimate one in the unselected state
.webix_selected webix_segment_0	the first tab in the selected state
.webix_all_segments	all segments of the control

Read more about Segmented

Tabbar

CSS Image Map



css class	Element
.webix_el_tabbar	The whole tabbar. Since most of the time a number of ".webix_item_tabs" will be rendered on top of it, the most often visible part of ".webix_el_tabbar" is the ".webix_tab_more_icon" at the end of the tab strip.
.webix_tab_more_icon	Icon at the end of the tab strip when there are more tabs to display than fit horizontally.
.webix_item_tab	A tab in the tabbar.
.webix_item_tab.webix_selected	The selected tab.
.webix_all_tabs	All tabs in the tabbar. Displayed on top of the ".webix_el_tabbar" and under ".webix_item_tab" and ".webix_tab_filler".
.webix_all_tabs.webixtype_bottom	All tabs in the tabbar of the "bottom" type
.webix_tab_filler	Space between ".webix_item_tab"s and on the right and left ends of a top tabbar.
.webix_before_all_tabs	A div before ".webix_el_tabbar". By default 100% wide and 1px tall.
.webix_after_all_tabs	A div after ".webix_el_tabbar", separating it from the content below. By default 100% wide and 6px tall.

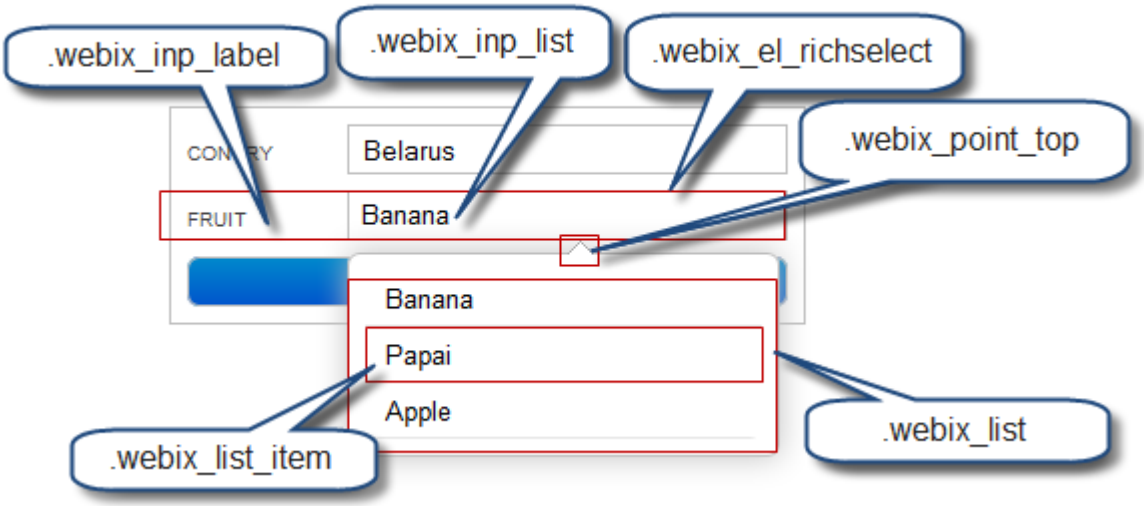
All the classes are applicable for both **top** and **bottom** tabbar except for:

- *.webix_all_tabs.webixtype_bottom* - class specific to bottom tabbar only;
- *.webix_tab_filler* - appears in the bottom tabbar only if total width of all tabs is less than available space.

Read more about Tabbar control.

Richselect

CSS Image Map

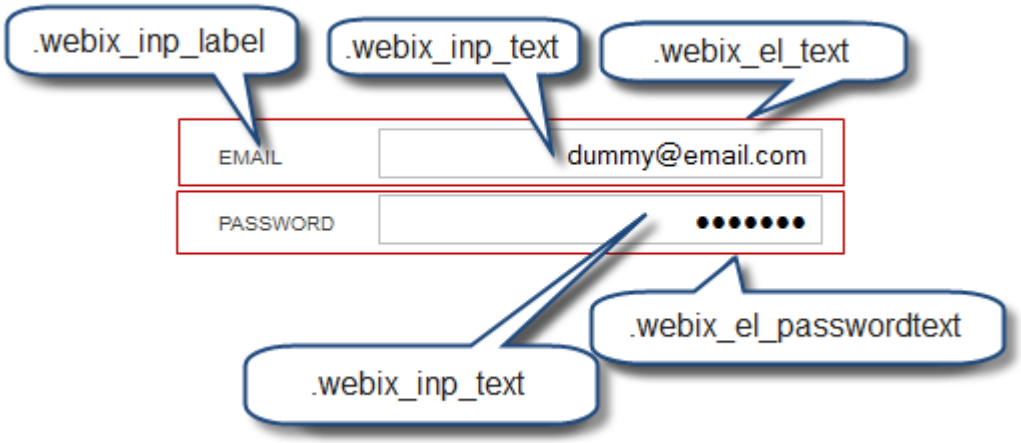


css class	Element
.webix_el_richselect	the whole 'richselect' control
.webix_inp_list	the input for the 'richselect' control
.webix_inp_label	the label area
.webix_point_top	the top arrow of the drop-down list
.webix_list	a drop-down list
.webix_list_item	an item of the list

Read more about Richselect

Text

CSS Image Map

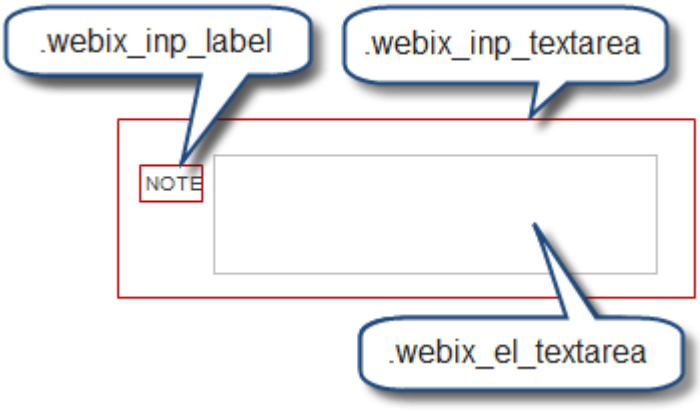


css class	Element
.webix_el_text	the whole 'text' control
.webix_inp_text	the input area
.webix_inp_label	the label area
.webix_el_passwordtext	the whole control, type 'password'

Read more about Text

Textarea

CSS Image Map

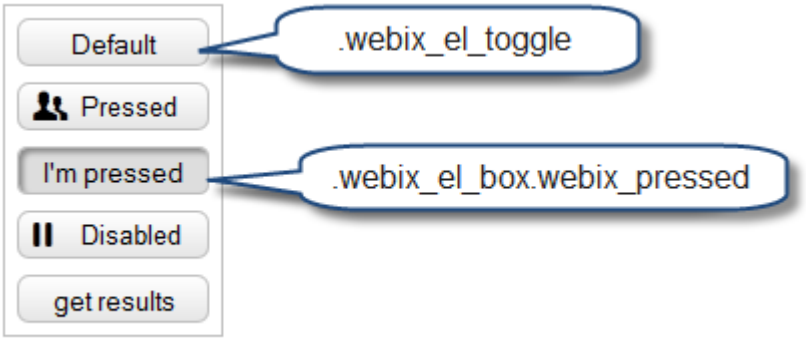


css class	Element
.webix_el_textarea	the whole textarea control
.webix_inp_label	text label for the control
.webix_inp_textarea	the input field of the textarea

Read more about Textarea

Toggle

CSS Image Map



Toggle is a two-state button, that's why it shares its types and CSS classes with the button control. Refer to its Image Map to find details while here toggle's specific classes are listed:

css class	Element
.webix_el_toggle	the toggle control
.webix_el_box.webix_pressed	text label for the control

Read more about Toggle

Intermediate

Animation

Animation Config

Animation is implemented within multiview and pager components. It sets the way the views will be changed as you switch between views or page through the component.

There exist two animation types with their own subtypes. Or, instead, you can choose on of the four animation **directions**:

- **slide** - in, out, together;
- **flip** - horizontal, vertical;
- **direction** - left, right, top, bottom.

Animation in Multiview and Pager

Multiview uses **slide:"together"** animation type by default.

Animation can be **switched off**:

```
{view:"multiview", animate:false }
```

Enabling Animation

1 . The simplest means is including *animate* property into the multiview or pager constructor and define the necessary type and subtype:

```
webix.ui({
   view:"multiview",
   animate:{
      type:"flip",
      subtype:"vertical"
   },
   cells:[]
});
```

2 . The values for **animate** property can be assigned directly out of the multiview constructor. If some values have already been defined with the help of the above-mentioned method, direct assignment will rewrite them:

```
$$("multi").config.animate.type = "flip";
$$("multi").config.animate.subtype = "vertical";
```

3 . By means of the **show()** method.The changes will be applied once while showing the stated view. Then, the default values (initial or 'rewritten') will be used.

```
$$(id).show({type:"flip", subtype:"horizontal"})
```

RELATED SAMPLE: Animated Multiview

RELATED SAMPLE: Paging animation types

Animating View Initialization

The moment of webix View initilization can be animated as well. It works for views created dynamically in the existing Webix layout.

To instantiate the component with animation, you should call **webix.ui.animate()** method instead of standard **webix.ui()**:

```
webix.ui.animate(obj, parent, config);
```

where:

- **obj** (obj) - configuration of the new component;
- **replace** (string, number) - ID of the component that will be replaced by the one stated in **obj**;
- **config** (obj, optional) - animation configuration, described above. If not defined the *{ type:"slide", direction:"left" }* animation will be used.

```
 webix.ui.animate({
    id:"aboutView", template:"About page...",
}, $$("listView"));
```

RELATED SAMPLE: Manual view recreating

Intermediate

Icons with UI Components

Icons can be nested into all components and their items. You can put icon on the toolbar in form of buttons, decorate tabbar tabs, accordion panes as well as list and dataview items with them, etc.

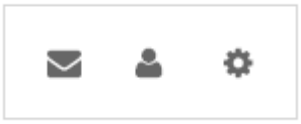
Icons can be initialized in three ways:

- as **separate views**;
- as **button types**;
- as **CSS classes**.

Icons are taken from the Font Awesome collection. There you can find plenty of icons to your choice and pick their names to insert them in your apps.

RELATED SAMPLE: Icons

View Icon



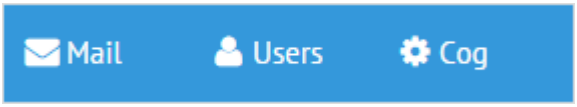
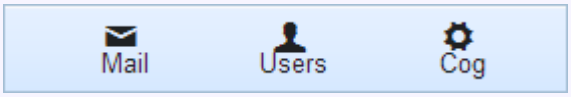
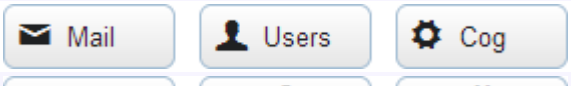
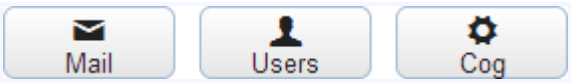
UI-related icon looks like a button with a border. Text label for this icon type is not supposed.

```
{ view:"icon", icon:"mail"},
{ view:"icon", icon:"users"}
```

The value for an **icon** property is the name of the needed icon.

Button with an Icon

Table 1 Icon types

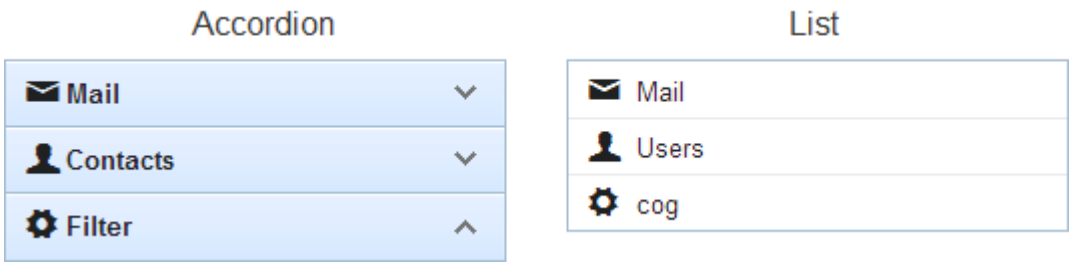
type:"icon"	
type:"iconTop"	
type:"iconButton"	
type:"iconButtonTop"	

```
{ view:"button", type:"icon", icon:"mail", label:"Mail", width:80 }
```

- **type** - the property that turns a standard button into an icon button:
  - **icon** - an icon button without a distinct border;
  - **iconTop** - an big borderless icon button with a text label below an icon;
  - **iconButton** - a small icon button with a label right of the icon;
  - **iconButtonTop** - a big icon button with a label below the icon;
- **icon** - name of an icon (looks for it on Entypo);
- **label** - text label for an icon button;
- **width** - width of a button with label and icon;
- **badge** - an orange circle that notifies about new messages.

'webix-icon' CSS Class

This method of icon initialization fits the components and items that don't normally take buttons - tabs (they are buttons themselves), panes, list and dataview items, datatable cells. In this case, icons are not buttons and perform a decorative function.



Possible CSS classesare as follows:

- 'webix_icon **fa-envelope**' - a standard icon (white or black depending on current skin);
- 'webix_input_icon **fa-envelope**' - a light grey icon adjusted for use in input fields.

Both of them include an icon name that can be checked on Font Awesome site.

Throught css property:

```
Multiview tabs

{ view:"tabbar", options:[
    { value:"mail",  label:"Mail", css:"webix_icon fa-envelope"},
    { value:"users", label:"Users", css:"webix_icon fa-users"},
    { value:"cog",   label:"cog" , css:"webix_icon fa-cog"}
]}
```

By adding extra HTML:

An additional HTML element with an 'icon' class can be added to the text value of any component item:

```
AccordionItem pane with an icon
```

```
{ header:"<span class='webix_icon fa-envelope'></span>Mail"}
```

```
List item with an icon

{ template:"<span class='webix_icon fa-#icon#'></span> #name#"}
```

Here the icon's name is in hash signs since it comes from the dataset, e.g. in JSON it looks like

```
var data=[
    {id:1, name:"item1", icon:"envelope"},
    {id:2, name:"item2", icon:"users"}
]
```

Changing Default Icons

Default FontAwesome icons that are used for various controls can be changed to custom ones in several ways.

Setting Other FontAwesome Icon

If you'd like to use another icon from FontAwesome collection instead of the default one, specify its name as value of **icon** property:

```
{view:"richselect", label: 'Other', options:options, icon:"caret-down"}
```

As a result, the default richselect) icon, "fa-angle-down", is replaced with the one you've set.

Setting a Custom Icon

To set a custom icon that you can provide as a background-image, write the new CSS rule that redefines the default one. Nothing is changed in the control configuration:

```
{view:"combo", label: 'Fruit', options:[...] }
```

As with richselect, combo control uses "fa-angle-down" icon which CSS should be modified:

```
.webix_input_icon.fa-angle-down:before{
    content:""; /*removes FontAwesome icon */
    background-image: url("search.png");
    width:20px;
    height:20px;
    display:block;
}
```

RELATED SAMPLE: Styling Combo Icons

If you want to style one combo on the page while leaving another one unchanged, define a separate **css** class

for the one in question:

```
{view:"combo", label: 'Fruit', options:[...], css:"custom" }
```

Then the necessary CSS will look like this:

```
.custom .webix_input_icon.fa-angle-down:before{
...
}
```

Component styling is described in detail in the corresponding article of the documentation.

Beginner

Setting Borders to the Components

Webix allows setting **borders**, **margin** and **padding** to the components and layout blocks. They should be specified within layout blocks and components to which you wish to apply the settings.

Within the form component borders are set in the same way.

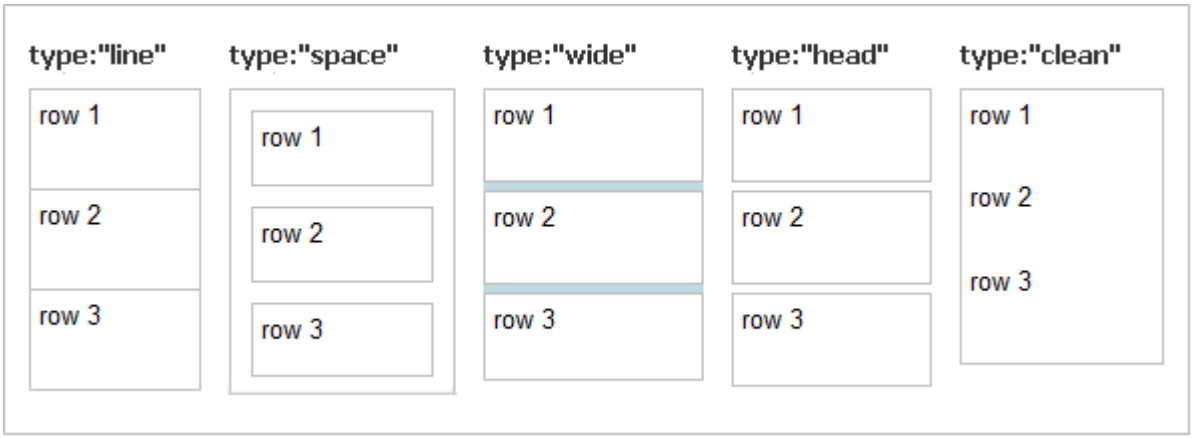
Border Types

Borders can be applied to:

- **layout blocks** (rows, cols, cells);
- the **whole layout** and its variations (**multiview, tabview**);
- any **UI component**;
- **part of UI component** (window or accordion body, head).

Border type is set by the **type** parameter.

```
webix.ui({
    type:"space",
    rows:[
        { ... },
        { ... }
        ]
});
```



By default rows and columns feature thin borders, which can as well be set manually as **type: 'line'**. To completely rid of borders you should either set a **'clean'** type or add **borderless**:*true* parameter.

Note that the chosen border, or borderless state is applied only the the layout block it's defined for it's children still feature the default border. You should set a border for them manually:

```
webix.ui({
    container:"divD",
```

```
    type:"space",
    rows:[
        { template:"text", borderless:true },
        { ... }
    ]
});
```

Here layout rows feature **space** border while a template that lies in one of these rows is **borderless**.

Margin and Padding

Margin specifies space from the view to the left inner border of its parent container (set "margin-left" property) while **padding** sets the offset from the component contents to its borders.

```
webix.ui({
    type:"space",
    margin:30,
    padding:20,
    cols:[{ }]
});
```

Padding can be set separately in horizontal and vertical modes:

- padding - offset for all the borders;
- paddingX - offset for left and right borders;
- paddingY - offset for upper and bottom borders.

Interacting with Users

- Creating Message Boxes
- Setting Progress Bar and Overlay Messages
- HTMLForm and Form Treatment
- Uploading Files
- Uploader and Form Integration
- Controls Guide
- Event Handling
- Keyboard Events. Focusing. Navigation
- Adding Active Elements to Components

Intermediate

Creating Message Boxes

Webix UI library helps the app communicate with users with the help of messages that can warn about error, log events, display variable values, etc.

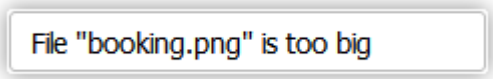
Info Message Box

Basic message is created with the help of **webix.message()** function that has message text as parameter (short form).

```
webix.message("All is correct");
```

Such message is displayed in the top right corner of the window and doesn't prevent the work of the parent app while modal message boxes (described later) do.

Table 1 Message boxes

Message	Image
Default	
Error (type:'error')	

RELATED SAMPLE: Validation with a Complex Rule

Message Types

As seen above, message boxes can be of two types:

- **default** (white background);
- **error** (red background).

The latter has **type** argument in addition to message **text** and features an **extended form** of initialization:

```
webix.message({type:"error", text:"Form Data is Invalid"});
```

Expire Interval

By default webix message disappears either on mouse clicking somewhere outside it, or in 4000 milliseconds after appearing.

To change the expire interval or cancel it - use the **expire** parameter:

```
webix.message({
    type:"error",
```

```
    text:"Form Data is Invalid",
    expire:10000
    //expire:-1   for cancelling expire period
})
```

If you cancel expire period, such message will disappear only on mouse click.

Message boxes can as well be hidden **programmatically**:

```
var message = webix.message("Hi!");
webix.message.hide(message);
```

Modal Message Boxes

Message boxes resemble ui-related modal Window yet they are initialized in a completely another way. They prevent the workflow on the parent app until you perform actions required by them (button clicking, as a rule). Message boxes close on a button click and callback function, if any, is executed.

Message boxes contain some text and an "OK" button. There exist three types of modal message boxes:

- webix.alert() - alert box with a button
- webix.confirm() - confirmation box with two button to agree or cancel.
- webix.modalbox() - a modal message box with as many buttons as you like.

The boxes share some common properties, namely:

- **title** - text of the header;
- **ok** - text of the "ok" button;
- **cancel** - text of the "cancel button" (only for a confirmation box);
- **text** - text of the window body;
- **callback** - function that is executed on button click. Callback function may include another message box which contents depends on your choice in the previous step.
- **type** - type of the button (warning or error).

The callback function takes **result** of user communication with a message box as parameter. It can be:

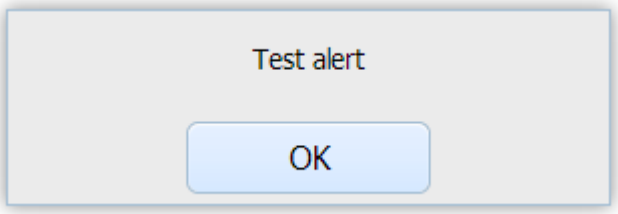
- **boolean true** - each time you press its **"OK"**, *true* is passed to the box callback. It's the only possible value for aler box as it feature an "OK" button only;
- **boolean false** - each time you press **"Cancel"**, *false* is passed to the callback. Works for confirm box;
- **Button index** - true for a modal box. Stored the index of a clicked button (zero-based numbering).

Result is used in callback function that defines further actions according to its value (*true* or *false*).

At the same time, callback can be set not depending on the result, as a fucntion to be executed on button click.

Initialization

webix.alert



The text of its only button is defined as value of **ok** parameter. Passing result into callback is optional.

```
Short form

//the shortest form
webix.alert("Text");

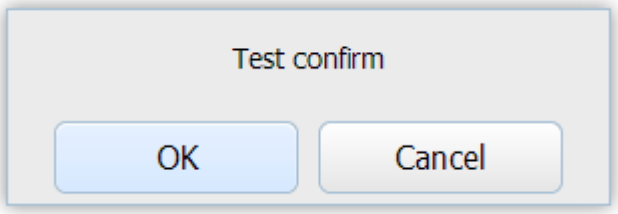
//short form with callback
webix.alert("Test alert", function(result){....});
```

```
Long Form

webix.alert({
    title:"Custom title",
    ok:"Custom text",
    type:"alert-warning",
    text:"Warning",
    callback:function(){...}
});
```

RELATED SAMPLE: Alert message

webix.confirm



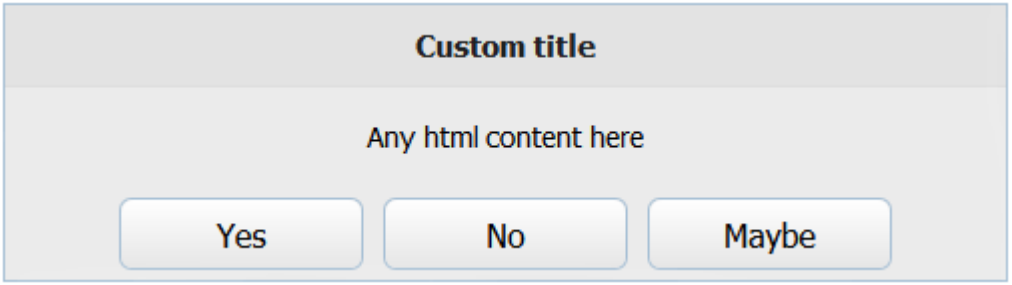
Confirm message boxes contain two buttons - "ok" and "cancel", which text is defined in the same-name properties.

```
webix.confirm({
    title:"Custom title",
    ok:"Yes",
    cancel:"No",
    type:"confirm-error",
    text:"Test confirm",
```

```
    callback:function(result){ //setting callback
        webix.alert({
            title:"Your Choice",
            text:"Result" +result //using callback
        });
   }
});
```

RELATED SAMPLE: Confirm message

Modal Box



Webix modalbox resembles alert and confirm in its modality, yet features several peculiarities:

- its **text** can include any **HTML** content;
- it may contain as many button as you wish defined in **buttons** array that contains text values for each one;
- the **callback** takes the **index** of the chosen button as parameter;
- it can be **sized** with width and height parameters.

```
webix.modalbox({
    title:"Custom title",
    buttons:["Yes", "No", "Maybe"],
    text:"Any html content here",
    width: "500px",
    callback: function(result){
        switch(result){
        case :
            //statement
            break;
        case 1:
            //statement
            break;
        ...
        }
    }
});
```

RELATED SAMPLE: Modal box

Modal Windows and Keyboard Interaction

Keyboard functionality for modax boxes is controlled by **webix.message.keyboard** property that is initially true.

By default modal boxes block keyboard events of the page. Users can use obly the following keys that set modal box value and close it:

- **"space"** and **"enter"** for setting **true** value as modal box result;
- **"escape"** for setting **false** value as modal box result.

To enable keyboard events (and disable above mentioned keys) you should set **keyboard** property to false:

```
webix.keyboard.message = false;
webix.modalbox({...})
```

From now on, user gets a possibility to use full keyboard, for instance, for typing values into inputs inside modal boxes.

Setting Progress Bar and Overlay Messages

Webix allows visualizing the application workflow by adding the following elements to its components:

- progress bar;
- progress icon;
- overlay box with or without message;

Setting Progress Bar/Icon

Progress bar and icon are provided by Webix **Progress Bar** module that can be easily mixed into any componenet including layouts.

```
webix.extend($$("view_id"), webix.ProgressBar);
```

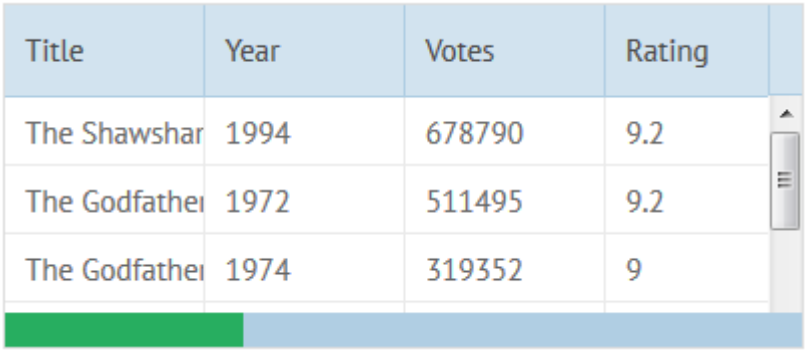
Since now, the component with "*vew_id*" ID features two methods:

- showProgress(config) - shows the progress bar/icon according to the passed configuration, or default one if the configuration is not provided;
- hideProgress - hides progress bar/icon that was shown by the above mentioned method.

Configuration (progress_config) for a progress bar/icon includes:

- **type** (string) - type of a progress element:
  - "top" - progress bar at the top of the view;
  - "bottom" - progress bar at the bottom of the view;
  - "icon" - progress icon at the center of the view;
- **icon** (string) - icon name (only if *type:"icon"* is set). Icon names can be checked at FontAwesome);
- **delay** (number) - life time in miliseconds of the progress bar, after which it is hidden (if **hide** parameter is enabled);
- **hide** (boolean) - specifies whether a progress bar/icon shold be hidden after its life time.
- **position** (number) - an optional parameter, true only for the progress bar. Specifies the position of a progress mark on a bar. Varies from 0 to 1.

Progress Bar for the Data Component




```
Progress Bar

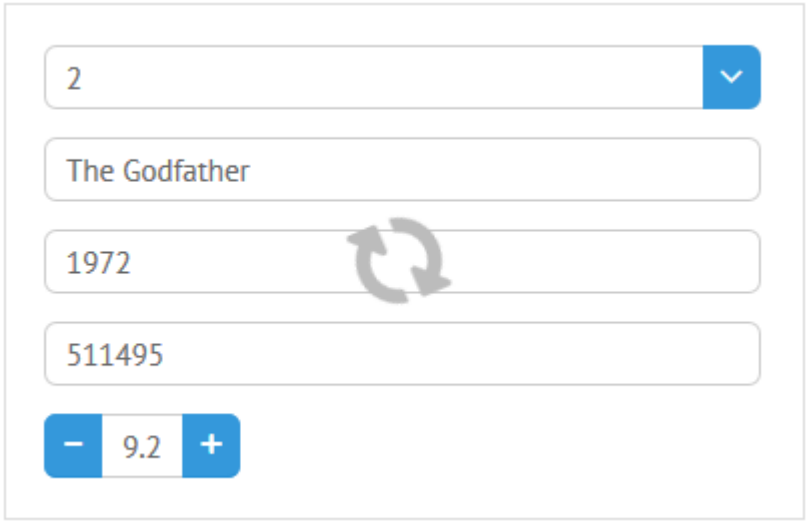
//initializing the component
webix.ui({
    view:"datatable", id:"data", ..
});

//adding progress bar functionality to it
webix.extend($$("data"), webix.ProgressBar);

...
//using the functionality
$$("data").showProgress({
    type:"bottom",
    delay:3000,
    hide:true
});
```

RELATED SAMPLE: Progress Bar::Data

Progress Icon for Layout




```
Progress Icon

//initializing the layout
webix.ui({
    id:"app", rows:[
```

```
        ..app config..
    ]
});

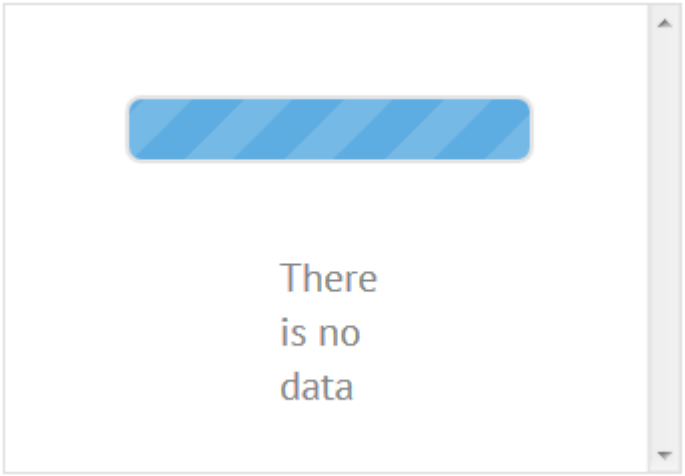
//adding progress bar functionality to it
webix.extend($$("app"), webix.ProgressBar);

...
//using the functionality
$$("app").showProgress({
   type:"icon",
   delay:3000
});
```

Additionally, an app can be disabled for the time being.

RELATED SAMPLE: Progress Bars::Layout

Setting Overlay Boxes to Components



RELATED SAMPLE: Empty list

Overlay box is provided by Webix OverlayBox module and allows creating placeholders for data components.

The module needs to be inluded into the component:

```
webix.extend($$("list1"), webix.OverlayBox);
```

After that the following methods become accessible from the component:

- **showOverlay**(text) - shows overlay box with the defined text or html;
- **hideOverlay()** - hides the overlay box;

A blue bar with **webix_loading_overlay** CSS class will be shown. To get rid of it, or set another loading image, redefine this class.

```
List with 'no data' message

webix.ui({
    view:"list",
    url:"...",
    ready:function(){
        if (!this.count()){ //if no data is available
            webix.extend(this, webix.OverlayBox);
            this.showOverlay("<div style='...'>There is no data</div>");
        }
    }
});
```

RELATED SAMPLE: Empty list

Component-specific support of overlay boxes

Webix datatable features prebuilt overlay support.

Related Articles

- Component Copying and Extending.

Intermediate

HTMLForm and Form Treatment

The most commonly used methods here include:

setValues()

Sets all the form values in one scope. Its argument is an object than contains 'property-value' pairs where new values are assigned to form elements.

Form elements (text fields, radios and checkboxes) are referred by either their **names**.

```
webix.ui({
    rows:[
        { view:"form", id:"myform", elements:[
            { view:"text", name:'field_a', label: 'from', value:
"Madrid"},
            { view:"text", name:'field_b', label: 'to', value: "Milan"}
        ]},
        { view:"button", label: 'Fill form', click:"set_form" }]
});

function set_form(){
    $$('myform').setValues({
        field_a: "London",
        field_b: "New York"
    });
};
```

There's a second parameter for the function that is responsible for **further form updates**.

By defaul, it is **false,** which means that if you apply this function with a value just for one field, it will override the previously set values and they will be lost in nowhere. set to **true**, it allows for updating exisiting values with new ones.

```
$$('my_form').setValues({ field_b:"Paris" }); //the value for filed_a is
lost!

$$('my_form').setValues({ field_b:"Paris" }, true);
// the form is updated with the specified value
```

Check also the API reference for this method.

load()

Loads values from a data file. Here you should pass the path to a data file as well as data format into the

function.

```
function load_form() {
    $$('myform').load("./data/book.xml", "xml");
}
```

getValues()

Gets all the form values including those of hidden and disabled fields. In the snippet below form values are pasted into the console log.

```
function get_form() {
    var values = $$('myform').getValues();
    console.log(values);
}
```

You can also specify whether to include values of disabled and hidden fields into the result:

```
$$('myform').getValues({hidden:false, disabled:false});
```

Check also the API reference for this method.

clear()

Clears all the form fields.

```
function clear_form() {
    $$('myform').clear();
}
```

focus()

Sets the focus into the desired field. If used without arguments sets focus into the component it's called from. In case the item's ID is passed into the function, the corresponding item gets focus.

```
<input type="button" value="Focus author" onclick="focus_form('author');"
/> //htmlform
<input type="button" value="Focus genre" onclick="focus_form('genre');" />

...
function focus_form(item) {
    if (!item)
        $$('htmlform1').focus();
    else
        $$('htmlform1').focus(item);
}
```

RELATED SAMPLE: Initialization in Document Body

isDirty()

Checks whether changes within form were made. The method is called before validation the form and saving form values and/or sending them to server.

```
function save_form(){
    var form = $$('form1');
    if(form.isDirty()){
        if(!form.validate())
            return false;
        form.save();
    }
 };
```

RELATED SAMPLE: Binding to List

Check the API reference for this method.

setDirty()

Marks the form as 'edited'.

```
$$('form1').setDirty();
```

Check the API reference for this method;

bind()

Binds the form to other component. In the snippet below list becomes a datasource for the form as well as helps save data from the form into the component.

More info on Data Binding

```
<div>
    <input type="number" name="rank" />
    <input type="text" name="title" />
    <input type="date" name="year" />
</div>
<script type="text/javascript">
webix.ui({
    view:"list",
    template:"#rank#. #title# Year:#year#"})

$$('htmlform1').bind($$('list1'));
//on clicking any list item the form is filled with the same values
</script>
```

RELATED SAMPLE: Binding to a native HTML form

validate()

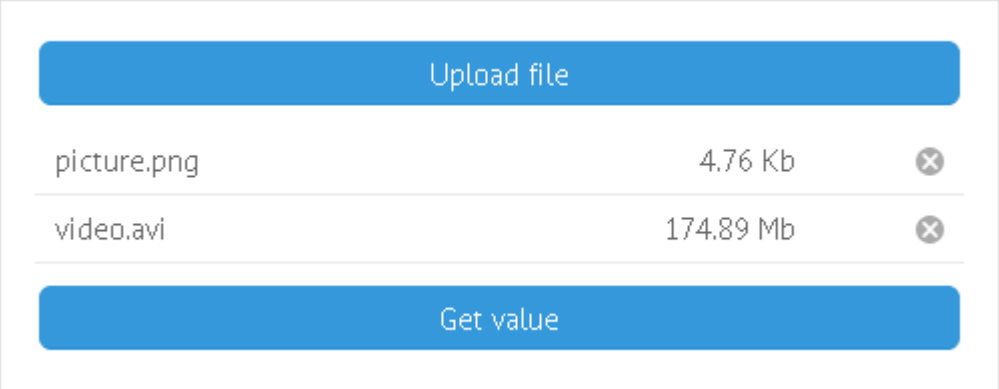
Checks where the input data complies with the defined rules for the form.

More info on data validation.

Don't forget to specify the ID of a form / htmlform to enable this functionality since the ID is used to call these functions.

Uploading Files

Files are uploaded with the help of dedicated UI component called **uploader.**



RELATED SAMPLE: Basic List Uploader

Webix uploader inherits from view and looks like a standard Webix button you need to click to start uploading.

The button can as well be made invisible and file uploading dialog can be started on any page event (except for IE8).

Uploader Initialization

Uploader is a control that looks like a standard button yet triggers an upload window on its clicking.

Uploader is normally initialized together with the UI component to display values of pre-uploaded and uploaded files. Such component (it may be list or template, etc.) is called **uploader body** and is inited separately. To link it to the uploader, specify its **ID** as value of uploader **link** property.

```
webix.ui({
    view:"form",
    rows:[
        {
            view:"uploader",
            id: "uploader_1",
            value:"Upload file",
            link:"mylist",
            upload:"php/upload.php"
        },
        {
            view:"list",  id:"mylist", type:"uploader",
            autoheight:true, borderless:true
        }
    ]
});
```

Uploader Properties:

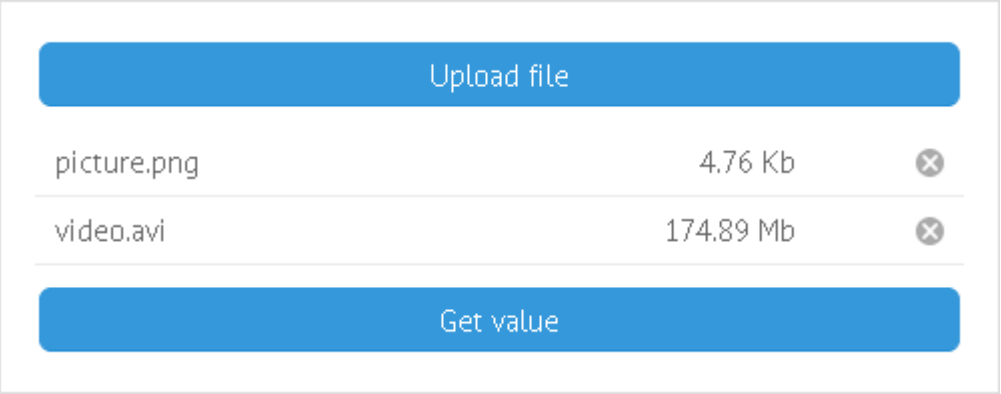
- uploader **ID** should be specified in case you are going to work with it later;
- **value** (string) - text value of the uploader button;
- **link** (string) - the ID of a component to display values of chosen files;
- **upload** (path) - path to a script file that will handle uploading;
- **multiple** (boolean) - *true* by default. Set to *false*, it enables removing the previously added file and replacing it with a new one so that you can upload only one file at a time;
- **autosend** (boolean) - *true* by default. Set to *false*, it cancels on-the-go file processing. The files are added to the uploader body only. They can be sent to server by calling the send() method later on.

Read more about **uploader configuration** and **server script**.

Uploader Display Modes

The library offers Webix list as default component to show properties of uploaded files. You can as well init any Webix component for these needs provided that you specify an appropriate **template** for it.

List Uploader

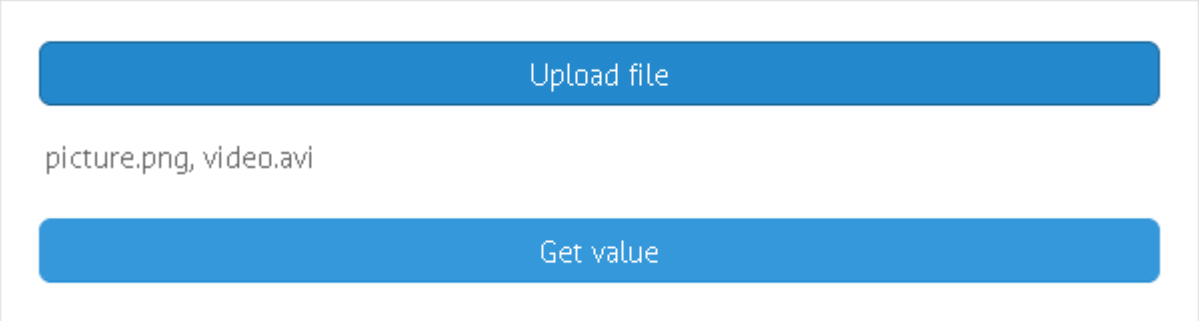


Set to **type:"uploader"**, list shows the following file info:

- **file name**;
- its **extension;**
- file **size**;
- **Cancel** icon;
- **"Upload Ready"** icon (isn't shown initially for *autosend:false* mode.)

You can as well create a **custom type** for list items. It's described in an Uploader Progress Bar article.

Template Uploader



To make template displays file properties, you should define a function template to form a display string.

In the snippet below, template shows file properties in a string comprised of file names joined together with a comma.

```
view:"template",
template:function(data){
    var names = [];
    if (data.each)
        data.each(function(obj){  names.push(obj.name);  });
        return names.join(", ");
}
```

RELATED SAMPLE: Template Uploader

Template uploader is really good for attaching files to some message.

Manipulations with Files

The properties of the uploaded files are stored in the uploader **files** Data Collection that features an array of file objects.

To get to all files currently added to the uploader body, apply the following code:

```
//"upl1" - uploader ID
$$("upl1").files.data.pull;
```

The line returns an array of data items (files). Each file item contains information about its size, type, status, etc.

Each file object includes:

- **id** - file ID generated automatically with the **webix.uid()** method per each uploading session;
- **name** - file title and extension;
- **type** - file extension only;
- **sizetext** - size of the file (as string like 522Kb);
- **size** - size of the file in bytes (as number);
- **status** - upload status that changes during uploading:

- "*client*" - file is added to the uploader body on client-side, the uploading hasn't been started yet or was aborted by user;
- "*transfer*" - file uploading is in progress;
- "*server*" - file has benn successfully uploaded;
- "*error*" - an error occured during uploading.

- **item(*id*).context** - the object uploader is connected with.

Item ID's are random numbers. They can be retrieved with **getFirstId()**, **getLastId()** and **getNextId()** methods.

Getting to Each File Object

File object can be retrieved by the **getItem** method with an ID of the necessary item as parameter:

```
var file_id = $$("upl1").files.getFirstId(); //getting the ID
var fileobj = $$("upl1").files.getItem(file_id).file; //getting file
object
file_name = $$("upl1").files.getItem(file_id).name; //getting properties
//or
filename - fileobj.name;
```

Webix offers rich API to manipulate with **files collection** and each **file object**. For instane, you can:

```
$$("upl1").files.parse(); //parse json data into files collection

$$("upl1").files.getFirstId(); //get ID of the first and last elements
from the collection
$$("upl1").files.getLastId();

$$("upl1").files.getNextId(id);//get ID of the next file according to the
specified one

$$(this.config.uploader).files.remove(id); //remove the specified file
```

Note that when you **parse JSON string** to the **files** object, no files are actually added to the uploader!

```
$$("uploader_1").files.parse([
    { name:"cover.jpg", sizetext:"54kb", status:"server" },
    { name:"page01.jpg", sizetext:"122kb", status:"server" },
    { name:"page02.jpg", sizetext:"142kb", status:"server" }
]);
```

RELATED SAMPLE: Adding files by API

You can get to file names simpler, with the dedicated **getValues()** method applied to the form that contains this uploader. The method returns **name** properties of each file joined into a comma-separated string.

```
webix.ui({
    view:"form", rows: [
        { view: "uploader", name:"records", ...},
        { view:"button", click: function() {
            var text = this.getParentView().getValues();
            text = JSON.stringify(text, "\n");
            webix.message("<pre>"+text+"</pre>");
            }
        }
});
```

RELATED SAMPLE: Template Uploader

Treating File Items in a Cycle

Cancelling uploading of several files at once:

```
function cancel(){
    var id= $$("upl1").files.getFirstId();
    while(id){
        $$("upl1").stopUpload(id);
        id = $$("upl1").files.getNextId(id);
    }
```

Defining function to execute on file uploading:

```
$$("uploader").files.data.each(function(obj){
    if (obj.status == "server")
         //statement
});
```

Array of files can be treated by file order as well:

```
var order = $$("uploader").files.data.order;
for (var i=; i<order.length; i++){
    if ($$("uploader").files.getItem(order[i]).status = "server")
        //statement
}
```

Uploader Events

Here are main events that allow for customized uploader behavior. Full event system is to be found in the API

reference.

- **onBeforeFileAdd** - triggers on adding a file to uploader menu;
- **onFileUpload** - trigers when upload process is started;
- **onUploadComplete** - triggers when file upload is successfully completed;
- **onFileRemove** - triggers when you try to remove a file from a pre-upload list (there appears a dedicated "remove" button as soon as you add a file to the uploader field);

```
Confirming Uploading

$$("upl1").attachEvent("onUploadComplete", function(){
    webix.message("done");
});
```

Related Articles

- Configuring Uploader
- Managing Server Part of Uploading
- Creating a Progress Bar for Uploaded Files
- Uploader and Form Integration
- Binding Uploader to Any Page Element
- Api Reference - Uploader Methods, Properties and Events
- Uploader CSS Image Map

Configuring Uploader

Single upload

The **uploader button** initiates a window for browsing files on your machine and choosing the necessary one. By default, uploader supports uploading of multiple file at once and you can push this button and add as many file as you like to the uploader.

If **multiple** property is set to *false*, it enables replacing of the previously added files with new ones.

```
webix.ui({
    view:"uploader",
    multiple:false,
    ...
});
```

RELATED SAMPLE: Single Upload

Manual send

By default, files are added to the uploader and get processed with the dedicated script at once. To prevent this, cancel the **autosend**:

```
webix.ui({
    rows: [
        { view:"uploader",
          autosend:false,
          ...
        },
        { view:"button",
          click: "$$('uploader_1').send()";
        }
    ]
});
```

RELATED SAMPLE: Sending on custom action

Files are send later when send() function is called for this uploader. It triggers upload script execution.

The benefit of manual sending are as follows:

- you can **vizualize** the **response of uploading script** on client side;
- you can enable **consecutive saving** of **uploader** and **form** data if a form contains extra field exept for uploader.

Defining additional data to send

Uploader control allows defining **extra data** to send to server together with the file data.

In the autosend mode

```
webix.ui({
    view:"uploader",
    formData:{
        param1:value1,
        param2:value2
    },
    ...
});
```

During manual sending

The above metioned code can be used for manual sending as well, but alternativaly you can provide extra as the second parameter of the send method:

```
//either when a single file is sent to server
$$("uploader").send(id, { param1:"value1", param2:"value2"});
```

The id parameter is file ID in the uploader files array.

```
//or with callback function as first parameter
$$("uploader").send(function(res){
        /*callback function*/
    },
    { param1:"value1",
      param2:"value2"
    }
);
```

Uploading particular file types

Uploader can be customized to accept only particular file types during the uploading. Types of files allowed for uploading are specified in the **accept** property:

```
webix.ui({
    maxWidth:800,
    cols:[{
        view:"form", rows: [
            {
                view: "uploader", value: 'Upload Images',
                name:"files", accept:"image/png, image/gif, image/jpg",
                link:"mylist",  upload:"php/upload.php"
```

```
            }
            ...
        ]
    }]
});
```

In the above example Uploader accepts only files of image type (.png, .gif and .jpg formats). The full list of common media types is given here.

RELATED SAMPLE: Uploader - configure to accept the filetype

Related Articles

- Managing Server Part of Uploading
- Uploader and Form Integration

Managing Server Part of Uploading

Server Script

In eccense file uploading is done via Ajax. Each time **send()** function is executed, Ajax POST request is sent behind the scene. Its **url** is a script you specify as value of **upload** property.

```
{view:"uploader", name:"uploader", upload:"upload.php", ...}
```

The custom script you write yourself on your favourite language, not just PHP. Regardless of the language, the script must contain:

- the desired **destination** for file uploading;
- script **response** in form of a JS string - *{status:"server"}* for success, *{status:"error"}* for failure.

Below **PHP solution** is explained:

Uploader is based on an HTML input, type "file", so files passed to the script will be stored in **$_FILES** array under the uploader **"name"** The properties of an uploaded file can be gotten right here:

```
$file = $_FILES['uploader']; //getting a file object

$file['name']; //name of the uploaded file
$file['tmp_name']; //name of the file in the temporary storage
```

The simplest way to set destination and move the files there is:

```
$destination = realpath('./files'); //define folder
$filename = $destination."/".preg_replace("|[\\\/]|", "", $file["name"]);
//set destination
move_uploaded_file($file["tmp_name"], $filename); //move files
```

Tuning Serverside Response

You must also tune the **script response** to notify you about the result of file uploading, otherwise you won't be able to get notification about upload success or failure.

Thescript ought to **return JSON string** that is later processed by Uploader to change the properties of the corresponding **file object**.

```
if(..some condition..)
    echo "{ status: 'server'}";
else
    echo "{ status:'error' }";
```

Why it's done so? Each file object features **status** that changes during uploading:

- **client** - when the file was just added to uploader and uploading hasn't started yet. Or when uploading was aborted by user;
- **transfer** - during uploading until scriot response;
- **server** - uploading finished, set when script responce is "{status:'server'}";
- **error** - an error has occurred during uploading, set when script responce is {status:'error'}.

The responce is available directly in the **send()** callback that takes it as parameter:

```
$$("upl1").send(function(response){
    if(response)
        webix.message(response.status);
        webix.message(response.sname);
});
```

In addition, script may return any **additional information**, e.g. file storage name encoded with MD5.

```
$sname = md5($file["name"]);;
....
echo "{ status: 'server', sname: '$sname'}";
```

This additional info is added to the file object and can be later on derived just like other properties.

```
$$("upl1").send(function(){
    var f = $$("upl1").files;
    var file_id = f.getFirstId();
    var status = f.getItem(file_id).status; //upload status
    var status = f.getItem(file_id).name; //file name
    var sname = f.getItem(file_id).sname; //storage name
    webix.message("Upload: "+status+" for "+name+" stored as "+sname );
});
```

Multiple Files

In case of **multiple files** uploaded at a time, direct responce won't suit as it contains info about one file.

Instead, start a cycle that will get to each file's value:

```
$$("upl1").send(function(){
    $$('upl1').files.data.each(function(obj){
        var status = obj.status;
        var name = obj.name;
        if(status=='server'){
            var sname = obj.sname; //came from upload script
            webix.message("Upload: "+status+" for "+ name+" stored as
"+sname );
        }
        else{
```

```
            webix.message("Upload: "+status+" for "+ name);
        }
});
```

RELATED SAMPLE: Uploader and Form Integration

Files Data Array

The same can be done applying directly to the uploader **datastore**, through **data** array:

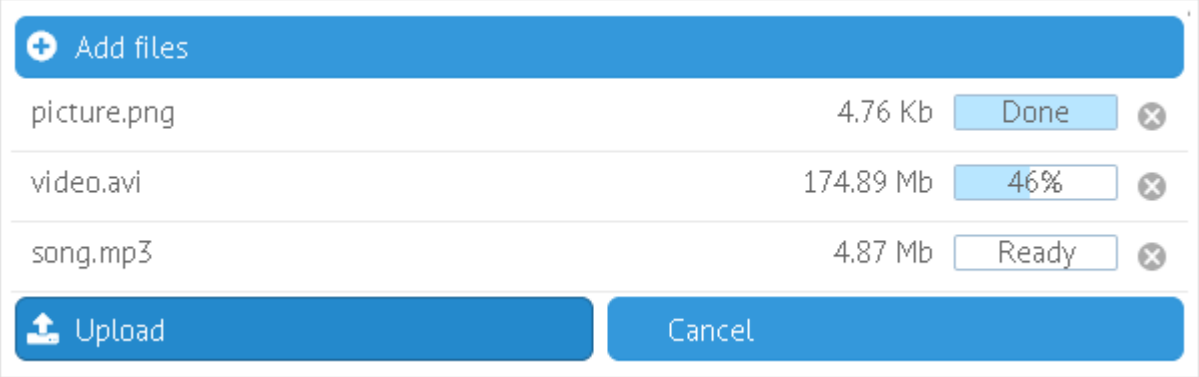
```
$$("upl1").send(function(){
    var text = f.data.pull[file_id].sname;
    webix.message("Test sname: "+ text);
});
```

Or, in a cycle in case of multiple files uploaded at a time:

```
for(var i in f.data.pull){
    var text = f.data.pull[i].sname;
    webix.message("Test sname: "+ text);
}
```

Progress Bar for Uploader

Progress bar is set with **CSS means** for each file separately and is displayed in the dedicated component(e.g. list) set to a custom **type**.



RELATED SAMPLE: Progress bar

Webix.type defines the way component items look. Here you specify the **component** you make the type for, **name** of this type and item **property(ies)** you set.

```
webix.type(webix.ui.list, {
                    name:"myUploader",
                    template: function(f, type){},
                    status:function(f){},
                    on_click:{},
                    height: 35
                    })
```

Template:

As a rule, displayed info contains the following data wrapped in the chosen CSS class (defined in the document head section).

- **f.name** - name and extension of the file;
- **cancel icon** (built-in icon set via CSS class *webix_remove_upload icon-cancel-circle webix_icon*). More About Iconization here;
- **f.status** - "transfer" or "server" with **f.percent** value for the latter;
- **f.sizetext** - size of the file.

```
template:function(f,type){
            var html = "<div class='overall'><div
class='name'>"+f.name+"</div>";
            html += "<div class='remove_file'><span style='color:#AAA'
class='webix_remove_upload icon-cancel-circle webix_icon'></span></div>";
            html += "<div class='status'>";
            html += "<div class='progress "+f.status+"'
style='width:"+(f.status == 'transfer'||f.status=="server"?f.percent+"%":
```

```
"0px")+"'></div>";
            html += "<div class='message "+
f.status+"'>"+type.status(f)+"</div>";
            html +=  "</div>";
            html += "<div class='size'>"+ f.sizetext+"</div></div>";
            return html;
}
```

Status:

Here you put a function that defines status message displayed in the progress bar.

```
status:function(f){
        var messages = {
            server: "Done",
            error: "Error",
            client: "Ready",
            transfer:  f.percent+"%"
            };
        return messages[f.status]
}
```

Functionality for the "cancel" icon.

```
on_click:{ "remove_file":function(ev, id){
                $$(this.config.uploader).files.remove(id);}
}
```

Minor parameters

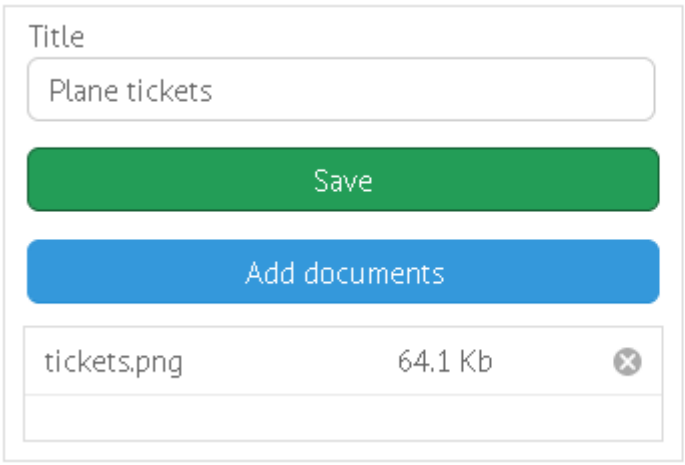
The ones like **width** and **height** of each list item.

Beware to set the created type for the list!

```
{view:"list",  id:"mylist", type:"myUploader", ...}
```

Uploader and Form Integration

The purpose of any form is to collect data from users. This library provides UI Form and Htmlform components with lots of controls while the uploader component helps get user files (text documents, pictures, etc.)




```
Array
(
    [title] => Plane tickets
    [files] => tickets.png
)
```

RELATED SAMPLE: Uploader and Form Integration

Uploader controls as well as the component to display pre-uploaded files is included into the form like any of its other controls:

```
{view:"form", elements:[
    { view:"text", label:"Title", labelPosition:"top", name:"title" },
    { view:"button", label:"Save", type:"form", click:save_form },
    {
        view:"uploader", upload:"php/upload.php",
        id:"files", name:"files",
        value:"Add documents",
        link:"doclist",
        autosend:false //!important
    },
    {
        view:"list", scroll:false, id:"doclist", type:"uploader"
    }]
}
```

Beware to **switch off autosending** to send files together with other form data.

Form Saving

While saving the form, follow the scheme:

- send files to server;
- (optionally) get response of uploading script and use its data. Described here;
- save the form data via Ajax POST request;
- (optionally) get response from a form-saving script and use its data.

Files are processed with a script defined as **upload** value of the uploader (read more) while form **data** is processed with the help of another one. You write both scripts on your favourite language to match your needs.

The Ajax request to send the form is defined as a callback to uploader **send()** method, so yo first get data from uploading script responce and can use it while saving form data.

Scheme of Form Saving

```
function save_form(){
    $$("files").send(function(){ //sending files
        //...callback

        webix.ajax().post(
            "php/saveform.php", //saving form
            $$("myform").getValues(),
            function(text){  //responce
                webix.message(text);
            }
        );
    });
}
```

To learn more about how to send Ajax requests in Webix, read the Ajax Operations article.

Intermediate

Controls Guide

Controls are the most important components for event handling since they enable interaction with users. They are used separately as well as can be nested into toolbar, form, and other components. All ui-related controls inherit from **input**.

List of Available Controls

Button, Checkbox, Colorpicker, Combo, Counter, Datepicker, Icon, Imagebutton, Label, Radio, Richselect, Search, Select, Segmented, Suggest List, Tabbar, Text, Textarea, Toggle

When working with controls, specify their IDs if you want to refer to them later and attach events to them.

Controls share a lot of features that work the same way with most of them or with very little difference.

Control Initialization

Control constructor includes its name and properties defining its look-and-feel and behavior.

Some properties duplicate HTML attributes, some are typically Webix control properties. In addition, you can set necessary HTML attributes to your control within **attributes** object property.

```
{view:"text", width:200, click:"", attributes:{ maxlength:10 }...}
```

Defining Text Label

The **label** (string) property sets a text label for a control, a string placed by the control.

- underwrites the controls and explains its purpose;
- label absence is defined as *label:"null"* or by omitting this property in the control constructor;
- horizontal position of the label is defined by the **labelAlign** (string) property. Its ossible values are: - "left"(default); - "right" (except for button and toggle that takes only left alignment);
- vertical position of the label is defined by the **labelPosition** (string). The only possible value is "top";
- width of the label is defined by the **labelWidth** property;
- text of the button is set by **value** property instead of label.

Please, don't mix it with a separate view label.

RELATED SAMPLE: Positioning Labels

Defining Special Modes

Any control can be set **disabled** and **readonly** with the help of dedicated properties:

```
elements:[
    {view:"text", readonly:true},
    {view:"datepicker", disabled:true}
]
```

Control Positioning

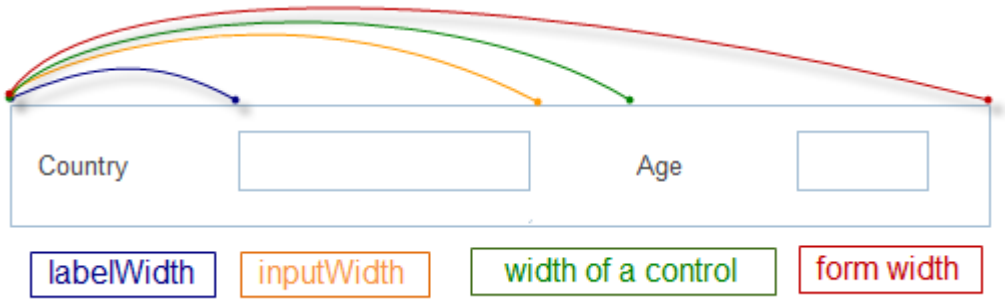
Here we make use of the **align** (string) property that defines horizontal position of the control and its label with respect to the parent container (e.g. form):

- "right";
- "left" (default);
- "center".

RELATED SAMPLE: Aligning Inputs and Labels

Setting Dimensions

Each time control is inited, space is reserved for it. It either equals to parent (form, toolbar) dimensions or is set as control **width**.



In case several controls are in the same row/column, the available space is distributed equally among them.

Total **width** of the control includes **inputWidth** and, therefore, **labelWidth** (study the picture). If you set width less than the combination of them, a part of the control won't be visible. At the same time, you can make width bigger - then, spacing will appear.

The following sizing options are possible:

- **height/width** (number) - dimensions of space reserved for this control including its label;
- **maxHeight/maxWidth** - maximum dimensions of the whole control. During resizing, the control can't

be bigger, but can be smaller;

- **minHeight/minWidth** - minimal dimensions of the whole control. During resizing, the control can't be smaller, but grow bigger;
- **inputHeight/inputWidth** (number) - dimensions of the input area/ button body;
- **optionWidth** (number) - sets the width of an option (segment of tab) for segmented and tabbar. Hence, for such controls **inputWidth = optionWidth * N options**. If this value is smaller than an **width**, a part of the control will disappear;
- **labelWidth** - width of the text label by the control. It can't be bigger than **inputWidth** and, thereby, **width**. If it is, the control will be incorrectly displayed. By default, it features static width of 106 px.
- **gravity** - relative sizing functor for controls placed in the same row/column.

```
//text2 is 3 times bigger than text1
rows:[
    {view:"text",   name:"text1", gravity:1},
    {view:"text", name:"text2", gravity:3}
]
```

Attaching Functions

Controls can trigger any function for any component in the application. Study Event Handling for details.

Inner events of controls include:

- **clicking**;
- **changing their state** (selection made, switching to another tab, adding new data into inputs)

Two major possibilities here include:

1 . Attaching event inside the control constructor.

With the **click** property. It equals to the "onItemClick" event.

```
{view:"button", click:"$$("my_form").save()" }
```

With the **on** property that allows attaching custom handlers to specified inner events of the control:

```
//clicking
{view:"button", on:{
    'onItemClick': function(){...}
    }
}

//state changing
{view:"text",  on:{'onChange': function(){... }
    }
}
```

2 . The same using the **attachEvent()** method.

```
//clicking
$$("my_button").attachEvent("onItemClick", function(){.... });

//state changing
$$("my_text").attachEvent("onChange", function(){.....});
```

The implementation of an **onChange()** function is described in the Form documentation.

Automatic Showing and Enabling of Related Form Fields

Each form field can have one **related** field defined by its relatedView property.

This field can react on changes in master field and change it's state:

- from **hidden** to **visible** and vice versa,
- from **disabled** to **enabed** and vice versa.

This behaviour is also defined by the master field with the relatedAction property:

```
Enabling a form field by changes in another

elements:[
    {view:"text", id:"master", relatedView:"slave",
relatedAction:"enable"},
    {view:"text", id:"slave", disabled:true}
]
```

The following conditions should be observed:

- the functionality allows for watching various value changes (*empty/not empty*, *selected/not selected*, *checked/unchecked*) for different controls;
- **relatedView** property takes ID of the needed control as parameter (if you don't define ID manually, it will be created automatically and can be derived as *view.config.id*);
- **relatedAction** offers two possibilities for now:
  - "enable";
  - "show";
- Slave field may have initial values to match with master relatedAction - *hidden:true* for **"show"** and *disabled:true* for **"enable"**.

RELATED SAMPLE: Automatic form reconfiguration

Defining Initial Values

Single-value controls

For such controls as text and textarea **value** property isn't normally set beforehand, as they are intended for user filling.

So these exist different variants of value treating:

- **omit** this property at all;
- make the property **empty** (*{view:"text", value:""}*);
- set **default value** that can be deleted or changed by users(*{view:"text", value:"My text"}*);
- still it's better to use **placeholder** (the default value for the input, a hint for user. Disappears as you start typing.);
- for two-state controls like checkbox, radio and toggle possible values are:
  - *true* or 1 - for a 'checked' state;
  - *false* or 0 - for 'unchecked' state.
- for counter and slider initial value is normally set as a **possible number** or the one in the middle of the predefined range.

Multiple-value controls

Such controls fall into two groups:

- those presupposing the choice from a pull-down or popup list of **predefined values** (select, combo, richselect);
- those offering the choice between tabs (segments) of integral buttons, namely tabbar and segmented.

These values are set within **options** property either as an array or as a collections of objects.

```
Array of options for Combo box

{ view:"combo", id:'field_m', label: 'Combo', value:"One", yCount:"3",
    options:["One", "Two", "Three"]}
```

The initial value is specified by the **value** property and corresponds to any value from the predefined ones:

```
Collection of objects

 { view:"radio", label:"Branch", value:1,options:[
     { value:"Master",  id:1 },
     { value:"Branch",  id:2 }]
}
```

Th initial value can be set as a **text** value of any option:

```
 { view:"radio", label:"Branch", text:"Master",options:[
     { value:"Master",  id:1 },
     { value:"Branch",  id:2 }]
}
```

If both **text** and **value** properties are defined, the **text** one will be ignored.

RELATED SAMPLE: Radio Button ('radio')

Serverside options for multiple-value controls

Such controls as select, richselect and combo can get values from the serverside. The script that returns the needed data, can be defined directly in the **options** property:

```
view:"combo", options:"server/data.json"
```

RELATED SAMPLE: Advanced options for select controls ('combo' and 'richselect')

Note that **the getValue()** method will return the **ID** of the currently selected tab/segment/option.

Setting Values Dynamically

Values can be as well set after initialization with the help of the dedicated **setValue()** method for a single control or **setValues()** to set the values for controls within the form. Find the latter in the related article.

Both methods take the needed value (or object with 'ID-value' pair(s)) as parameter and are called from the necessary control or the form where the controls are nested:

```
$$('text1').setValue('Anna'); //for a single text field with "text1" ID

$$('form1').setValues( //value for 2 text field in the form
    { text1:"Anna",
      text2:"Brown"}
  )
```

Values of two-state controls(checkbox and radio) can be changed programmatically in different ways:

- **setValue()** - with either *0 (false)* to uncheck or *1 (true)* to check;
- **toggle()** - to change the initial value (the toggle button as well takes this method);

```
$$("mycheckbox").toggle(); //-> if checked - becomes unchecked and vice
versa
```

Getting Values

Within the library we have two methods for getting values:

- **getValue()** - for a component that has one value (any control);
- **getValues()** - for a component that stores a number of controls (form / htmlform)

In case you want to get the value of a separate control, call the **getValue** method from it:

```
{view:"select", id:"my_sel", value:1, options:[
              { id:1, value:"Master" },
```

```
                { id:2, value:"Release" } ]
 }

var value = $$("my_sel").getValue(); //- > returns 1
```

In case controls are embedded into a form, we call the **getValues()** method from this form while specifying **name** or **ID** of the dedicated control.

```
webix.ui({
    view:"form", elements:[
        {view:"text", id:"title", value:""},
        ...//other controls
    ]
})

var title = $$("myform").getValues().title, //-> retuns value of text
input
```

Conrol Value can be different:

- **text string** in the text or textarea control;
- **state** (boolean) of two-state controls (checkbox, radio and toggle);
- **ID of selected tab** of integral buttons (tabbar and segmented);
- **ID of selected option** of selection controls (select, richselect, combo);
- **text** on the button;
- **selected date** or **color item** (datepicker, colorpicker).

In case the **getValue()** method returns object (e.g. date object) and you'd like to see the string result at once, you can include the **stringResult** property into the control constructor:

```
webix.ui({
    view:"datepicker",
    stringResult:true, // false by default
    ..config
});
```

Visibility of Controls

All controls can be shown and hidden on demand:

- one by one applying show() and hide() methods to each of the needed controls or setting hidden property.
- in groups by implementing **batches** functionality. More info.

For **hidden inputs** within the form the library offer peculiar API that actually eliminates the need of creating hidden fields. Here you manipulate with values only.

Styling

Controls can be styled to your taste with the help of CSS Image Maps.

Common Configuration within Form and Toolbar

If you have several instances of one and the same control within a form or a toolbar (a lot of text fields in the form or several buttons on a toolbar) and want them to have the same properties, you can define these properties only once in an **elementsConfig** object.

Find more info in the corresponding article of our documentation.

Hidden Input Fields

Hidden input fields are often included into the form to **set default values** into them, the values that are not visible to users yet required for required to be sent to server They are used for saving state in multi-page forms, collecting cookies or just storing data apart from the filled in.

Within HTML developers make use of the hidden property on the INPUT tag, yet the library offers to omit the unnesessary actions and manipulate just with data.

In fact, you needn't create any hidden text fields. Because all you need is their **values**.

First, create a form with collection of the needed controls and input fields.

```
webix.ui({
    view:"form", id:"my_form",
    elements: [
                { view:"text", label:'Login', name:"field_a", width:150,
labelAlign:"right" },
                { view:"text", label:'Email', name:"field_b", width:200,
labelAlign:"right" }
              ]
})
```

These are visible input fields for users to fill in.

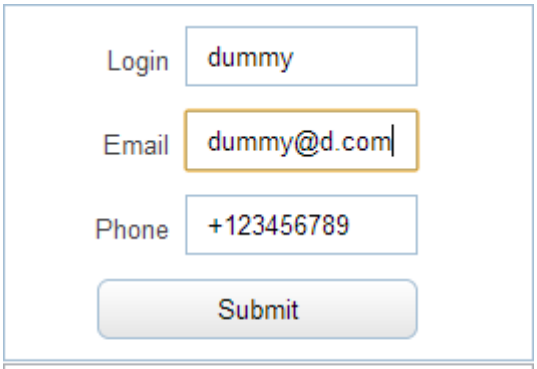
Then, set values for invisible inputs without initializing them. They are automtically created with your values set:

```
$$('my_form').setValues(
            { field_d:"Hidden Info"}
        );
```

When getting values for the filled in form, the function will get values from all the field, regardless of their visibility.

Let's paste them to a textarea to see that it works:

```
{ view:"button", click:function(){
    values = JSON.stringify($$('my_form').getValues(), null, 2);
    $$('log').setValue(values);}
}
```




```
{
  "field_d": "Hidden Info",
  "field_a": "dummy",
  "field_b": "dummy@d.com",
  "field_c": "+123456789"
}
```


RELATED SAMPLE: Hidden Inputs

Common Configuration for Form and Toolbar Controls

UI-related Toolbar and Form are specific components since they don't mean much as they are yet act as containers for controls like buttons and icons, input fields and selection controls.

All the controls are included in a toolbar and form as an array of rows/columns or elements respectively. Each control has its own configuration. However, sometimes some properties like width/height, label alignment and label position coincide, which is done to achieve the uniformity of the component's look-and-feel.

To avoid multiple repetition of one and the same property for controls belonging to one and the same toolbar you can make use of the **elementsConfig** property. Included into a form or toolbar it sets common configuration for all its children (controls).

```
Form Configuration

webix.ui({
view:"form",
    elements: [//..elenemts
        ]
    elementsConfig:{
        labelPosition:"top",
        on:{ onchange:function(newv, oldv){
                webix.message("Value changed from: "+oldv+" to: "+newv);
            })
   })
```

RELATED SAMPLE: Common Configuration

All the controls in this form with have the label on their top and respond to data alteration in input fields.

```
Toolbar Configuration

webix.ui({
view:"toolbar",
        elements: [
                { view:"button"},
                { view:"segmented", options:[...]},
                { view:"text"},
                { view:"select", options:[...]}
                ],
        elementsConfig:{
                width: 150,
                height: 30,
                value:"edit"}
})
```

RELATED SAMPLE: Toolbar: Common Configuration

No matter how different the controls on the toolbar are, they will all get the same height and width as well as the same "edit" value.

Intermediate

Event Handling

Events that occur on the page with a Webix-based application fall into three groups:

- inner events of Webix components ("onAfterSelect", "onChange", "onItemClick", etc);
- global Webix events connected with its application flow;
- native DOM events (can be handled in Webix way as well).

The main part of the article below concerns inner events of Webix components, while the details of the other two groups can be checked by a link.

Attaching Handlers to Inner Component Events

Common inner events for all ui components are **mouse events** (*onItemClick, onBeforeContextMenu, onMouseMoving, onMouseOut*, etc) since all the components can be clicked (and right-clicked) and respond to a mouse pointer.

Other inner events depend on the component functionality and are provided in API reference for each component.

A handler can be attached to a components in different ways:

1 . The most common way is to use **any event** of the component (*check its API ref*) within:

- **attachEvent()** method, or;
- the component's **on** property.

2 . Depending on the component nature, you can attach a handler to some specific events defined by component properties:

- **ready** property of data components;
- **click** property of buttons and inputs;
- the component's data **scheme** object for events connected with data loading;
- some **specific properties** that point to mouse events (*onClick, onContext, onDblClick, onMouseMove*).

Now let's study each possibility in detail.

Using the attachEvent() method

An **attachEvent()** method takes the name of the component's **inner event** and the function it fires (**handler**) as parameters. The syntax is:

```
$$("element_id").attachEvent("onSomeEvent", some_function(){...});
```

Event names are case-insensitive.

```
A button to collapse the all tree nodes

var myEvent = $$("button_id").attachEvent("onItemClick", function(){
    $$("dtree").closeAll();
});
```

Later on, you can easily detach this event from the component with the help of the opposite method:

```
$$("button_id").detachEvent(my Event);
```

Attaching a function in the component constructor

You can also define event handlers right in the component constructor within either of the following attributes:

- **on** - specifies the function that fires when the stated event happens;
- **ready** - specifies the function that fires when the component is ready to use (has been initialized and loaded data, if any);
- **click** - specifies the function that fires when the component is clicked. Works for some form controls like simple buttons and inputs.

```
webix.ui({
    view:"list",
    $init: function(){...},
    ready:function(){
        this.select(this.getFirstId()); //selects the first item
        this.load("data/books2.xml","xml"); //loads data from XML dataset
    },
    on:{
        'onItemClick': function(){
            alert("you have clicked an item");
        },
        'onAfterLoad':function(){ ... }
    }
});
```

When attaching an event you can specify only the name of a function to attach and declare it outside the *webix.ui()* constructor:

```
function select_first(){
    $$("datalist").select($$("datalist").getFirstId();)
};

webix.ui({
    view:"button",
    on:{
```

```
        'onItemClick': select_first
    }
});
```

RELATED SAMPLE: Creating a custom table

For buttons you can make it more compact by using **click** attribute:

```
function close_tree(){...};

{ view:"button", id:"sample_button", value:"Close", width:100,
click:"close_tree"}
```

Attaching events in the data scheme.

Data scheme allows changing the default pattern of data handling within the component.

For instance, the function specified in its **$init** key will run each time when data is loaded/reloaded to the component and when the add method is called.

```
webix.ui({
    view:"list",
    scheme:{
        $init:function(obj){...},
        $update:function(obj){...}
   }
});
```

RELATED SAMPLE: Horizontal List

More possibilities of the data scheme are described separately).

Using specific properties for mouse events

Such properties come in pairs and include:

- **onClick** and **on_click** handlers for clicking;
- **onContext** and **on_context** handlers for right-clicking;
- **onDblClick on_dblclick** handlers for double-clicking;
- **onMouseMove** and **on_mouse_move** handlers for setting a mouse pointer over the component (item).

For instance, you need an event that fires each time you click an item in the list and performs an action that depends on the data derived from this item.

```
    $$("mylist").attachEvent("onAfterSelect",function(id){
       alert(this.getItem(id).title);
    })
```

- those included into the component by default (when select property is true, clicking means selection) and
- those attached with the help of an "onItemClick" event.

Both on_click and onClick can be modified if needed.

1 . With **on_click** redefining can be done only **after** the component initialization.

The *on_click*

```
grid = new webix.ui({
    view:"datatable",
    columns:[
        { id:"", template:"<input  type='button' class='delbtn'
value='Delete'>"},
        {id:"title", ...}
        ],
    on:{"onItemClick", function(){...} //default click behavior
    },
    select:true
});

grid.on_click.delbtn=function(e, id, trg){
    webix.message("Delete row: "+id);
    return false;//blocks default onclick event
};
```

RELATED SAMPLE: Datatable: Custom Handler

One of the columns features an HTML template with a button styled with '*delbtn*' CSS class. All cells including the one with this button will react on default click behavior while when you click on the button itself rather than on its cell area, the default behaviour will be overridden with the function attached to the 'delbtn' CSS class.

2 . The **onClick** allows redefining current click behavior right in the component constructor.

```
webix.ui({
    view:"list",
    template:"#votes# <span class='info'>Details</span>",
    select:"multiselect",
    onClick:{
        info:function(e, id){
            webix.message(this.item(id).title);
            return false; /blocks default onclick event
        }
    },
});
```

When you click list item styled with **info** CSS class, the item won't be selected and custom message will

appear.

RELATED SAMPLE: List: Event Handlers

The same pattern is true for other pairs of properties.

Routing Events from One Component to Another

If you have already attached an event to the component and described the function that this event triggers, you can attach this event to another component in your app.

For these needs, use the **mapEvent** method and pass a **map** into it, where the map is an event-object correspondence:

```
button.mapEvent({
    onItemClick:list // where list - some other component
});
```

As a result, when the button is clicked, the function that was initially attached only to it, will be executed for list as well.

Event Mapping helps to get rid of repetitions in code when one and the same event should be attached to different components.

Event Handling with Component Items

Inner events make it possible to work with component items provided that the **ID** of an item you click, select, etc. is passed to the attached function.

For instance, you need an event that fires each time you click an item in the list and performs an action that manipulates its data.

```
$$("mylist").attachEvent("onAfterSelect",function(id){
     alert(this.item(id).title);
});
```

RELATED SAMPLE: Sizing and events

Blockable Events

Events can be divided into two groups:

- events that don't affect the app's work by their result;
- events that affect the app's work depending with their result. If an event title contains the word "before", the continuation of the current action depends on whether the function returns *true* or *false*.

```
The function returns false, selection is blocked

$$("my_list").attachEvent("onBeforeSelect", function(){ return false; })
```

You can also temporarily block all events assosiated with the component by using the **blockEvent()** method. To re-enable event triggering use the **unblockEvent()** method.

```
$$("component_id").blockEvent();

$$("component_id").unblockEvent();
```

Delayed Events

The default value for response to events is 500ms. With such events as **onMouseMove** and **onMouseOut**, you can delay the server response for as much time (in milliseconds) as you need:

```
webix.ui({
   view:"menu",
   mouseEventDelay:100,
   ...
});
```

Keyboard Events

Components that are in focus at the moment automatically receive the ability to respond to keyboard actions and can take keyboard events, namely

- **onKeyPress** - to respond to keys specified by dedicated key codes;
- **onEditKeyPress** - the same, but with an editor opened;
- **onTimedKeyPress** - to trigger actions after typing has been finished.

Look how the **onTimedKeyPress** event works with a text field to filter data in view list:

```
$$("list_input").attachEvent("onTimedKeyPress",function(){
    var value = this.getValue().toLowerCase();
    $$("list").filter(function(obj){
        return obj.title.toLowerCase().indexOf(value)==;
    });
});
//"list_input" id the ID of the dedicated "text" input control
```

RELATED SAMPLE: List: Filtering

More information about key codes and hot keys is to be found in the UI Manager article.

Global Webix Events

Webix global events are not connected with any specific component and can be used to control general application issues (clicks, touch movements, serverside requests, etc). Some of them repeat native DOM events.

Events connected with Webix components functionality

- **onDataTable**(config, obj) - fires the moment before a Webix datatable is renreded on the page;
- **onEditEnd** - fires when a **popup data editor** (date, color, richselect) is closed in any of the components on the page. See Data editors article for details;
- **onLiveValidation**(editor, result, obj, value) - fires when editing starts in the component with live validation enabled;
- **onSyncUnknown**(obj, source, filter) - fires when a component tries to sync its data with a non-Webix collection;
- **onReconstruct**(obj) - fires when application layout is reconstructed (view is removed, added, replaced). See Dynamic UI Modifications article for details;
- **onLayoutResize**(cells) - fires when application layout is resized (not window).

General page events

- **onClick**(e) - fires when a click/double click happens in any application part;
- **onFocusChange**(newf, oldf) -fires when focus is changed in the application. Its parameters are Webix objects, the one in focus and the one previously focused, or null if non-Webix object is focused;
- **onReady** - fires at the same time with window onload event;
- **unload** - fires at the same time with window onunload event.

Page events in touch environment

- **onRotate**(orientation) - fires when screen is rotated;
- **onAfterScroll**(pos) - fires when some webix view has been scrolled;
- **onTouchStart**(context) - fires on touch start;
- **onLongTouch**(context) - fires on long touch;
- **onTouchEnd**(start_context, current_context) - fires on touch end;
- **onTouchMove**(start_context, current_context) - fires on touch movement;
- **onSwipeX**(start_context, current_context) - fires when a horizontal swipe is detected;
- **onSwipeY**(start_context, current_context) -fires when a vertical swipe is detected.

The details on Touch Events are given in the related part of API reference

Events connected with serverside requests from the page

- **onBeforeAjax**(mode, url, data, request, headers) - fires when any Ajax request is issued. It can be developer request or automatic (when using Data Connectors);
- **onAjaxError**(request_obj) - fires when any Ajax request results in an error;
- **onLoadError**(text, xml, xhttp, obj) - fires when an error has happened during loading the data into a component. See Management of loading errors article for details.

All such events are attached to **webix** object:

```
webix.attachEvent("onRotate", function(mode){
    ..// logic
});
```

Handling of Native DOM Events

Native DOM events can be handled with the help of Webix **event();** and **eventRemove()** functions that are called from **webix** object:

```
var eventId = webix.event("divId", "click", function(e){
    do_something();
});

webix.eventRemove(eventId);
```

Comments:

- **divID** - the ID of an HTML node you want to attach event to;
- **click** - event name (note that they are used here without *on* prefix);
- **handler** - function to run (its **e** argument points to native event object).

Advanced

Keyboard Events. Focusing. Navigation

Each time you create a component on the page, even a single one, **UIManager** module is initialized (though indirectly). Its main tasks are:

- watching which component (and component item) is focused at the moment;
- setting and removing focusing;
- assigning keyboard events to the component in focus;
- memorizing the component's outer scheme.

Focusing Methods

You can set and remove focus programmatically with the help of two opposite methods - **focus()** and **blur()** that are called from the component object. It can be specified in the component's **ready** property to set/remove focusing on page loading.

The **focus()** method is either used without parameters or takes an **ID** or **name** of an item as an argument. In the latter case, focus is set to this item rather to the whole component.

```
$$('my_toolbar').focus(); // the toolbar is focused
$$('my_toolbar').blur(); //the toolbar is no longer in focus

$$('my_toolbar').focus('my_text');
// a text input with ID/name = "mytext" on this toolbar is focused

$$("form").focus(); //the first focusable element in the form is focused
```

Through UI Manager you can control focus with the following methods that are called from UIManager with the ID of the needed view as an argument:

- **getFocus**() - returns the view object that is currently focused;
- **setFocus**(id) - sets focus into the specified location;
- **hasFocus**(id) - checks whether the component is in focus and returns *true* or *false* respectively
- **canFocus**(id) - checks whether the component can take focus. Invisible (hidden) views and their items as well as disabled views cannot be focused.

```
Focusing an item with ID 'books'

webix.ui({
    id:"books",
    view:"list"
    ...
});
```

```
webix.UIManager.setFocus("books");
```

Focusing Events

Every Webix component features a pair of focusing events **onFocus** and **onBlur**:

```
$$("datatable1").attachEvent("onFocus", function(current_view, prev_view){
    //current_view if the datatable in question
});

$$("datatable1").attachEvent("onBlur", function(prev_view){
    //prev_view if the datatable in question
});
```

In addition, Webix **onFocusChange** global event) is triggered each time focus is shifted from one component to another. The following code retrieves the ID of the view that is in focus now and puts in into the console log.

```
Watching focus changes

webix.attachEvent("onFocusChange", function(current_view, prev_view) {
    webix.console.log("focused: " + (!view ? 'null' : view.config.id));
});
```

Keyboards Events and Hotkeys

Built-in Keyboard Events

1) 'Esc' key closes a non-modal window when it is focused.

2) 'Esc', 'space' and 'enter' keys are enabled by default for modal message boxes.

3) Editors of data component items react on the following keys:

- 'esc' - to close without saving data changes;
- 'enter' - to close with data changes saved.

For other situations keyboard can be connected to your application in several ways:

Attaching Hot keys

Component Navigation Keys

Hotkeys for navigation (arrow keys, Home, End, Page Up, Page Down) in data components like datatable and list can be enabled by navigation property:

```
{ view:"datatable",  navigation:true }
```

RELATED SAMPLE: Navigation in DataTable

Other data components need to be extended with the KeysNavigation module. Consult the dedicated article for details.

Hotkeys for Controls

For controls there exists a possibility to define a hotkey that will trigger it onClick event. The key name (e.g. 'enter' or 'space') is specified by **hotkey** property:

```
{ view:"button", click: doOnClick, hotkey: "enter" }
```

RELATED SAMPLE: Hotkeys for buttons

The *doOnClick* function will fire now either or pressing 'enter' or on clicking.

Key combinations joined by **+** or **-** sign are as well possible:

```
{ view:"button", click: doOnClick, hotkey: "enter-shift" }
```

Note that such functionality will work with simple controls like buttons and inputs, not with multiple-choice one.

Defining Custom Hotkeys

The addHotKey function is called from the UIManager object and has two mandatory parameters - key name and event handler. Key combinations joined by **+** or **-** sign are as well possible.

You can make hot keys **global**, which means that they will trigger the function regardless of the component. The one in focus will be subject to hot key actions.

```
webix.UIManager.addHotKey("Ctrl+V", function() {
    webix.console.log("Ctrl+V for any");
});
```

At the same time, you can specify any instance of a **Webix component** that should react on this or that hot key by passing the its into the function as a third parameter.

In case you want all the view instances react on the specified hot key, state the view name instead of ID:

```
//hot keys for the component with 'details' ID
webix.UIManager.addHotKey("Ctrl+Enter", function() {
    webix.console.log("Ctrl+Enter for details");
    return false;
}, $$('details')); // for "details" list only


//hot keys for all list instances on the page.
webix.UIManager.addHotKey("Ctrl+Space", function() {
    webix.console.log("Ctrl+Space is detected for list");
```

```
}, 'list'); // for all lists on the page
```

RELATED SAMPLE: Basic use of editors

Removing Hotkeys

The removeHotKey function is used to remove a hotkey. As a parameter it takes the key name:

```
//adding a hotkey
webix.UIManager.addHotKey("Ctrl+Space", function() { ... }, 'list');

//removing the hotkey
webix.UIManager.removeHotKey("Ctrl+Space");
```

Attaching Keyboard events

- **onKeyPress** *(code, ctrl flag, shift flag, native event)* - takes key codes to perform actions on pressing the specified key/keys.
- **onEditKeyPress** *(code, ctrl flag, shift flag, native event)* - has the same functionality as the event above, but the event can be applied to an opened editor.
- **onTimedKeyPress** - fires with delay and starts function execution only when a user stops typing.

In the code below the 'Enter' key opens 'details' accordionitem. Before this, you check that it is not combined with either Ctrl, Shift or Alt key:

```
'Enter' key in action

$$("books").attachEvent("onKeyPress", function(code, e) {
    if (code === 13 && !e.ctrlKey && !e.shiftKey && !e.altKey) {
        $$('details').getParentView().expand('details');
    return false;
    }
});
```

These events require that a developer should know key codes used by UI Manager. Here they are:

Key Codes

- 'enter': 13,
- 'tab': 9,
- 'esc': 27,
- 'escape': 27,
- 'up': 38,
- 'down': 40,
- 'left': 37,
- 'right': 39,
- 'pgdown': 34,

- 'pagedown': 34,
- 'pgup': 33,
- 'pageup': 33,
- 'end': 35,
- 'home': 36,
- 'delete': 46,
- 'backspace': 8,
- 'space': 32,
- 'meta': 91,
- 'win': 91,
- 'mac': 91

Global Tab Navigation

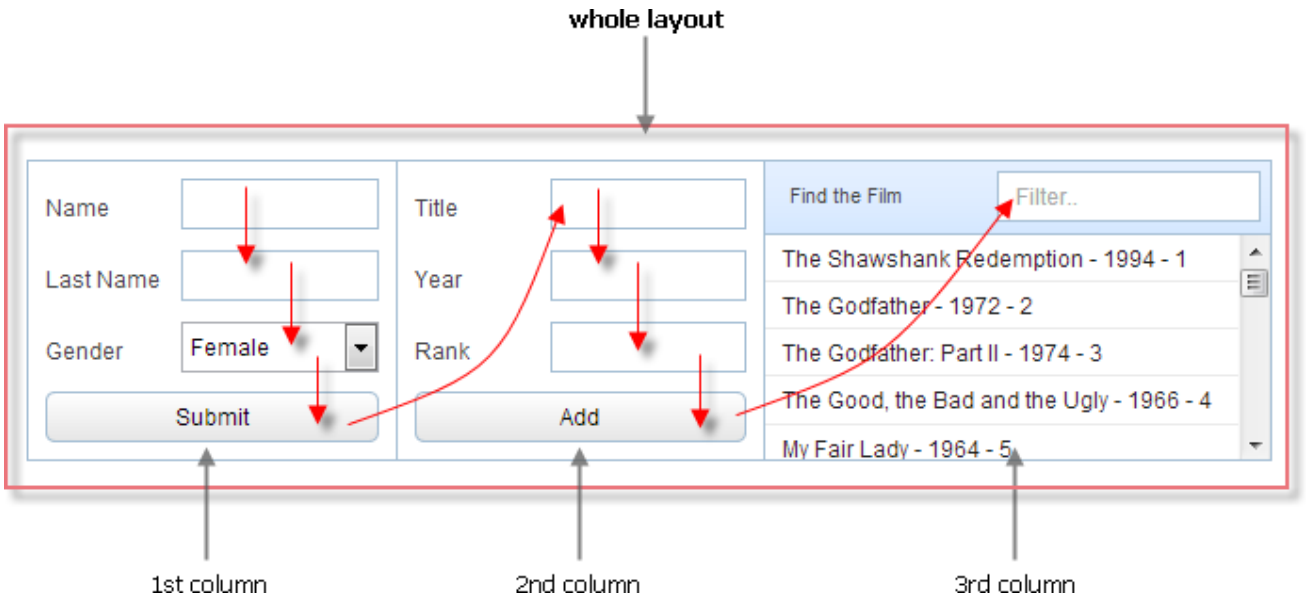
Tab/focus Order Logic

Any app on the page is put into layout, even if it isn't declared directly, so as you open the page the whole layout is focused. All the components are children of this layout. When you put cursor into any component or its item **tab navigation** can be enabled:

```
Tab Navigation

webix.UIMananger.tabControl = true;
```

From now on, you can move through your app with '*tab*' and '*shift+tab*' keys. For instance, if you have two forms in different layout columns, focus will move through all the input fields of the first form, then move to the other, then will proceed to other components and return to the parent (layout).



RELATED SAMPLE: Tab Navigation

If you want to exclude a single component from tab navigation order, make use of the **tabFocus** property, which is true by default:

```
{view:'text' tabFocus: false, ... }
```


Tab navigation within the whole app/page is possible only if all the components are intialized as part of one and the same webix.ui constructor.

What does it mean?

The 'tab-navigated' components must be children of one and the same layout since by logic focus moves from parent view to children and vice versa.

```
Tab Navigation between form and toolbar

webix.ui({
    rows:[
        {view:"form",..},
        {view:"toolbar",...}
    ]
});
```

In case of two separate webix.ui constructors you have two separate objects without the possibility to shift focus from one to the other with the help of a 'tab' key. So move a mouse pointer instead.

```
Tab navigation won't work!

webix.ui({
    view:"form", ..config..
});

webix.ui({
    view:"toolbar", ..config..
});
```

To get the neighbouring views in focus/tab order, you should apply the following methods to the currently focused view.

Methods to Shift Focus

- **getNext();** - gets the next view;
- **getPrev();** - gets the previous view;
- **getTop();** - gets the parent view to all the components.

All these methods are called from UIManager and take the necessary view as an argument.

Let's consider the picture above and see how these methods will work for a text input field (its ID is 'year').

```
var prev = webix.UIManager.getPrev($$("year")); //returns 'title' input
field object

var next = webix.UIManager.getNext($$("year")); //returns 'rank' input
field object

var top = webix.UIManager.getTop($$("year")); //returns 'layout' object
```

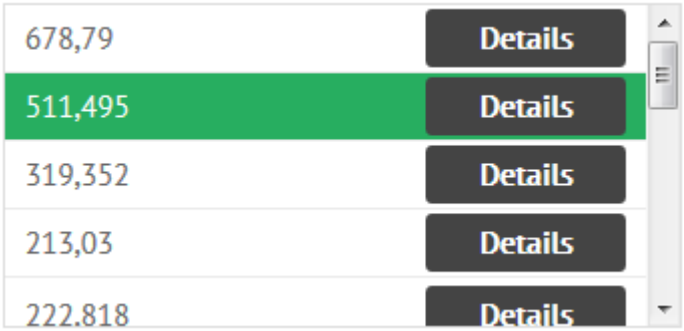
Intermediate

Adding Active Elements to Components

According to default Webix pattern, applications are made by putting pre-built meaningful components (like grid, tree, controls) into layouts. The areas with extra interactivity can be added to Webix components in two ways:

Defining Active Zones through Component templates

Clickable (or in some other way 'active') areas inside components are defined through HTML templates and CSS classes. More on data templates.



RELATED SAMPLE: List: Event Handlers

In the list above the task is solved in the following way:

- each item has a template with HTML content in addition to list item data.
- the **onClick** event of a list component is redefined for *info* CSS class. Clicking on this area ('*Details*' *button*) invokes showing a message while clicking any list item outside the button invokes default list behaviour for clicking (here item is selected);

```
CSS for active element

.info{
    width:100px;
    text-align: center;
    font-weight:bold;
    float:right;
    background-color:#444;
    color:white;
    border-radius:3px;
}
```

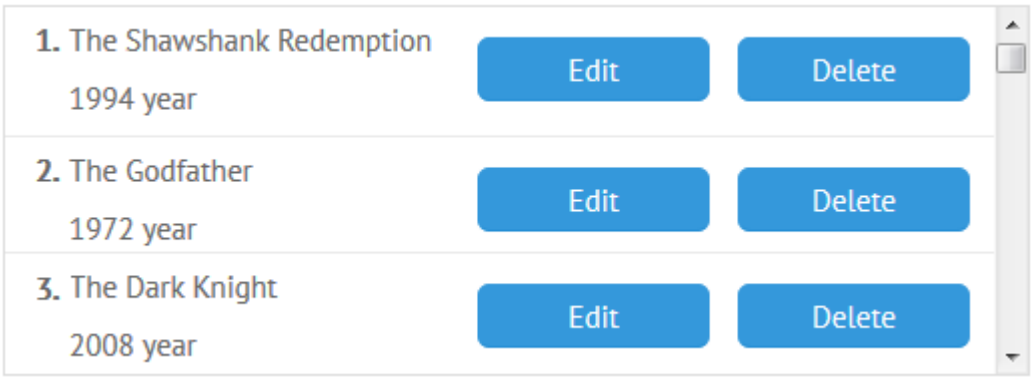
```
List config with active span element

webix.ui({
    view:"list",
```

```
    template:"#votes# <span class='info'>Details</span>",
    select:true,
    onClick:{
        info:function(e, id){
            webix.message(this.getItem(id).title);
            return false;
        }
    }
});
```

Using ActiveContent Module

ActiveContent module lets developers get rid of drawing interactive elements themselves and makes it possible to use existing Webix controls for it. In other words it allows for inserting one Webix view into another. (Normally, it goes only for layouts).



RELATED SAMPLE: List: Active Content

First of all the **ActiveContent** module should be added to the needed component by extending its default functionality:

```
webix.protoUI({
    name:"activeList" //give it some name you like
},webix.ui.list, webix.ActiveContent);
```

Now, you can use it. Note that the **name** of the newly created component is used as **view** property value:

```
webix.ui({
    view: "activeList",
    id:"mylist",
    activeContent:{
        deleteButton:{
            id:"deleteButtonId",
            view:"button",
            label:"Delete",
            width:120
        },
        markCheckbox:{
```

```
            view:"checkbox", ...
        }
    },
    template:
"#title#<div>{common.markCheckbox()}{common.deleteButton()}</div>"
});
```

"Active" template can be also defined via a function:

```
view:"activelist",
template: function (obj, common) {
   return obj.title+"<div>"+common.deleteButton(obj, common)+"</div>";
}
```

- the **activeContent** property contains an array of key-value pairs where:
  - *key* is a used-defined **name** of an active area (here *deleteButton*, *editButton*);
  - *value* is its **configuration**.
- config of the active element is stored in the **common** property of the housing component and can be derived by its *name*;
- template for each list item includes these configurations as {common.*name*};

If an active content element can have a **variable value** (like text field, or checkbox) it can be passed to it within data item properties:

```
Value for 'markCheckbox' active element

view:"activeList",
data:[
    { id:5, title:"My Fair Lady", year:1964, rank:5, markCheckbox:1}
]
```

Click and other **events** are attached to the buttons according to Webix Event handling pattern:

```
view:"button", click:function(){...}
//or
button.attachEvent("onItemClick", function(){...});
```

Getting master of an active content element

- using locate function:
  - in any mouse event handler of the active element (e.g. *onItemClick* event, or *click* property).

```
$$('deleteButtonId').attachEvent("onItemClick", function(id, e){
    var item_id = $$('mylist').locate(e);
    //returns id of a list item
});
```

- through the active element configuration:
  - if a handler is attached to non-mouse event (all component-specific Webix events);
  - if an active element is redrawn on a mouse event (like toggle, checkbox).

```
on:{
    'onChange':function(newv, oldv){
        var item_id = this.config.$masterId;
    }
}
```

Datatable and treetable specificity

With datatable and treetable you can get a row object by **$master** property that includes:

- **row** - row id;
- **rind** - row index;
- **column** - columns id;
- **cind** - column index.

```
on:{
    'onChange':function(newv, oldv){
        var item_id = this.config.$masterId.row;
    }
}
```

Related articles

- Adding a custom element to the datatable header

Web and Internet

- Webix Storage Interface
- Saving Application State

Advanced

Webix Storage Interface

Webix library offers its own API for working with application cache and browser local storages - permanent cache, per session cache and cookies.

Accordingly, **webix.storage** module contains three objects:

- webix.storage.local - works with window localStorage object;
- webix.storage.session - works with window sessionStorage object;
- webix.storage.cookie - works with document cookie object.

The objects share three methods:

- **put**(*name, data, [domain, expires]*) - the last two params concern only cookie storage. The method allows putting data to browser cache under the stated name. Data is passed as JSON object;
- **get**(*name*) - the method derived stored data from local cache for further usage. Data comes in JSON format;
- **remove**(*name, [domain]*) - the last param concern only cookie storage. The method removes data saved under the stated name from local cache.

Working with Local Storage

Local storage can be used for storing any application data, e.g. current state of a component derived with getState() method.

To do this, you **put()** state object into Webix local storage under a **custom name**. Data you save should be in **JSON** format(which is exactly what getState() returns).

```
function save_state(){
    webix.storage.local.put("state", grid.getState());
}
```

If data you'd like to save is plain Javascript object, apply Webix serialize() method to it.

To retrieve data from local storage, use **get()** method pointing to the name under which you've save this or that data.

```
function restore_state() {
    var state = webix.storage.local.get("state");
    if (state)
        grid.setState(state);
}
```

RELATED SAMPLE: DataTable state object::Session

To save and track state of a multiview-based app, follow other instructions.

Using Local Storage for Offline Support of ServerSide Applications

Local storage interface can be used for **storing serverside data** in case of disconnect, which ensures that local changes won't be lost even after page refresh.

Local storage functionality is set by adding a prefix of desired mode - **cache** or **offline** to you load and save scripts.

```
webix.ui({
    view:"datatable",
    ..config..
    url:"offline->load.php",
    save:"offline->save.php"
});
```

In either of these modes, serverside data is loaded to a component, and is additionally cached to your machine(**webix.local.put()** function is executed in background).

What happens next depends on the chosen mode:

Offline mode:

- The component will always try to get up-to-date data data from server. If server is unavailable, data is reloaded from cache (*webix.local.get()* function is executed behind the scene). That's why you can refresh page as many times as you wish;
- Data changes you make while online are pushed to server and to cache. So cache will always contain the latest copy of serverside data;
- Data changes you make while offline, are pushed to cache only and should be sent to server manually.

Cache mode:

- Once data has been loaded to a component it is put to cache (local storage) and later on the component will work only with cached data without trying to get to server;
- Data changes you make while either offline or online are pushed to cache and should be sent to server manually.

Local mode:

- The component works with in-browser local storage only. Data is loaded from there (is any), and all CRUD operations are saved to local storage;
- You can access localStorage objects to get data from and put data to.

Read also main Offline Support article.

Working with cached data

Although Webix library offers a ready-to-use solution, you can write your own proxy object to enable offline support and set it as the component's **url** property:

```
webix.ui({
    view:"datatable",
    ..config..
    url:{
        $proxy: true,
        load: function(view) {//save data after editing
             view.attachEvent("onAfterEditStop", function(){
                webix.storage.local.put("app_data", this.serialize());
              });
             view.load("data/data.json", {
                success: function(text){//fresh data, save in cache
                     webix.storage.local.put("app_data", text)
                },
                error: function(){ //probably in offline mode
                     view.parse(webix.storage.local.get("app_data"));
                }
            });
        }
     }
});
```

- **onAfterEditStop** event is used to catch data changes and put edited data to cache;
- cached data is inline data, that's why it is **parsed** into a component, not loaded.

Saving Application State

There exist different ways to save the state of the application and its components after page refresh.

1) UIManager offers a possibility to save and restore the state of the **whole Webix application**. Get details.

2) Webix DataState module makes it possible to save state of any **separate component**. Get details.

3) Webix Histoty API makes it possible to track the **latest opened tab** in multiview applications. Get details.



The first and the second methods are available in **Webix Pro** edition only.

Available only in **PRO Edition**

Saving and Restoring Application State

Webix UIManager allows saving current visual scheme of the component, its 'state' with the following methods:

- getState() - to capture current state of a Webix application;
- setState() - to set the saved state for the application.

The data for the state itself is stored in the webix.storage.local object (non-pro), which is your browser cache. Simply put, data is saved to your local machine.

The state of any view included into the application state object contains the following information:

```
{
    id:"main",
    width:500,
    height:400,
    gravity:1
}
```

Some views featuring tabs, panels or cells in their configuration contain additional properties:

```
{
    collapsed:"tab1", //currently collapsed panel
    activeCell:"tab2" //currently active tab
}
```

Getting and Saving State

The **getState** method is called from the UIManager and gets the app's general structure (namely ID, width, height, gravity). It takes two arguments:

- the **ID of the component** which state you'd like to save (as a rule, the ID of the top-most component, layout);
- boolean **true or false** for component children. If true, state of nested components is saved alongside with layout state.

To save the data and make it usable later on, your should **put** it into the local storage, defining the key for it:

```
webix.attachEvent("unload", function(){
    webix.storage.local.put("stateApp", webix.UIManager.getState("main",
true));
});
```

Restoring State

To use the method, you should firstly **get** the stored data from the local storage (using the key you defined for it at the previous step):

```
var stateApp = webix.storage.local.get("stateApp");
```

And then call **setState** from UIManager to restore the state of the application.

```
if(stateApp)
   webix.UIManager.setState(stateApp);
```

RELATED SAMPLE: Application state::Session

Available only in **PRO Edition**

Saving and Restoring Component State

UIManager helps save and restore only the outer view parameters while inner ones like selection and scrolling direction are saved within the DataState module.

Webix components that use this module feature the saving and restoring functionaluty as well:

- datatable;
- treetable;
- datatree.

Saving Datatable State

Webix datatable provides a possibility to store/restore the state of the tree to cookie, local or session storage.

The state object will contain information about current datatable state:

- properties (keys) of each data item;
- column sizes;
- IDs of currently selected items (each ID is an object with *row, column* and *id* properties);
- scroll position;
- values of built-in filters;
- IDs of currently hidden data items.

```
{
    ids: ["id", "value", "chapter"],
    size:[50, 250, 200],
    select:[{"row":217,"column":"title","id":"217_title"}],
    scroll:{ x:, y:},
    filter:{"year":"1924"},
    hidden:["votes"]
}
```

- To save the current tree state to the local storage you should call the getState method as in:

```
var state = grid.getState();
webix.storage.local.put("state", state);
```

- To restore the saved state you should call the setState method:

```
var state = webix.storage.local.get("state");
if (state)
```

```
    grid.setState(state);
```

RELATED SAMPLE: DataTable state object::Session

Saving Treetable State

Webix treetable provides a possibility to store/restore the state of the tree to cookie, local or session storage.

The state object will contain information about current treetable state:

- IDs of currently hidden data items;
- properties (keys) of each data item;
- open treetable nodes;
- scroll position;
- IDs of currently selected items (each ID is an object with *row, column* and *value* properties);
- column sizes.

```
{
    hidden:[],
    ids: ["id", "value", "chapter"],
    open:["1", "1.2"],
    scroll:{ x:, y:},
    select:[{"row":"1.2","column":"value","id":"1.2"}],
    size:[50, 250, 200]
}
```

- To save the current tree state to the local storage you should call the getState method as in:

```
var state = grid.getState();
webix.storage.local.put("treetable_state", state);
```

- To restore the saved state you should call the setState method:

```
var state = webix.storage.local.get("treetable_state");
if (state)
    grid.setState(state);
```

RELATED SAMPLE: TreeTable state object

Saving Tree State

Webix tree provides a possibility to store/restore the state of the tree to cookie, local or seccion storage.

The tree state object will contain information about opened and selected nodes:

```
{
    open:['1', '2', 'root'],
    select: ['1.3']
}
```

where '1', '2', 'root' and '1.3' are the ids of the related nodes.

- To save the current tree state to the local storage you should call the getState method as in:

```
Saving the tree state to the local storage

var state = $$("tree").getState();
webix.storage.local.put("state", state);
```

- To restore the saved state you should call the setState method:

```
Restoring the tree state from the local storage

var state = webix.storage.local.get("state");
if (state)
    $$("tree").setState(state);
```

RELATED SAMPLE: Tree State Object::Session

Intermediate

Tracking the Latest Opened Tab

Here we discuss the state of an application created with multiview layout.

With multiview application design you can create as many cells as you wish and switch between them with the help of dedicated buttons, namely tabbar and segmented, or resort to tabview functionality.

Tab values concide with IDs of the views you place into multiview cells.

Initilially, either the first tab is visible, or the one you've set to be "selected" during intialization. Normally, when you refresh the page, the application is redrawn according to this initialization pattern.

```
webix.ui({
    rows:[
      {view:"segmented", id:'tabbar', selected:"listView", multiview:true,
options:[
            { value: 'List',  id:'listView'},
            { value: 'Form',  id:'formView'},
            { value: 'About', id:'aboutView'}]
      },
      {cells:[
            {id:"listView",  view:"list"},
            {id:"formView",  view:"form"},
            {id:"aboutView", view:"template"}]
      }
    ]
});
```

Nevertheless, the library offers **history API** to store the current application state, or simply put, to memorize the latest opened tab.

History track method allows remembering the currently opened tab and showing it after page loading. It takes two parameters:

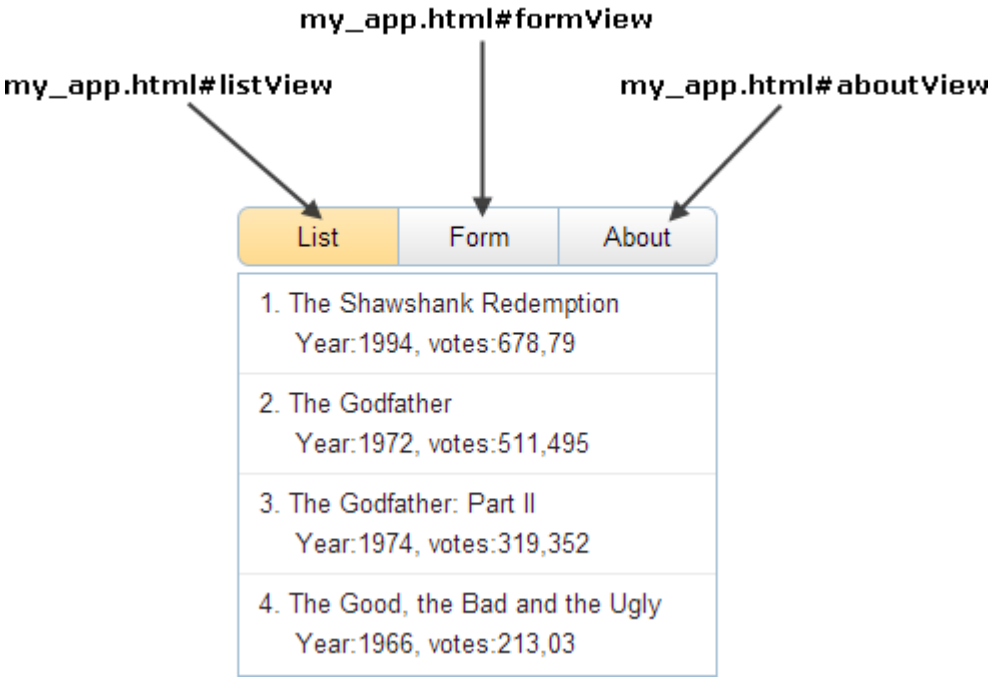
- **id** - the **ID** of the segmented, tabbar and tabview that is used for switching between miltiview views.
- **url** - **ID** of a multiview **cell** that forms the hashbang url that is loaded after page refresh. By default it's the ID of the currently opened cell.

```
webix.history.track('tabbar');
```

Each time you click the tab to switch for another cell, a **hashbang URL** is generated on the base of **cell ID** (under *http://current_url.html#!/cellID* pattern) and memorized. You'll see it in the url bar of your browser.

If for the **track()** function you define the url parameter as cell ID, this cell will always be shown after refresh. If you don't specify this parameter, the **last opened cell** will be displayed.

Compare ID-s from the snippet above and hashbang URL-s for each of the tabs on the picture.



RELATED SAMPLE: History API with Multiview

Now, each time, you refresh the page, the url wil be form according to the info stored in the state object and th altest opened tab wil be shown.

Pushing New State to History

In addition, there exists a possiblity to programmatically add a new state to history with the dedicated **push()** method. Its argumnts are:

- **view** - ID of related switching control;
- **url** - ID of the cell that will form in-page url for new state (if you pass *cellID*, the url will be formed as *#!/cellID*);
- **value** - any related data in the form of a JSON object. If will be stored in the **state** object.

The method is applied to **history** object:

```
webix.history.push("tabbar", "detailView" "details");
```

..where details are

```
var details = {
    view:"template",
    template:"Some very long text"
};
```

However, this will cause the URL bar to display *http://current_url.html#!/detailView*, but won't cause the browser to look for *detailView* anchor or even check whether it exists. It just adds this info to window history object.

Extending components

- Component Copying and Extending
- Creating a Custom Component

Advanced

Component Copying and Extending

Components in webix UI library are objects that can be cloned and copied as well as inherit properties from each other.

Let's take for instance the toolbar component and play around with it. It has three properties - width, height and cols object with three buttons.

```
var barA = {view:"toolbar", width:500, paddingY:2,
            cols:[
                { view:"button", value:"Load" },
                { view:"button", value:"Save" },
                { view:"button", value:"Info" }]
        };
```

Copying

Shallow Copying

Shallow copying is implemented via the **clone();** method. A clone of an object is an empty object with a prototype reference to the original one.It doesn't go deep during copying and clones the structure of the source component. All the changes you make in the source object are reflected
in the target object.

```
var barB = webix.clone(barA);

barA.width = 400;

barA.width --> 400
barB.width --> 400
```

Deep Copying

During deep copying a physical copy of a parent object is being made. The copy is fully independent, so, if later on you make changes in the source object, the properties of the target object remain unchanged. However, the method doesn't work for structures with inner loops.

```
var barB = copy(barA);

barA.width = 400;

barA.width --> 400
barB.width --> 500 // initial value from the source object
```

The method is useful when you'd like to create two independant instances of one anfd the same object on the page.

This is how a new pager is created on the base of the *pagerA* one.

```
$$('pagerA').clone({
        container:"pagingB_here", //own parameters
        size:50,
        group:5
    });
 });
```

RELATED SAMPLE: Two-area pager

Inheritance

New components can be created on the base of already existing ones. There is an **extend();** function that merges the properties of two objects into a new one. If any of the shared properties gets changed in the target object, it is changed as well in the source one. Native properties of the two objects are independent.

extend(object **target**, object **source**, [boolean **overwrite**])

- **Target** is an object that is being extended. It receives new properties.
- **Source** is a component that acts as a source for the target component;
- **Overwrite** is *false* by default, yet it can be set to *true*. In this case, the properties that coincide are overwritten to the values of the source object (sometimes you can get a full copy).

```
barB = {view:"toolbar", cols: [{view:"button", value:"Accept"}]};

webix.extend(barB, barA);

//here we get:

barB = {view:"toolbar", width:500, paddingY:2, cols: [{view:"button",
value:"Accept"}]};
//width and padding are inherited properties
```

```
Full (object-based) copy with the help of extend();


webix.extend(barB, barA, true);

//here we get a copy with all the properties overwritten:
```

```
var barB = {view:"toolbar", width:500, paddingY:2,
            cols:[
                { view:"button", value:"Load" },
                { view:"button", value:"Save" },
                { view:"button", value:"Info" }]
        };
```

Extended Functionality

Late Binding

Here we try to find how to extend component functionality with properties that aren't typical of it and not stated in its API. For instance, we want a component to be movable. It can be enabled with the **extend();** method while the source object here is **webix.Movable** module.

```
webix.ui({
    view:"toolbar",
    ...
})
webix.extend($$("my_toolbar"),webix.Movable);
```

Now our toolbar can can be moved across the screen.

It is a so-called **late binding**. In fact the target object here is supplied with some properties from the source object while preserving its native properties. Moreover, iherited methods won't overwrite the existed methods with the same name.

Early Binding

If you want to put apart inherited and native methods of the same name, you should refer to **early binding**, where prototypes of two objects are merged.

```
Movable Toolbar

mytoolbar = webix.proto({webix.Movable},webix.ui.toolbar);
var obj = new mytoolbar({
                view:"toolbar",... );
// outputs a movable toolbar
```

To extend component's existing functionality you can as well use **protoUI();** method that creates the new view based on some of the existing ones. This is how it looks in case you want to make your list editable.

```
Editable List

webix.protoUI({name:"mylist"},
            webix.EditAbility,
            webix.ui.list);

webix.ui({view:"editlist",
        ..
```

```
        });
```

RELATED SAMPLE: List: Editing

Advanced

Creating a Custom Component

You can create a custom component from any of the existing ones, assigning properties, methods and events to it.

The library allows extending functionality of any component by adding building modules to it (however, most components feature all the necessary functionality by default) as well as creating a totally custom component that inherits from the basic view.

Simply put, the technique uses the **protoUI();** command to construct the new component with it own name, properties and functions. All the new components must be based on existing ones.

```
webix.protoUI({
   name:"newComponent", // the name of a new component
   $init:function(config){
        thid.load(data.xml)
        //functions executed on component initialization
   },
   defaults:
        {width:200}
   on:{'onItemClick', function(){}}, //attached events
   custom_func:function(){..function description..}, //any custom function
   $getSize: function(){..},
   $setSize:function(){..},
}, webix.Movable, // extended functionality
webix.ui.list); //the name of the base component
```

As a result, we get a **custom component** that

- inherits from ui.list;
- on its initialization loads data from the 'data.xml' file
- can be moved over the screen (thanks to the webix.Movable module);
- has a default width of 200px;
- has a *custom_func* function;
- is sized with the help of $getSize and $setSize methods (will be described later).

Later on, the component is inserted into the app the same way you do with library components:

```
webix.ui({
    view:"newComponent",
    ..config options..

})
```

The must-have elements for such constructor are new component's name and the component it inherits from.

Additional functionality can be included to your choice.

```
Editable List

webix.protoUI({
    name:"editlist",
        },
webix.EditAbility, webix.ui.list);
```

The **protoUI constructor** eliminates the need of repeated usage of the **extend();** method each time you want to initialize a new instance of your custom component. Read more about extended component functionality.

Main Methods and Properties

Before we describe how the mentioned technique works, it will be helpful to summarize the main methods and properties you may use while developing a component.

Properties:

- **$view** - the HTML ( ) component container.
- **$width** - the width of the component container.
- **$height** - the height of the component container.
- **$ready** - an array of commands. You may use this property to call some function as soon as the **$init** method and all the setters are executed. The functions are added to the array through the command **push** (call this command in the method **$init**):

```
this.$ready.push(...);
```

Methods:

- **defaults** - defines default settings of the component.
- **$init** - for the most part defines component initialization. Called at the very beginning.

In this method you add HTML elements, specify event handlers. As a parameter the method takes the object 'config' that contains all the properties which a user set in the constructor. Inside the method you can refer to the HTML object of the component through **this.$view**.
Note, the method doesn't take properties set in the method '**defaults**'. They are available just after initialization is finished completely (i.e. **$init** and all the property setters are executed).

- **$getSize** - defines the size of the whole component. You should use this method to set the size of the component that differs from the size allocated by the parent view. Called when inner code inquires about height or width which should be set for the component.

The method returns the array *[gravW;width;gravH;height]* (width gravity, fixed width, height gravity, fixed

height).

The values of the array conform to the following rules:

- Width(height) can take the next values:
  - any natural number;

    This value is used as width(height) of the component.
  - -1;

    This value tells the script to ignore the parameter.
- GravW(gravH) can take any natural value. The value '1' tells the script to ignore the parameter.
- When width(height) is set to any natural value, gravW(gravH) is ignored. When width(height) is set to -1, the script asks for gravW(gravH) value.

The possible scenarios:

Sc.1 You want the parent container to set the size automatically (autosizing).

Solution: don't redefine the function or return [1;-1;1;-1]

Sc.2 You want to set relative width(height).

Solution:

a)for relative width return [yourValue;-1;1;-1]
b)for relative height return [1;-1;yourValue;-1]

Sc.3 You want to set fixed width(height).

Solution:

a)for fixed width return [1;yourValue;1;-1]
b)for fixed height return [1;-1;1;yourValue]

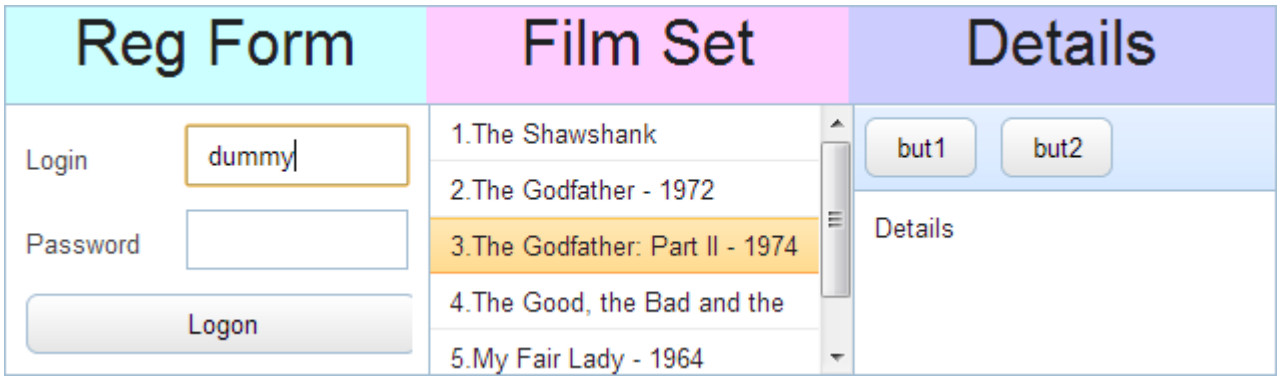
- **$setSize** - defines behaviour and sizes of the specific elements of the component. You should use this method if your component has a complex structure. Called as soon as the size of the whole component is set (after **$getSize** method).

Let's consider some existing component, for example, window. It has a header and a body. Assume, you hide the header. The body of the window will keep the previous size but instead of the header you'll see empty space. To prevent this, you should specify a logic defining behaviour of the elements in such the situation. The appropriate logic must be specified in the method **$setSize**.

A Custom Component

Fully custom components inherit from view, as a rule, which is an empty container for library components. Basic modules are also included into your component to support app-user interaction.

Let's see how can we create a multi-colored header for a three-column app.




```
webix.protoUI({
    name:custom",
    ..configuration..},
    webix.MouseEvents, webix.EventSystem, webix.ui.view);
```

- **webix.MouseEvent** - a module that makes your component support mouse events such as '*onItemClick*';
- **webix.EventSystem** - to support event handling functionality, such as *attachEvent()* method.

Adding Functionality to the Component:

1 . For a component you can set **CSS classes** with the help of the *className();* method defined for *$view* property that points to an HTML container of the component. It is included into the **$init** function alongside with the **innerHTML** method to set containers for component items.

```
$init:function(config){
    this.$view.className = "my_control";
    this.$view.innerHTML = "<div class='item1'></div><div
class='item2'></div>
    <div class='item3'></div>";
}
```

2 . Define default properties:

```
defaults:{ controlHeight:50 }
```

3 . Create component items by defining item setter functions:

```
item1_setter:function(value){
    if(value)
        this.$view.childNodes[].innerHTML = value;
    return value;
}
```

These functions are used in the component code to set inner HTML for component items:

```
webix.ui({
    view:"custom",
    item1:"Reg Form",
    item2:"Filmset",
    item3:"Details"
})
```

4 . Size the component. Here two methods are used:

- **$getSize** - returns the current size of the component in the form of an array (min width, max width, min height, max height, gravity). The method is optional here;
  - Use **-1** if you want to ignore any of these parameters.
- **$setSize(x, y)** - takes width and height as arguments and sets component dimensions.

Both of them are used for inner logic. To size and resize existing components refer to the dedicated article.

```
$getSize:function(){
    return
[1,100000,this.config.controlHeight,this.config.controlHeight,1];
},

$setSize:function(x,y){
    if (webix.ui.view.prototype.$setSize.call(this,x,y)){
        var itemWidth = Math.round(this.$width/3);
        for(var i=;i<3;i++){
            this.$view.childNodes[i].style.width = (i==2?this.$width-
            (itemWidth*2):itemWidth) +"px";
            this.$view.childNodes[i].style.height = this.$height+"px";
        }
    }
}
```

5 . Attach events to the component

```
on: {'onMouseOver': function(){...);}}
```

As a result, we've got a multi-colored header template within a three-columnn design and 'elastic' sizing, which means that the app can adjust its size to the current window size.

RELATED SAMPLE: Custom Component

Third party integration

- JQuery Integration
- Angular JS Integration
- Backbone Integration
- Integration with DHTMLX Components
- DIY - Custom Integration Pattern

Also, be sure to check UI Extensions

Intermediate

JQuery Integration

webix library works in tandem with JQuery libraries as well, which is an obvious bonus for those who are used to working with JQuery.

JQuery is included by a link specified in your document's head section

```
<script src="../path/library_name.js" type="text/javascript"
charset="utf-8"></script>
```

Working with JQuery, you can create all the components from the webix UI library. The syntax differs a little bit; however, it grants you new possibilities.

Component initialization:

```
$(#div ID/ .CSS selector').webix_component({config});
```

Let it be list:

```
<div id="listA"></div>
  ...
$("#listA").webix_list({
    width:320, height:600,
    template:"#votes# #rank# #title#",
    data:big_film_set
});
```

RELATED SAMPLE: JQuery List

Working with Components

You can get to any component using the same scheme and apply functions to it.

```
Counting data items

$('#div ID/ .CSS selector').webix_component().count()
```

Advantages of JQuery Integration

1 . An HTML container for any component can be specified by its **ID** or **CSS class** used with it. With the webix library means HTML container can be defined by its ID only:

```
<div class='calendar_here'></div>
<div id='list_here'></div>
...
$("#listA").webix_calendar({config}); // div ID
$(".calendar_here").webix_calendar(); //CSS selector
```

2 . If you use the same **CSS class for several nodes**, you can create **multiple instances** of the same view by pointing to this CSS class during component initialization.

```
Two calendars will be displayed

<div class='calendar_here'></div>
<div class='calendar_here'></div>
...
$(".calendar_here").webix_calendar();
```

RELATED SAMPLE: JQuery Calendar

3 . An **HTML table** can be the **datasource** for the component inited in the same container. Each column in the table is reflected as a template item for the component:

```
Table row with three columns

<tr>
    <td>1</td>
    <td>Record #102</td>
    <td>5406</td>
</tr>
```

Columns in template are called ##data## and are numbered starting from 0.

```
List template to grab the data from the table

{
    template:"<div class='mark'>#data2# </div> #data0#. #data1#",
    ...
}
```

RELATED SAMPLE: Jquery List from HTML table

Angular JS Integration

Webix-Angular App Structure

To use **Angular JS framework** for Webix component you should:

- include **Angular** and **Webix** scripts into the document. Note that order does matter here - Angular JS script must come first;
- set **ngApp directive** to the document root to bootstrap an application. For convenience means, it's recommended to use **"webixApp"** as **module name**.
- **create a new module** passing module name from the previous step and "*webix*" required parameter to connect it to Webix library.

```
<!doctype html> <!--setting directive-->
<html lang="en" ng-app="webixApp">
<head>
  <meta charset="utf-8">
  <title>Webix-Angular App</title>
  <script src="js/angular.min.js"></script>
  <script type="text/javascript" src="../../codebase/webix.js"></script>
  <link rel="stylesheet" type="text/css" href="../../codebase/webix.css">

  <script type="text/javascript">
     var app = angular.module('webixApp', [ "webix" ]); //creating module
     ..//app's js code (controllers)
</script>
</head>
<body>
<!-- app's html -->
</body>
</html>
```

However, since app logic is typically complex, it's a good practice to store controllers separately:

```
<script type="text/javascript" src="controllers.js">
```

Initing Webix Components

Bound to Angular JS, Webix offers a special **webix-ui** directive that bootstraps an application. Here two methods are possible:

- Initing from HTML markup - the directive is used without an argument, the application is inited via HTML:

```
<div webix-ui type="space"> <!-- app html --> </div>
```

- Initing from a JS configuration object - the directive is used with the app's config object as an argument, no additional markup is used. Config object is stored in the app's controller:

```
<div webix-ui="config" webix-ready="doSome(root)" style="width:450px;
height:300px;">
```

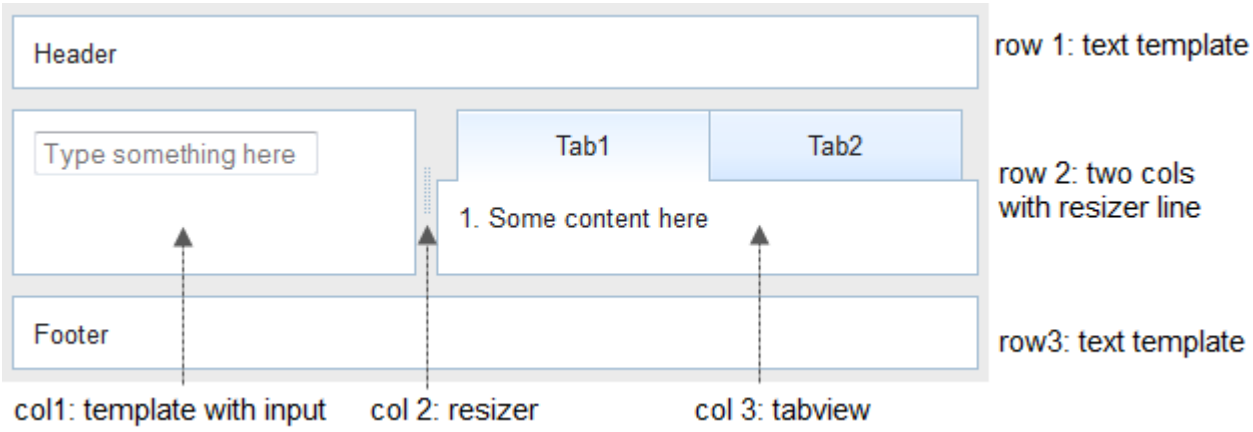
Two patterns are equal in functionality yet differ in the way this functionality is implemented.

Initing from from HTML markup

This method resembles Webix HTML Initialization technique but has its own peculiarities:

- Webix application lies in a div block with **webix-ui** directive:
  - **view** attribute of such block specifies Webix component you want to init;
  - if you don't specify the view attribute - a layout row will be created;
- all div blocks within "webix-ui block" are Webix views(components):
  - **view** attribute of a div specifies Webix component you want to init;
  - a div block without the view attribute inits Webix template;
  - other attributes of div blocks are component **properties**;
- As with standard Angular JS, you can easily insert Angular directives into tags including inputs and buttons (like in standard HTML).

Layout with Tabbar and Multiview




```
<body>
  <div webix-ui type="space"> <!--layout rows with type "space" are
created-->
    <div height="35">Header { {app} }</div> <!--Webix template is inited -
->
    <div view="cols" type="wide" margin="10"> <!--Webix layout cols are
inited-->
```

```
        <div width="200">
            <input type="text" placeholder="Type here" ng-model="app">
        </div>
        <div view="resizer"></div>
        <div view="tabview">
            <div header="Tab1">
                1. Some content here
            </div>
            <div header="Tab2">
                2. Other content here
            </div>
        </div>
    </div>
    <div height="35">Footer</div>
  </div>
</body>
```

RELATED SAMPLE: Webix - Angular : Layouts

The input and header template are bound together by **ng-model** directive.

You can check how these views are inited in Webix: layout, resizer, tabview, template.

Datatable Component

Webix datatable is a complex component for working with large amounts of data. The table below features **four columns**, **row selection** and **auto height**. Each column has its own **ID**, **sorting method** and **header** (the latter is defined by innerHTML).

```
<div style="width:750px;">
 <div webix-ui view="datatable" webix-data="records" autoheight="true"
select="row">
   <div view="column" id="rating" sort="int" css="rating">Rating</div>
   <div view="column" id="year"   sort="int">Year</div>
   <div view="column" id="votes"  sort="int">Votes</div>
   <div view="column" id="title"  sort="string"
fillspace="true">Title</div>
 </div>
</div>
```

RELATED SAMPLE: Webix - Angular : Grid

Note that any HTML outside "webix-ui" block is parsed as standard markup while a div with **webix-ui** attribute constructs Webix app.

Data loading issues are explained separately.

Chart Component

Webix chart is a handy data visualization tool. The chart below is of a **bar** type, shows sale rates and comes

without borders.

```
<div style="width:500px; height:200px;">
  <div webix-ui view="chart" id="mychart" webix-data="lines" type="bar"
value="#sales#" borderless="true"></div>
</div>
```

RELATED SAMPLE: Webix - Angular : Chart

Note that any HTML outside "webix-ui" block is parsed as standard markup while a div with **webix-ui** attribute starts Webix app.

Data loading issues are explained separately.

Rendering Arrays

Webix to Angular integration allows for rendering both arrays of components as well as arrays in component configuration with Angulat **ng-repeat** directive.

Rendering an array of the same Webix components

Webix-ui directive can be combined with Angular **ng-repeat** to render an arry of views

```
<div webix-ui>
    <div ng-repeat="chart in charts" view="chart">
</div>
```

where *chart* is a scope variable that includes an array of component configurations:

```
$scope.charts = [
    { data:chart_data, series:[...] },
    { data:chart_data, series:[...] }
  ];
```

Rendering arrays within component configuration

Arrays in component config are denoted by **stack** property. Inside array notation you can use **ng-repeat** to iterate through it and specify the information to display:

```
<config stack='series'>
    <config ng-repeat="serie in chart.series"
value="#.template#"></config>
</config>
```

where *series* is a property of scope *chart* variable that contains an object with chart series config:

```
{
    data:chart_data,
    series:[
        { template:"sales3", color:"#ff8" },
```

```
        { template:"sales", color:"#f8f" }
    ]
}
```

RELATED SAMPLE: Webix - Angular : Using ng-repeat

Rendering Complex Layouts

Webix-Angular integration allows for complex application design with different inter-related Webix components in it.

For more details, study the dedicated sample RELATED SAMPLE: Webix - Angular : Complex Init

Further Reading

- Angular Controllers and Data Loading **Webix-data** directive;
- Event Handling during Webix-Angular Integration **Webix-event** directive;
- Changing Visibility of Components in the App **Webix-show** directive;

Initing from Config Object

Initialization via config object helps **get rid of markup** and hence, markup-specific directives, and move the entire application code to **Angular controller**.

The only markup line you need is:

```
<body ng-controller="webixTestController">
    <div webix-ui="config" webix-ready="doSome(root)"></div>
</body>
```

- This method is closer to Webix initialization pattern. **Config** object is JSON object you would pass into **webix.ui()** constructor if you were working with Webix alone;
- Event handlers are attached with the help of **webix-ready** directive that executes a controller function taking **config root** as parameter. Described later.

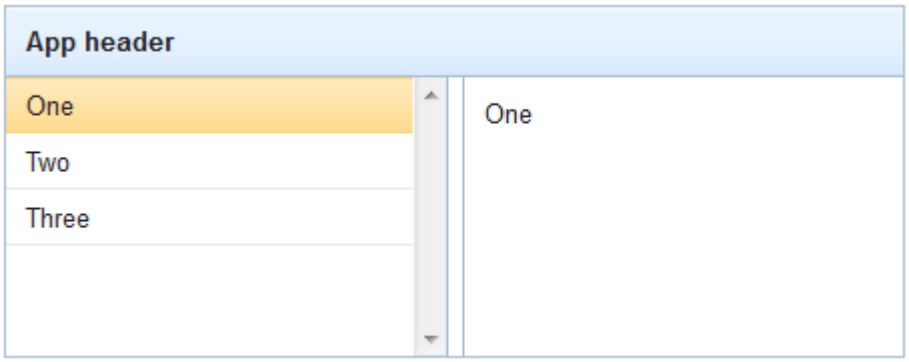
```
Controller Code

var app = angular.module('webixApp', [ "webix" ]);

app.controller("webixTestController", function($scope){
    var header = { type:"header", template:"App header" };
    var left = { view:"list", id:"a1", select:true, data:["One", "Two",
"Three"] };
    var right = { template:"Right area", id:"a2" };

    //config object
    $scope.config = {
```

```
        isolate:true, rows:[ //two rows
            header,
            { cols:[
                left,  //list
                { view:"resizer" }, //resizer line
                right //template
            ]}
        ]
    };
};
```



RELATED SAMPLE: Webix - Angular : Initing from Config

- Components used in the sample are list, template, layout, resizer line.
- **Top parent view** of the config object (here: two-row layout) should have an **isolate** property to **avoid ID mess** in a situation when there're same IDs in another config object on the page. Similar functionality is offered by Webix IdSpace mixin.

Attaching Events with webix-ready Directive

Webix-ready directive executes a controller function with a **config root** as an argument and makes it possible to attach event handlers for all components in current configuration. If you init from HTML markup, use webix-event directive for these needs.

```
<body ng-controller="webixTestController">
    <div webix-ui="config" webix-ready="doSome(root)"></div>
</body>
```

Root is a **top parent view** of your application config. Here root is a *two-row layout*.

Root has an **isolate** property, which means that the IDs of its child views (header, list, template) can be not unique within the page (there can be same IDs in another config object). But, when attaching event handlers, you should refer to components via their root.

```
Controller Code

app.controller("webixTestController", function($scope){
    $scope.doSome = function(root){
    var list = root.$$("a1"); //referring to list via root object
```

```
    var template = root.$$("a2"); //referring to template via root object

    list.attachEvent("onAfterSelect", function(id){
      template.setHTML(this.getItem(id).value);
    });

    list.select(list.getFirstId());
  };
};
```

Inside the fucntion invoked by **webix-ready** directive, Webix-Angular integrated app complies to standard Webix event handling pattern.

Angular Controllers and Data Loading

Since Angular controllers are heart of any Angular-based application, Webix-Angular integration preserves controller functionality.

As in Angular, controller directives are included into the app as an attribute of any tag.

```
<html ng-app="webixApp"> <!--Angular App inited-->
 <body ng-controller="webixTestController"> <!--Angular controller-->
  <div webix-ui view="datatable" webix-data="records" select="row"> <!--
Webix App-->
    ...<!--datatable config-->
  </div>
 </body>
</html>
```

Loading data via Controller

Scope variables are accessed by standard pattern except for data loading. To populate the datatable with data from controller **$scope**, use **webix-data** directive. It is used in a div that initiates this datatable.

The code below gets *$scope.records* and parses them into the datatable:

```
<div webix-ui view="datatable" webix-data="records" > ...</div>
```

Filtering and sorting options can be used as well:

```
<input ng-model="query" ... />
<select ng-model = "orderProp">...</select>

<div webix-ui view="datatable" webix-
data="records|filter:query|orderBy:orderProp">
...
</div>
```

Controller code

```
var app = angular.module('webixApp', [ "webix" ]);

app.controller("webixTestController", function($scope){
  $scope.records = [
    { id:1, title:"The Shawshank Redemption", year:1994, votes:678790,
rating:9.2, rank:1},
    { id:2, title:"The Godfather", year:1972, votes:511495, rating:9.2,
rank:2},
    ...
  ];
```

```
  $scope.addRecord = function(){
    $scope.records.push({
        title:"New Record",
        rating:999,
        votes:,
        year:2013
    });
  };
```

RELATED SAMPLE: Webix - Angular : Grid

As you might have already noticed in the code above, all data manipulations with datatable data are done by **changing scope dataset**. In HTML, the **addRecords()** function is called by an **ng-click** directive:

```
<body ng-controller="webixTestController"> <!--Angular controller-->
  <div webix-ui view="datatable" webix-data="records" select="row"> <!--
Webix App-->
    ...<!--datatable config-->
  </div>
  <button ng-click="addRecord()">Add Row</button>
```

Datatable data will be updated the moment **$scope.records** is changed, which is ensured by Angular binding philosophy.

Using Component methods via Controller

To comply with Angular concept, **any function** you use with this or that component **should be stored in the app's controller**, even if it is the **component's own method**.

```
<body ng-controller="webixTestController">

    <div style="width:500px; height:200px;">
    <div webix-ui view="chart" id="mychart" webix-data="lines" type="bar"
value="#sales#" borderless="true"></div>
    </div>

    <button ng-click="changeLine('line')">Show Line Chart</button>
</body>
```

ChangeLine() controller function redefines the chart by setting another type for it:

```
Controller Code

  $scope.lines = [
    //chart dataset
    { id:1, sales:20, year:"02"},
    { id:2, sales:55, year:"03"},
    ...
  ];
```

```
$scope.changeLine = function(type){
  //methods are applied to the chart by its ID
  $$("mychart").define("type", type);
  $$("mychart").render();
};
```

RELATED SAMPLE: Webix - Angular : Chart

Event Handling With Angular

To understand Webix **event handling concept** consult the corresponding documentation article.

Angular JS offer a number of directives to catch on-page events (ngClick, ngChange, ngKeyPress, ngMouseMove) and all of them can be successfully used in Angular-Webix integrated applications.

```
<button ng-click="changeLine('line')">Show Line Chart</button>
```

RELATED SAMPLE: Webix - Angular : Chart

At the same time, Webix library offers a set component-specific events such as **onItemClick**, **onSelectChange**, **onAfterTabClick**, etc that cannot be handled by Angular means.

webix-event Directive

All Webix events can be handled with a special **webix-event** directive that calls a **scope function** to perform a desired action:

```
<body ng-controller="webixTestController">

<div webix-ui view="cols" type="space">
  <!--datatable column-->-
  <div view="datatable" webix-data="records"
       webix-event="onItemClick = showDetails(id, details)" select="row">
    <!--datatable columns-->
  </div>
  <!-- template column -->
  <div>
    Selected ID: { { selectedId } }
  </div>
</div>

</body>
```

In the code above, an onItemClick datatable event is handled with **showDetails()** scope function to show an ID of any selected item in a template as *{{ selectedId }}*.

Event Handler in Controller

```
App's Controller

var app = angular.module('webixApp', [ "webix" ]);

app.controller("webixTestController", function($scope){
  $scope.records = [ ..datatable data..];

  $scope.showDetails = function(id, details){
```

```
    $scope.selectedId = id.row; //setting value for selectedId
    $scope.$digest();
  }
});
```

RELATED SAMPLE: Webix - Angular : Events

Note that the function called by *webix-event* directive is limited to have only **two parameters**:

- **ID** of an item in question (sent by Webix event);
- **event details** that encompass all params sent by Webix event.

For instance, datatable specific onItemClick event has three params - *item info object (id, row, column)*, *native event object* and and *target HTML element*:

```
Webix Event Handling Pattern

grid.attachEvent("onItemClick", function(id, event, node){
    //... some code here ...
});
```

Handled by Angular, **event object** and **target element** are passed in **details** together with item info:

```
app.controller("webixTestController", function($scope){
 $scope.showDetails = function(id, details){
    $scope.selectedId = id.row;
    /*or you can get ID as*/
    $scope.selectedId = details[].row;

    $scope.eventType = details[1].target.getAttribue("class");
    $scope.nativeElement = details[2].tagName;

    $scope.$digest();
  }
}
```

Note that **HTML objects** (event, element) are not processed by Angular and the object itself will come as if empty. Nevertheless, you can still retrieve all of its properties:

```
$scope.showDetails = function(id, details){
    $scope.eventType = details[1]; //returns {}
    $scope.nativeElement = details[2] //returns {}

    $scope.eventType = details[1].type; //returns "click"
    $scope.nativeElement = details[2].tagName; //returns "DIV"
}
```

Handling of Mupliple Events per Component

Webix-event directive allows attaching any number of events at a time:

```
<div view="datatable" webix-data="records"
webix-event="onItemClick = show1(id, details); onAfterSort = show2(id,
details);">
    <!--datatable columns-->
</div>
```

Accordingly, you need to specify scope functions to handle each event (or a common function for the attached events):

```
app.controller("webixTestController", function($scope){
  $scope.show1 = function(id, details){
    ...
  };

  $scope.show2 = function(id, details){
    ...
  };
}
```

Study the sample to see how it works:

RELATED SAMPLE: Webix - Angular : Events

Changing View Visibility

From the very beginning it should be said that **Webix views can't be hidden** and **shown back** with Angular **ngShow** and **ngHide** directives that can still be used to manupulate visibility of pure HTML elements.

Let's remind us what are **Webix views**. They are created with the help of **webix.ui()** constructor that is called by **webix-ui** directive.

```
<body ng-app="Webix App">
    <div style="width:800px;"></div> <!--standard HTML element-->
    <div webix-ui type="wide">
        <!-- all divs inside this and including this are parsed as Webix
views-->
    </div>
</body>
```

To show and hide Webix Views a special **webix-show** directive should be used. Remember that all divs without *view* attribute are parsed as Webix templates. Views are initially **hidden**.

```
<div webix-ui type="wide">
    <div width="200" webix-show="showLeft">Left</div>
</div>
```

The **webix-show** directive declares a variable which value (boolean) can be afterwards changed on some event. Here button click is used:

```
<!-- showing view -->
<button ng-click="showLeft=true">Show Left</button>

<!-- hiding view -->
<button ng-click="showLeft=false">Hide Left</button>
```

RELATED SAMPLE: Webix - Angular : Visibility

Advanced

Backbone Integration

Backbone.js is a lightweight Javascript framework that allows for adding structure to client-side code in the MVC pattern. Backbone works in conjunction with its hard-weight dependency, **Underscore.js**, the utility library that provides a lot of the functional programming support and contains a number of functions as well as specialized helpers for function binding, javascript templating,and so on.

Backbone is based on the **MVC principle** that translates as Model, View and Controller classes where:

Bonuses of Backbone usage:

- The approach used in the framework helps get rid of unstructured 'Spaghetti' code in your applications;
- All the changes are performed on Models. The corresponding views respond to these changes are re-render themselves according to the new info, which eliminates the need of either looking through the DOM structure to find the necessary ID or update HTML manually;
- Tasks are clearly divided between application parts;
- RESTful API (Representational state transfer).

To get the profound knowledge of the framework, go to the Backbone.js homepage.

How it can be used

Webix and Backbone can be cross-integrated on multiple levels:

- Webix components can load data from backbone collection, check Backbone Collections for more details;
- Webix components can be wraped in a Backbone Views, check Backbone Views for more details;
- visibility of Webix Components can be controlled by Backbone router, check Working with Backbone Routers for more details.

Example of usage

To integrate Backbone framework into your webpage your should include links to it into the document head. Be attentive to specify the right relative paths to the places where the files are stored on your machine. The important point - library need to be included BEFORE webix.js:

```
<head>
  <script type="text/javascript"
src="http://code.jquery.com/jquery-1.7.1.min.js"></script>
  <script type="text/javascript" src="./common/underscore.js"></script>
```

```
  <script type="text/javascript" src="./common/backbone.js"></script>

  <link rel="stylesheet" href="../../codebase/webix.css" type="text/css"
media="screen" charset="utf-8">
  <script src="../../codebase/webix.js" type="text/javascript"
charset="utf-8"></script>
</head>
```

Creating Views

In an integrated app, pure Backbone views and Backbone-Webix views co-exist.

Backbone views are created under Backbone standard pattern:

```
cView = Backbone.View.extend({
    tagName: "h2",
    render: function(){
        $(this.el).html("Child View");
    },
});
var v2 = new cView();
v2.render();
```

Backbone-Webix integrated views resemble this pattern yet feature a **config** property where Webix configuration is stored. To instantiate such a view, a special **WebixView()** constructor is used:

```
var ui_config ={
    type:"wide", rows:[
        { template:"top", height:35 },
        { type:"wide", cols:[ ..cols ..]},
        { template:"bottom", height:35 }
    ]
};

var view  = new WebixView({
    el: ".app1_here", // will be rendered to div with "app1_here" class
    config:ui_config
}).render();
```

RELATED SAMPLE: Backbone integration - View

In the code above Webix layout with 3 rows and columns in the second row is created. It is rendered to a div block with "*app1_here*" class. As a rule, views should be **manually rendered** with a dedicated function.

For more into about Webix-Backbone integrated views, read the Backbone Views article.

Creating Models and Collections

```
FilmRecord = Backbone.Model.extend({});
```

```
FilmList = Backbone.Collection.extend({
    model: FilmRecord,
    url:"./common/data.json" //adding data to collection
});
```

Loading the Collection into a View

First, lets create a Webix data component, for instance view list:

```
View: Webix List

$(".app1_here").webix_list({
    id:"mylist", width:200,
    template:"#title#", select:true
});
```

Now, when we have a view and a model we can render the model into the list view with the syncing method:

```
//get data
var films = new FilmList();
films.fetch();

//sync collection data with list
$$("mylist").sync(films);
```

RELATED SAMPLE: Backbone Integration. Loading data from Collection

The code above renders list view and fills it with data from FilmList Collection that is comprised of FirmRecord Models.

The sync between data collection and the component can be as well removed by the opposite method:

```
$$("mylist").unsync(films);
```

Conclusion

It should be said that the above scenario is not the only way to use Webix and Backbone together. Check the below docs for more details.

Working with Backbone.js

- Loading from Backbone Collections
- Using Webix in Backbone Views
- Working with Backbone Routers

Backbone Collections

While Webix components have their own logic to load and save data, you can also use Backbone Collection as data source, and handle loading and saving in Backbone's way.

Data Loading

First, we need to create a Backbone collection and add data to it:

```
FilmRecord = Backbone.Model.extend({});
FilmList = Backbone.Collection.extend({
    model: FilmRecord,
    url:"./common/data.json"
});

films = new FilmList();
films.fetch(); //getting collection data
```

Each data component has sync method that can be used to load data from a Backbone Collection.

There can be several methods to add data from a Backbone collection to a Webix view. It depends on how you init the view.

```
By direct rendering into the needed node

var list = $(".app1_here").webix_list({
    template:"#title#", select:true
});

list.sync(films);
```

RELATED SAMPLE: Backbone Integration. Loading data from Collection

```
By creating a Backbone view that houses this component

MyView = WebixView.extend({
    config:{
        view:"list", //Webix list config
        id:"mylist",
        width:200,
        template:"#title#",
        select:true
    },
    afterRender:function(){
        this.getChild("mylist").sync(this.options.collection); // syncing
after rendering
    }
});

new MyView({
```

```
    el: ".app1_here",
    collection: films //defining collection as view option
}).render();
```

RELATED SAMPLE: Backbone Integration. Views and Models

After executing the sync command any changes to the "*films*" collection will be reflected in the list. Also, any changes, done by list API (adding, deleting, editing) will be reflected in a Backbone collection as well.

Data Saving

Nevetheless, changes in the component don't trigger saving to the Backbone Collection on their own. You need to use additional Webix events that are handled to enable data saving:

- **webix:add** & **webix:change** - fire when a new item is added or an item is updated in the view (here:list);
- **webix:remove** - fires when a record is removed from the view.

```
//films is Backbone collection with FilmRecord model
films.on("webix:add webix:change", function(model){ //no delimiter between events!
    model.save();
});
films.on("webix:remove", function(model){
    model.destroy();
});
```

RELATED SAMPLE: Backbone Integration. Saving data from collection.

Loading data in Form

Not all Webix components can work with Backbone Collections directly. Components that represent only one model can't use sync API, so instead of it you can use parse method to load data to them:

```
$$("form").parse( films.first().toJSON());
```

This command loads data from the model into Webix form, but any changes in the form or in the model are not reflected or saved.

Adding extra methods to Collections

Backbone collection can be extended to have as many various methods as you need. These are test methods that change collection data:

```
FilmRecord = Backbone.Model.extend({});

FilmList = Backbone.Collection.extend({
    //basic properties
```

```
    model: FilmRecord,
    url:"./common/data.json",

    //necessary for test buttons only
    addSample:function(){
        this.add(new FilmRecord({ //adds new record to collection
            title:"New Record"
        }));
    },
    deleteFirst:function(){ //removes first record from collection
        this.remove(this.first().cid);
    }
});
```

This is how these functions are triggered in the app:

```
<input type="button" value="Add" onclick='films.addSample()'>
<input type="button" value="Delete first" onclick='films.deleteFirst()'>
```

RELATED SAMPLE: Backbone Integration. Saving data from collection.

Backbone Views

This feature is fully optional, you can safely init Webix in any HTML container, so normal Backbone views will work fine enough.

Creating View

To create a new View you can use the next code:

```
var myview = new WebixView({
   el:".app1_here",
   config:{
      rows:[
        {template:"ABC"},
        {cols:[...]},
        {view:"some"}
      ]
   }
});
```

- **el** - HTML locator or HTML node of the element in which a view needs to be rendered;
- **config** - Webix UI configuration.

To render view somewhere on the page you should call:

```
myview.render();
```

RELATED SAMPLE: Backbone integration - View

Isolating Views

It's a good practive to **isolate** backbone views that are comprised of several Webix views so that you can use same IDs in another Backbone view on the page:

```
var ui_config={
    isolate:true, rows:[
        {template:"ABC", id:"abc"},
        {cols:[...]},
        {view:"some"}
    ]
};

var firstview = new WebixView({
   el:".app1_here",
   config: ui_config
```

```
});

var secondview = new WebixView({
   el:".app2_here",
   config:ui_config
});
```

Each Backbone view has a template with *abc* ID, but since config features an *isolate* property you can still refer to the necessary template (for instance in view functions that refer to these components).

You can also render any Webix view directly into an HTML element:

```
$(".app1_here").webix_list({
    id:"mylist", width:200, //config
    template:"#title#", select:true
});
```

RELATED SAMPLE: Backbone Integration. Loading data from Collection

Predefined methods of view

WebixView is an instance of Backbone.View.

There is a few methods that are already implemented in views created with this constructor:

- **render()** - renders the view;
- **destroy()** - calls the view destructor;
- **getRoot()** - get the top Webix control in the view;
- **getChild(id)** - get the child Webix control by its ID.

Nesting HTML view into Webix view

You can define a template view, and later render there a normal Backbone View by using the code like next:

```
//webix ui config
var ui_config ={
    isolate:true, type:"wide", rows:[
        { template:"top", height:35 },
        { type:"wide", cols:[
            { template:"left", id:"left" },
            { template:"center", id:"center" },
            { template:"right", id:"right" }
        ]},
        { template:"bottom", height:35 }
    ]
};

//create top level view
var v1 = new WebixView({
```

```
        config: ui_config,
        el: ".app1_here"
    });
    v1.render();

//html child view
cView = Backbone.View.extend({
    tagName: "h2",
    render: function(){
        $(this.el).html("Child View");
    },
});

//create child view
var v2 = new cView();
    v2.render();
    v1.getChild("left").setContent(v2.el);
```

- **getChild** method is used here to get layout child with "left" ID;
- **setContent(content)** method of Webix view is used here to attach Backbone View to template.

RELATED SAMPLE: Backbone integration - Nested Views

Nesting Webix view into another Webix view

Similar to the method above, you can render a Webix view as layout child view with the code like next:

```
//create top level view
var v1 = new WebixView({
        config: ui_config,
        el: ".app1_here"
    });
    v1.render();

//create child webix view
var v3 = new WebixView({
        config: ui_config,
        el:v1.getChild("right")
    });
    v3.render();
```

RELATED SAMPLE: Backbone integration - Nested Views

The place where a UI component need to be attached is defined through "el". New view (*v3*) will replace the child of *v1* view.

Extending View

Same as with Backbone View you can extend WebixView:

```
MyView = WebixView.extend({
    config:{
       view:"list", url:"data.json"
    },
    afterRender:function(){ ...},
    beforeRender:fucntion(){ ...}
    someMethod:function(){ ...}
});

var view1 = new MyView({ el : "areaA" });
```

There are 2 special methods that can be defined in such way:

- **beforeRender()** - will be called during rendering before creating a Webix view;
- **afterRender()** - will be called during rendering when Webix view is already created. It's a good place to **define event handlers** for Webix components of this Backbone view or **load collection data**.

And at the same time you can add you custom methods here.

```
MyView = WebixView.extend({
    config:{
        rows:[
            {view:"list", id:"mylist", ..},
            {view:"template", id:"details", ..}
        ]
    },
    afterRender:function(){
        //adding a handler to list selection
this.getChild("mylist").attachEvent("onAfterSelect",_.bind(this.listSelect
ed,this));
        //syncing view data with collection
        this.getChild("mylist").sync(this.options.collection);
    },
    listSelected:function(id){
        this.getChild("details").parse({ "id": id });
    }
});
```

RELATED SAMPLE: Backbone Integration. Views and Models

Webix event handling pattern is described in the corresponding documentation article.

Working with Backbone Routers

Take that you have a list and a template and you want to show either one or another by clicking on their elements. There you can have different patterns depending on how you init the components:

- Backbone Router for pure Webix Views
- BackBone Router for Backbone-integrated Views
- BackBone Router for Updating Layout Sections

Backbone Router for pure Webix Views

To create a UI we can use a Webix code like next:

```
webix.ui({
  container:"app1_here", rows:[
     {template:"Click on item", type:"header" },
     {cells:[
       {view:"list", url:"data.json", template:"#title#", select:true,
id:"mylist"},
       {template:"<button onclick='history.back()'>Back</button>",
id:"details" }
     ]}
  ]
});
```

Now, let's create a **Backbone Router**.

```
var routes = new (Backbone.Router.extend({
    routes:{
        "":"index",
        "films/:id":"details" //details() f-n is triggered
    },
    details:function(id){
        $$("details").show(); //it shows template with film details
    },
    index:function(){
        $$("mylist").show();
    }
}));
```

As you can see the router shows the necessary view based on current document URL. After switching to "details" template, the application receives a hashbang url *localhost/myapp.html#films/id* to show in it the ID of a currently selected list item.

To have the app working we need to add a few extra lines of code:

```
$$("mylist").attachEvent("onAfterSelect", function(id){  //navigation rule
    routes.navigate("films/"+id, { trigger:true });
});
Backbone.history.start(); //router starts memorizing its steps
```

RELATED SAMPLE: Routes. The Webix way.

These lines define the **navigation rule** after selecting an item in the list and **starts the router**. Template, in its turn, uses an existing router and makes a **step back** on button click (look at its code above).

BackBone Router for Backbone-integrated Views

Now lets try to recreate the same app, but stick closer to the native Backbone approach. Instead of creating Webix UI directly, lets define Backbone-Webix integrated views for both states:

```
var layout = new WebixView({
    el:"#app1_here",
    config:{
        rows:[
            { template:"Click on item", type:"header" },
            {
                view:"list", url:"common/data.json",
                template:"#title#", select:true,
                id:"mylist"
            }
        ]
    }
});

DView = Backbone.View.extend({
    el:"#app1_here", //the same element as above
    tagName: "div",
    render: function(){
        $(this.el).html("<button onclick='history.back()'>Back</button>");
    },
});
var template = new DView();
```

Now a router looks a bit more different:

```
var routes = new (Backbone.Router.extend({
    routes:{
        "":"index",
        "films/:id":"details" //details() f-n is triggered
    },
    details:function(id){
        template.render(); //it renders the template into its "el"
    },
```

```
    index:function(){
        layout.render(); //rendereing layout back
        //attaching select event after rendering
        $$("mylist").attachEvent("onAfterSelect", function(id){
            routes.navigate("films/"+id, { trigger:true });
        });
    }
}));
```

RELATED SAMPLE: Routes. The Backbone way.

Instead of switching between views in Webix layout we are rendering different Backbone Views into one and the same element depending on current state. Still, the resulting behavior is the same.

Yet, the router should be triggered separately at once after the app has been inited:

```
//init app
Backbone.history.start();
```

Updating Layout Section

When using views we can update not only the top view, but any section of Webix layout in it.

Lets create 3 views, one is layout while the two others (list and template) are children of this layout. They will be shown in this layout in different cases:

```
var template = new WebixView({
    config:{ template:"<button onclick='history.back()'>Back</button>" }
});
var list = new WebixView({
    config:{
        view:"list", url:"common/data.json",
        template:"#title#", select:true,
        id:"mylist"
    }
});
//layout wil Webix config
var layout = new WebixView({
    el:"#app1_here",
    config:{
        rows:[
            { template:"Click on item", type:"header" },
            { }
        ]
    }
});
```

The router for such pattern is as follows:

```
var routes = new (Backbone.Router.extend({
```

```
    routes:{
        "":"index",
        "films/:id":"details"
    },
    details:function(id){
        //setting node for template as layout child view
        template.el = layout.root.getChildViews()[1];
        template.render();
    },
    index:function(){
        //setting node for list as layout child view
        list.el = layout.root.getChildViews()[1];
        list.render();
        $$("mylist").attachEvent("onAfterSelect", function(id){
            routes.navigate("films/"+id, { trigger:true });
        });
    }
}));
```

The code above locates the necessary cell in the layout and renders the needed view into the target cell.

RELATED SAMPLE: Routes. Changing the Webix Layout

Integration with DHTMLX Components

DHTMLX is a **component Javascript Library** for building enterprise-level Web apps with clean interface, fast performance, and rich user experience. Since, DHTMLX is our friendly project, Webix components resemble DHTMLX ones, so the views from both libraries can naturally co-exist in one and the same application.

JS file that contains the integration code can be taken from Github while the needed components are downloaded from DHTMLX site.

DHTMLX is comprised from **various components** each of which can be downloaded separately to minimize the total weight of dependent scripts in the application. Yet, you need to enumerate them all in your document head.

```
<!-- compulsory: JS file with the code that integrated Webix and DHTMLX -
->
<script type="text/javascript" src="./dhtmlx.js"></script>

<!--up to your choice: dependencies for the needed components -->
<script type="text/javascript" src="./dhtmlx/dhtmlxcommon.js"></script>
<script type="text/javascript" src="./dhtmlx/dhtmlxcontainer.js"></script>
<script type="text/javascript" src="./dhtmlx/dhtmlxlayout.js"></script>
<link rel="stylesheet" href="./dhtmlx/dhtmlxlayout.css" type="text/css">

<!--up to your choice: skin for the needed component -->
<link rel="stylesheet" href="./dhtmlx/skins/dhtmlxlayout_dhx_skyblue.css">
```

The compulsory dependency creates a **dhtmlx** object on the page. It helps you to set the chosen skin:

```
dhtmlx.skin = "dhx_skyblue";
```

Now lets, create some DHTMLX component. Let it be the DHTMLX Layout where we put Webix layout and Webix Datatable.

```
//creating layout with 3 cells (3U type)
dhxLayout = new dhtmlXLayoutObject("parentId", "3U");

//setting header texts for each cell
dhxLayout.cells("a").setText("Main Page");
dhxLayout.cells("b").setText("Layout");
dhxLayout.cells("c").setText("DataTable");
```

To put Webix component into DHTMLX layout, use **attachWebix()** method with a configuration object inside:

```
//inserting Webix layout into the second cell
dhxLayout.cells("b").attachWebix({
```

```
    type:"wide", rows:[
        { template:"1" },
        { type:"wide", cols:[
            { template:"2" },
            { template:"3" }
        ]}
    ]
});

//inserting Webix datatable into the third cell
dhxLayout.cells("c").attachWebix({
    view:"datatable",
    columns:[...],
    data:[...]
});
```

Related Sample: Integration with DHTMLX Layout

All methods of DHTMLX Layout you can find in DHTMX documentation.

DIY - Custom Integration Pattern

Webix offers a number of ready-made solutions that integrate it with other Javascript libraries and frameworks.

- integration with structural frameworks BackBoneJS and Angular JS;
- integration with popular text editors;
- intergation with charts and planning tools.

However, it's not possible to provide support for all the libraries or tools that exist in big quantities on the Web. If you haven't found the needed solution among built-in ones, you can write **your own integration pattern**. Webix offers rather an elegant way to do this.

The main thing here is to create a new Webix component that will display the needed tool on the page.

```
webix.protoUI({
    name:"myCustomName",
    $init:function(config){..},
    ..//other methods
}, webix.ui.view);
```

This prototype component should inherit from view, a base class for all the ui-related components. A pure view doesn't contain any into and acts as a placeholder for the real component.

It can have any number of properties and methods with two of them being mandatory:

- **name** - property that defines view name you will use during initialization;
- **$init**(config) - function that is executed on component initialization. It's argument is the component's configuration object.

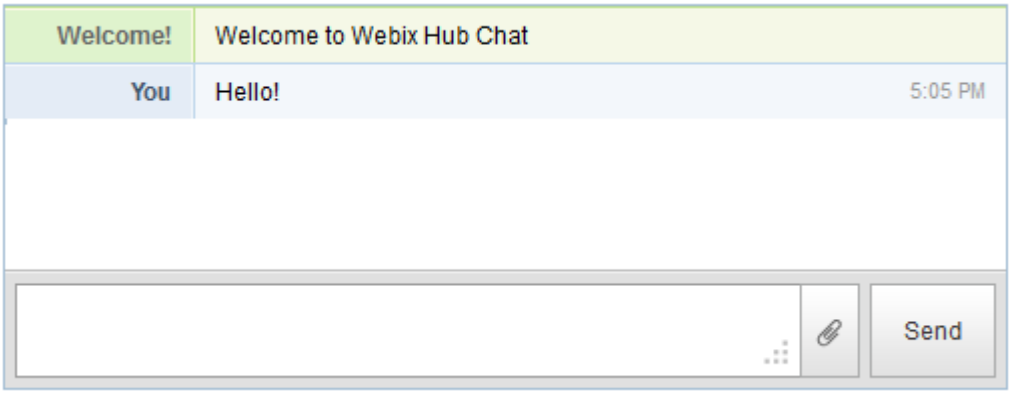
The following code will then instantiate the above-created object and display it on the page:

```
webix.ui({
    view:"myCustomName",
    ..//config
});
```

To learn more about how to **create a custom object in Webix**, go to the corresponding documentation article.

HipChat Integration

Let's take a HipChat, a popular professional chat solution that has lately got an apportunity to work on the web.




```
Prototype Component

webix.protoUI({
    name:"hipchat",
    $init:function(config){
        //configuration params
        var params = [
            "anonymous=true",
            "minimal=true",
            "timezone="+config.timezone,
            "welcome_msg="+(config.welcome_msg||"Welcome to Webix Hub
Chat")
        ];
        //building full url
        var
url=config.url+(config.url.indexOf("?")>?"&":"?")+params.join("&");
        if (url.indexOf("https://") !== )
            url = "https://" + url;

        //loading the hipchat
        var html="<iframe src='"+ url+"'frameborder='0' width='100%'
height='100%'>
                    </iframe>";
        this.$view.innerHTML = html;
    },
}, webix.ui.view);
```

Initing HipChat on page:

```
webix.ui({
    view:"hipchat",  url: "YOUR_GUEST_ACCESS_URL", timezone: "PST"
});
```

You get the URL value through Hipchat client when you share the room access.

Related Sample: HipChat Integration

Auxiliary Resources

- Top 10 Helpers You Should Know About
- Ajax operations
- Advanced Drag-and-Drop
- Data drivers
- Number formatting methods
- Date formatting methods
- Ui.resizearea. Common tips

Intermediate

Top 10 Helpers You Should Know About

webix.ui

This method is used mainly to **instantiate Webix component** of any complexity level, either it is a control or the complete application layout.

It converts a **JSON structure** passed into it as parameter to client-side layout objects - this is what Webix is used for:)

```
webix.ui({ view:"", ... });
```

Normally, you need one **webix.ui()** contructor for the whole application while any **window** or **popup** needs an extra one as windows lie above page layout.

```
webix.ui({ view:"window", ... }).show();
```

More opportunities of using this method are described in the chapter Rebuilding Application Layout.

webix.ready

This method is a cross-browser alternative to the **onDocumentReady** event and can be used instead of **onload()** method. Code, which was put inside of it, will be called just after the complete page has parsed, protecting you from potential errors. The thing is optional but we recommend to use it.

It can be used multiple times.

```
webix.ready(function(){
    webix.ui({
        container:"box",
        view:"window",
               ...
    });
})
```

API reference

webix.bind

An easy way to bind a function to an object (inside bound function, **this** will point to an object instance).

```
var t = webix.bind(function(){
    alert(this);
}, "master");

t(); //will alert "master"
```

webix.copy

This method allows you to make a full (independent) copy of object

```
var new_obj = webix.copy(source_obj)// full copy
```

API reference

webix.extend

Adds method and properies of source object to the target object

```
var target_obj = new webix.ui.toolbar(config);
webix.extend(target_obj, webix.Movable);
```

Also, it can be used as one more method to get an object copy. In such case, you must specify an **empty object** as second parameter.

```
var new_obj = webix.extend({},source_obj) //object based copy
```

API reference

webix.exec

Next helper allows to execute code string at runtime.

```
webix.exec(" var t = 2; ");
```

It is have advantage over eval - variables defined in code will be defined in global scope, the same as in case of normal js code.

API reference

webix.delay

Delay routine. If you need to delay some code from executing at runtime you are at the right place. **webix.delay** waits for the specified number of milliseconds and then executes the specified code.

```
webix.delay(webix.animate, webix,[node,animation],10)
```

API reference

webix.uid

A tool for getting an unique id. (id is unique per session, not globally unique)

```
var name = webix.uid();
```

API reference

webix.event

webix.event is used to attach event handler and **webix.eventRemove** - to remove it.

```
var display_handler = webix.event(document.body,'click',function(e){})
...
webix.removeEvent(display_handler);
```

API reference

webix.html.offset

With the help of this method you can get the position of any html element.

```
var offset = webix.html.offset(ev.target)
```

API reference

webix.html.addCss

Two methods implementing css manipulations. **webix.html.addCss** lets to add css class to element, **webix.html.removeCss** - removes some defined css class.

```
webix.html.addCss(this._headbutton, "collapsed");
...
webix.html.removeCss(this._headbutton, "collapsed");
```

API reference

All Webix helpers are listed in the related of API Reference section.

Intermediate

Ajax operations

Sending Data

Simplest ajax call can be done as:

```
webix.ajax("some.php", function(text){
    webix.message(text); //show server side response
});
```

The code above makes an asynchronous GET request to the "some.php" script and shows it response through callback function.

Passing parameters

You can pass extra parameters by including them in the request string:

```
webix.ajax("some.php?action=info&id=123");
```

by a separate string with parameters:

```
webix.ajax().get("some.php", "action=info&id=123");
```

or by using a separate JSON object:

```
webix.ajax().get("some.php",{ action:"info", id:123 });

//nested JSON object is possible
webix.ajax().get("some.php",{ action:"info", id:{ row:123, column:"title"
}});
```

In case of object notation, the lib will escape values automatically.

Post mode

Same function can be used to make post requests:

```
webix.ajax().post("some.php");
```

Here you can use two methods to provide extra params:

```
//$_POST["action"]="info", $_POST["id"]="123"
webix.ajax().post("some.php","action=info&id=123");
```

```
webix.ajax().post("some.php", { action:"info", id:123});
webix.ajax().post("some.php", { action:"info", id:{ row:123,
column:"title" }});
```


Note that when passing extra parameters into the server script by a separate string/object, you should specify the type of a request (GET or POST).

Sync mode

There are two ways that Ajax can access the server:

- synchronous (before continuing, script waits for the server back reply).
- asynchronous (script continues page processing and handles reply as it comes).

By default all request done in async. mode, but there is a sync. alternative

```
webix.ajax().sync().get("some.php");
webix.ajax().sync().post("some.php");
```

Callback

For async ajax calls you can define a callback function in the last parameters of the method. For GET and POST requests the syntax is the same.

The callback function takes the following arguments:

- **text** - full text of response;
- **data** - response parsed as JSON or XML, whichever is applicable;
- **XmlHttpRequest** - loader object;

GET requests:

```
webix.ajax("some.php", function(text, data, XmlHttpRequest){
    //..
});

webix.ajax().get("some.php", function(text, data, XmlHttpRequest){
    //...
});
```

POST request:

```
webix.ajax().post("some.php", function(text, data, XmlHttpRequest){
    //...
});
```

If you pass parameters to the serverside script, the syntax is as follows:

```
webix.ajax().get(url, params, callback);
webix.ajax().post(url, params, callback);
```

It's possible to define two callback methods, one for valid server side response and second for the error response

```
webix.ajax("some.php",{
    error:function(text, data, XmlHttpRequest){
        alert("error");
    },
    success:function(text, data, XmlHttpRequest){
        alert("success");
    }
});
```

If you need to trigger multiple callbacks after method finishing, you can define an array of callbacks as last parameter

```
//GET requests
webix.ajax().get(url, [array of callbacks]);
webix.ajax().get(url, params, [array of callbacks]);

//POST requests
webix.ajax().post(url, [array of callbacks]);
webix.ajax().post(url, params, [array of callbacks]);
```

In some cases you may need to provide some extra data for callback, which can be done in the next way:

```
function after_call(text){
     alert("Call: "+this.data+", response: "+text);
};
webix.ajax().get("some.php", after_call, { data:"my ajax"});
```

Basically, the parameter after callback will be accessible as "this" in the callback method

For **sync** requests, there is no callback, because there is no need for waiting

```
var xhr = webix.ajax().sync().get("some.php");
var text = xhr.responseText;
```

Extra headers

If server side code requires some extra headers you can send them as

```
webix.ajax().headers({
    "Sent-by":"My Script" //any custom headers can be define in such way
```

```
}).get("./data.php");
```

You can use headers in combination with async, get and post methods

JSON sending

If you are working with websirvice, you may need to send json content to the server side, it can be done as

```
webix.ajax().headers({
    "Content-type":"application/json"
}).post("./data.php", JSON.stringify(payload), function(text){
    //optional callback here
});
```

Incoming Data Parsing

Plain response

In common case you can use text parameter of callback to access the data

```
webix.ajax("./data.php", function(text){
    webix.message(text);
})
```

JSON response

While you can use the above code to access raw text response, there is a more neat solution

```
webix.ajax("some.php", function(text,data){
    //text = '{ some :"abc"}'
    alert(data.json().some); //abc
});
```

the second parameter of response can provide you with a ready-to-use json object

XML response

Same as above, you can use

```
webix.ajax("some.php", function(text,xml){
    //text = '<data some='abc'></data>'
    alert(obj.xml().some); //abc
});
```

If necessary you can access raw xml object by using data.rawxml();

Other formats

```
CVS
```

```
webix.ajax("some.php", function(text,xml){
    var obj = webix.DataDriver.csv.toObject(text,xml);
    alert(obj[][]); //row and column indexes
});
```

JSArray

```
webix.ajax("some.php", function(text,xml){
    var obj = webix.DataDriver.jsarray.toObject(text,xml);
    alert(obj[][]); //row and column indexes
});
```

HTML

```
webix.ajax("some.php", function(text,xml){
    //text = '<ul><li some='abc'></ul>'
    var obj = webix.DataDriver.html.toObject(text,xml);
    alert(obj[].some); //abc
});
```

Extra parameters

If all above is not enough you can access raw xmlHttpRequest and poll it for additional data

```
webix.ajax("some.php", function(text,xml, ajax){
    alert(ajax.status);
});
```

Getting Binary Data

Webix Ajax class offers a pattern for retrieving binary data objects from the server.

It can be done with a dedicated **response()** method that allows to set responseType directly. Now only **"blob"** type is fully supported and **"document"** with some restrictions.

```
webix.ajax().response("blob").get("patch.zip", function(text, data){
  //data - is a data blob
});
```

Functionality will not work in IE8-IE9 as these browsers cannot work with blob objects at all.

Promise API for Ajax requests

Webix is integrated with the Promiz.js library, which allows treating the result of asynchronous operations without callbacks.

In other words, any Ajax request performed by Webix returns a **promise** object that represents the eventual result of this request and can be treated by **then()** method that is automatically executed when server

response arrives.

```
//"promise" object is returned by either of these methods
var promise = webix.ajax("some.php");
var promise = webix.ajax().get("some.php");
var promise = webix.ajax().post("some.php");

//realdata - data that came from server
promise.then(function(realdata){
    ...
});
```

See how it is implemented in Webix data loading pattern.

Forming a chain of requests with Promises

```
webix.ajax("someA.php").then(function(realdataA){
    return webix.ajax("someB.php");
}).then(function(realdataB){
    return webix.ajax("someC.php")
}).then(function(realdataC){
    ...
});
```

Error handling with Promise objects

then() method can be used to treat errors:

```
var promise = webix.ajax("some.php")
promise.then(function success(realdata){}, function error(err){});
```

Additionally, **fail()** method can be used for error handling:

```
var promise = webix.ajax("a.php");
...
promise.then(function(realdata){ /*success*/});
promise.fail(function(err){/*error*/});
```

Or, you can do it in one and the same function:

```
promise.then(function(realdata){
    //success
}).fail(function(err){
    //error
});
```

All the **methods of the promise object** come from the **Promiz.js** lib and can be found there.

webix.promise interface

Webix offers an interface for working with promise objects - **webix.promise** - featuring a set of methods that

duplicate Promise object methods.

It can be used, for instance, to treat several Ajax requests and perform action when callbacks have been received from all of them:

```
a = webix.ajax("someA.php");
b = webix.ajax("someB.php");
c = webix.ajax("someC.php");


webix.promise.all(a,b,c).then(function(a,b,c){
    ...
});
```

Advanced

Advanced Drag-and-Drop

Here we speak about **advanced drag-n-drop options** and find solutions how to make any elements on the page draggable and control each stage of DnD.

For simple dragging and dropping of items within one and the same component as well as between different views and their instances - study the atricle about built-in Webix DnD options.

DnD syntax

If you want to make some element on page draggable you can use

```
webix.DragControl.addDrag(node);
```

where *node* - id or html node object (or object itself)

RELATED SAMPLE: Drag-and-Drop from HTML

If you want to define some area as drop target you can use

```
webix.DragControl.addDrop(node);
```

where *node* - id or html node object (or object itself).

Controlled DnD

If you want to control different aspects of dnd process, you can use the above methods but with an extra **control parameter** (**$drag** or **$drop**) with a custom function:

```
//for dragging
webix.DragControl.addDrag(node, $drag:function(source, target,
event){...});
//for dropping
webix.DragControl.addDrop(node, $drop:function(source, target,
event){...});
```

The control function takes three parameters:

- **source** - source html object;
- **target** - target html object;
- **event** - native event.

This is how you can set value of an input by drag-n-drop

```
webix.DragControl.addDrop("mytext", {
    $drop:function(source, target, event){
        var dnd = webix.DragControl.getContext();
        target.value = dnd.from.getItem(dnd.source[]).title; // setting
source item title as input value
    }
});
```

RELATED SAMPLE: Drag-and-Drop to HTML

DnD context

Context object of dnd can be accessed as:

```
var state = webix.DragControl.getContext();
```

it is also available as parameter in all dnd related events

```
state = {
    source:[1,2,3], //array of IDs for dragged elements
    target:id, //ID of currently active target item
    from:some_obj, // reference to the source object
    to:some_obj, // reference to the target object
    html:text //optional, custom text, which is rendered as drag-item
}
```

Set of the drag control methods

There are 6 DnD control event: 3 of them are used for dragged element (addDrag) and 3 - for drop target (addDrop). All these events can be used to define a custom behavior for elements included in current dra-n-drop.

addDrag() related events:

- *onDrag* - fires when the elements is dragged;
- *onDragCreate* - fires the moment drag has been created;
- *onDragDestroy* - fires when drag has been completed;

addDrop() related events:

- *onDragIn* - fires when a dragged element enters the drop area (If you want to deny drop area, return false).
- *onDragOut* - fires when a dragged elements leaves the drop area.
- *onDrop* - fires when you drop the dpagged element.

```
{
  onDragIn:function(source, target, event){ ... },   //drag moves in
```

```
potential drop area
  onDragOut:function(source, target, event){ ... },  //drag moves out from
the drop area
  onDrop:function(source, target, event){ ... },     //drag was released
  onDrag:function(source, target, event){ ... },     //drag is started
  onDragCreate:function(from, event){ ... }     //dnd is started
  onDragDestroy:function:(from, text){ ... } //dnd is finished
}
```

Where:

- **source** - source html object;
- **target** - target html object;
- **event** - native event;
- **from** - control object related to source of dnd;
- **text** - content of drag marker.

If any of onDrag handlers redefined - there won't be the default processing of the action, code expects that your custom handler will do all job.

DnD masters

Master is a component object or a hash code of the control methods. It specifies how element will behave itself in the drop target.

If you specify the master, it will answer for the appropriate dnd behavior. If you don't do it, the standard dnd processing will occur.

For the methods, where source and target masters are not provided, you can get them by the getMaster method:

```
var source_master = webix.DragControl.getMaster(source);
var target_master = webix.DragControl.getMaster(target);
```

Interface-based DnD

You can make an element draggable by adding one of the 2 appropriate interfaces:

- webix.Movable

Low level interface. A component which inherits it can be moved by dnd (to learn how to inherit the interface read the Component Copying and Extending article).

- webix.DragItem

High level interface. Adds dnd ability to component which uses DataStore (to learn how to inherit the interface read the Component Copying and Extending article).

Samples of usage

Drag from custom HTML

```
<div package="DragPackage" version="1.0" maintainer="Webix Team"
    id="drag_1"
    style='width:150px; height:50px; color:white; background-color:navy;'>
    Drag me into the dataview
</div>

webix.DragControl.addDrag("drag_1");
data1.attachEvent("onBeforeDrop",function(context){
    if (context.from == webix.DragControl){
        this.add({
            Package:context.source.getAttribute("package"),
            Version:context.source.getAttribute("version"),
            Maintainer:context.source.getAttribute("maintainer")
        }), this.getIndexById(context.target || this.getFirstId());
        return false;
    }
    return true;
});
```

Drag to custom HTML

```
<div id="data_container2" style="width:400px;height:396px;border:1px solid
red;">
    Drop some item here
</div>

webix.DragControl.addDrop("data_container2",{
    onDrop:function(source, target, d, e){
        var context = webix.DragControl.getContext();
        var item = context.from.getItem(context.source[]);

        var d = document.createElement("DIV");
        d.innerHTML = item.Package+" - "+item.Version;
        target.appendChild(d);
    }
});
```

Advanced

Data drivers

There are 5 default data drivers:

- *json*
- *xml*
- *html*
- *csv*
- *jsarray*

Customizing data driver

XML

In case of XML you can redefine

Element	Type	Code
data tag	xpath	`webix.DataDriver.xml.records = "/*/item";`
configuration tag	xpath	`webix.DataDriver.xml.config = "/*/config";`
child tag	tag name	`webix.DataDriver.xml.child = "item";`

For example to alert the default processing of XML data you can add the following line to your code:

```
webix.DataDriver.xml.records = "/*/book";
webix.DataDriver.xml.child = "part";
```

The line above will redefine data parsing logic for all XML sources; if you wish to put apart default and custom logic you need to create a custom XML processor on the base of the default one.

```
var myxml = webix.DataDriver.myxml = webix.copy(webix.DataDriver.xml);
myxml.records = "/*/book";
myxml.child = "part";
```

RELATED SAMPLE: Tree: XML Dataset

Child tags can be specified by a function:

```
var myxml2 = webix.DataDriver.myxml2 = webix.copy(webix.DataDriver.xml);
myxml2.records = "/*/book";
myxml2.child = function(obj){
    if (obj.$level == 1)
        return obj.part;
```

```
    if (obj.$level == 2)
        return obj.page;
}
```

RELATED SAMPLE: Tree: XML Dataset

CSV

It is possible to define row and cell separators:

```
webix.DataDriver.csv.row = "\n";
webix.DataDriver.csv.cell = ",";
```

The above line resets processing of all the SCV sources in the application. To avoid this, create a custom CSV format:

```
var myscv = webix.DataDriver.myscv = webix.copy(webix.DataDriver.scv);
csv.row = "\n";
csv.cell = ",";
```

HTML

You can redefine which tag in html will be processed as data tag:

```
webix.DataDriver.html.tag = "LI";
```

The above line resets processing of all the HTML sources in the application. To avoid this, create a custom HTML format:

```
var myhtml = webix.DataDriver.myhtml = webix.copy(webix.DataDriver.html);
html.tag = "LI";
```

JSON

You can redefine which object propery stores sub-items (used for hierarchical dataset, like one for tree, treegrid, grouplist etc.)

```
webix.DataDriver.json.child = "data";
```

The above line resets processing of all the JSON sources in the application. To avoid this, create a custom JSON format:

```
var myjson = webix.DataDriver.json = webix.copy(webix.DataDriver.json);
myjson.child = "data";
```

Methods of data driver

```
//convert string to object
var data = driver.toObject(data);
//return array of all records in datasource
var records = driver.getRecords(data)
//return single data object
var data = driver.getDetails(records[]);
```

XML

```
//run xpath
var elements = driver.xpath("/some/xpath", data);
//convert xml tag to json object
var obj = driver.tagToObject(elements[]);
//convert string to js data-types, numbers and booleans
obj = driver.assignTypes(obj);
```

Creating new data driver

In addition to built in data types, it possible to define custom ones

The structure of driver is the following:

```
Webix.DataDriver.some={ //some - the name of the type
    toObject:function(text,xml){
                ...
        return text;
    },
    getRecords:function(data){
                var result = [];
                ...
        return result;
    },
    getDetails:function(data){
                var result = {}
                ...
        return result;
    },
    getInfo:function(data){
        return {
         _size:some,
         _from:other
        };
    }
};
```

You can check the structure of the data driver in Webix sourse code (webix_debug.js)

For a start, we have some data that we want to use as a new data type. The first thing we need to make with this data - conversion into an intermediate object (an object that will be used as an input parameter in the other functions).

1 . So, our first step - **toObject(text, xml)** function. The function is called after data loading or directly in the **parse** method and returns the intermediate data object.

- *text* - incoming data.
- *xml* - xml object (for xml loading scenario. For other data types can be ignored).

2 . Then, we form an array of records using data from our intermediate object - **getRecords(data)** function.

- data - the intermediate object from the previous step.

Having an array of records we need to specify a set of properties for each single record - the third step - **getDetails(data)**.

- data - an element of the array from the previous step.

3 . The last action has sense just for dynamic data loading scenarios. It's a function that gets count of data and position at which new data needs to be inserted.

4 . The fourth step - **getInfo(data)**. The function returns count of data and the appropriate position mentioned above: '0' if a value is unknown or unnecessary.

- data - the intermediate object from step 1

So, making these four or even three steps you can specify any data type and then use it while developing.

Creating new data driver based on an existing one

Creating a completely new driver is not the most common task, but a user may need to alter the processing logic of some built-in driver.

It can be done with the help of Webix extending pattern that allows to extend any JSON object:

```
webix.DataDriver.custom = webix.extend({
    records:"/data/items/item",
    getDetails:function(obj){
        var res = {};
        var cells = this.xpath(obj, "cell");
        for (var i = ; i < cells.length; i++)
            res[cells[i].getAttribute("id")] = this.nodeValue(cells[i]);
        return res;
    }
}, webix.DataDriver.xml);
```

RELATED SAMPLE: Loading from custom XML

Intermediate

Number formatting methods

There are 2 predefined methods for number formatting:

Localization

```
var string1 = webix.i18n.numberFormat("123.45");
var string2 = webix.i18n.intFormat("123.45")
var string3 = webix.i18n.priceFormat("123.45");
```

They will format data according to rules stated in the current locale.

Applying locales within component:

```
//datatable
{ header:"LongDate", width:170,  id:"start",
format:webix.i18n.numberFormat }

//other components
template:function(obj) {return webix.i18n.numberFormat(obj.start); }
```

Dive into Date and Number localization article.

Custom Formatting

If you need to format number with custom formatting rules you can use methods in the Number class that offers two methods: format and numToStr.

Here you need to specify rules right in a data component.

```
var string1 = webix.Number.format("123.45",{
    groupDelimiter:",",
    groupSize:3,
    decimalDelimiter:".",
    decimalSize:2
});
```

Applying format within a component:

```
//datatable
{ header:"LongDate", width:170,  id:"votes",
format:webix.Number.numToStr({
                        groupDelimiter:"",
                        groupSize:,
```

```
                        decimalDelimiter:",",
                        decimalSize:5})
}

//other components
template:function(obj) {
    return webix.Number.numToStr(obj.votes, {
        groupDelimiter:"",
        groupSize:,
        decimalDelimiter:",",
        decimalSize:5
    });
}
```

RELATED SAMPLE: Using number templates

Intermediate

Date formatting methods

Formatting and parsing

There 8 predefined methods for date formatting, which can be used to convert dates to strings and strings to dates

```
Convert dates to strings

var str1 = webix.i18n.dateFormatStr(date_obj);  //date
var str2 = webix.i18n.longDateFormatStr(date_obj);//long date
representation
var str3 = webix.i18n.timeFormatStr(date_obj);  //time
var str4 = webix.i18n.fullDateFormatStr(date_obj);//date + time
var str5 = webix.i18n.parseFormatStr(date_obj); //used for date
serialization
```

```
Convert strings to dates

var date1 = webix.i18n.dateFormatDate(string);  //date
var date2 = webix.i18n.longDateFormatDate(string);//long date
representation
var date3 = webix.i18n.timeFormatDate(string);  //time
var date4 = webix.i18n.fullDateFormatDate(string);//date + time
var date5 = webix.i18n.parseFormatDate(string); //used for date parsing
```

Formats can be defined by locales to adapt numbers and dates to the target location.

If necessary you can create your own conversion methods

```
Custom date convert

var format = webix.Date.dateToStr("%Y.%m");
var string = format(new Date()); //2012.05

var parser = webix.Date.strToDate("%Y.%m");
var date = parser("2012.05");
```

Supported format options

- %d - the day as a number with a leading zero (01 to 31);
- %j - the day as a number without a leading zero (1 to 31);
- %D - the day as an abbreviation (Sun to Sat);
- %l - the day as a full name (Sunday to Saturday);

- %m - the month as a number without a leading zero (1 to 12);
- %n - the month as a number with a leading zero (01 to 12);
- %M - the month as an abbreviation (Jan to Dec);
- %F - the month as a full name (January to December);
- %y - the year as a two-digit number (00 to 99);
- %Y - the year as a four-digit number (1900□9999);
- %h - the hour based on the 12-hour clock with a leading zero (00 to 11);
- %g - the hour based on the 12-hour clock (0 to 11);
- %H - the hour based on the 24-hour clock with a leading zero (00 to 23);
- %G - the hour based on the 24-hour clock (0 to 23);
- %i - the minute as a number with a leading zero (00 to 59);
- %s - the second as a number without a leading zero (00 to 59);
- %a - displays am (for times from midnight until noon) and pm (for times from noon until midnight);
- %A - displays AM (for times from midnight until noon) and PM (for times from noon until midnight).

For instance, **"%d-%F-%Y,%D"** will display the date as 11-September-2012, Tue.

strings like 20120502, where there are no separators between numbers, can't be parsed as dates

Ui.resizearea. Common tips

Limiting the area available for resizing

By default, the area available for resizing is the whole viewport. To limit the area, you should initialize the component in a container with **style='position:relative'**

```
<div id='mydiv' style='position:relative;width:600px;
height:300px;border:1px solid;'></div>

webix.event(document.getElementById("mydiv"),"mousedown",function(e){
            var resizeStick = new webix.ui.resizearea({
                    container:"mydiv",
                    dir:"y",
                    eventPos:e.pageY,
                    start:e.pageY-
webix.html.offset(webix.toNode("mydiv")).y
            });
})
```

DIVs created while resizing

While moving a **resizearea** object, 3 DIVs are created:

Table 1 Vertical resizing

Container	Description
&ltdiv class="webix_resize_area"&gt&lt/div&gt	the area available for resizing
&ltdiv class="webix_resize_handle_y"&gt&lt/div&gt	the movable container
&ltdiv class="webix_resize_origin_y"&gt&lt/div&gt	the initial fixed container

Table 1 Horizontal resizing

Container	Description
&ltdiv class="webix_resize_area"&gt&lt/div&gt	the area available for resizing
&ltdiv class="webix_resize_handle_x"&gt&lt/div&gt	the movable container
&ltdiv class="webix_resize_origin_x"&gt&lt/div&gt	the initial fixed container


.dhx_resize_area
.dhx_resize_handle_y
.dhx_resize_origin_y

Migration to latest Webix version

Webix 2.2 -> 2.3

Parse format

In version 2.2 the default parseFormat was "*%Y-%m-%d*" while from version 2.3 it becomes "*%Y-%m-%d %H:%i*".

It can be altered within current locale as:

```
webix.i18n.parseFormat = "%Y-%m-%d";
webix.i18n.setLocate();
```

Sizing in layout

In Webix 2.2 layout height and width were including padding and margin

```
//in Webix 2.2 will result in 140px height ( 20 + 100 + 20)
//in Webix 2.3 will result in 100px height
webix.ui({
    view:"layout", padding:20, height:100, rows:[{}]
})
```

In webix 2.3 size of layout is a full size, that already includes padding and margin. The same code as above one will result in layout with 100px height. If you preffer the old behavior, you can move height settings in the inner view

```
//in both Webix 2.2 and Webix 2.3 will result in 140px height
webix.ui({
    view:"layout", padding:20, rows:[{ height:100 }]
})
```

Webix 2.1 -> 2.2

Templates in datatable

In version 2.1 you could use "#value#" in the column's template to apply the same property that was defined as the column's id.

Version 2.2 will support this feature as well, but you need to use "#$value#" instead of the old syntax. It does affect only the "#value#" keyword, all other templates ("#some#") will work without any changes.

Ajax

ajax.header method was deprecated in favor of ajax.headers

Sizing

Starting from Webix 2.2, list, datataview and template view, width and height of container will include the width of scrollbar. It means that the resulting view will take exactly the same size that was defined for it (previously, it was the defined as:"size + size of scrollbar").

Webix 1.x -> 2.0

Number formating

"decimalDelimeter" property in number format was changed to "decimalDelimiter"

DHTMLX Touch -> Webix 2.0

Please check Migraton from DHTMLX Touch.

Migration from DHTMLX Touch

If you want to migrate to the Webix library, you should include two additional libraries: **touchui_migration.js** and **touchui_migration.css**, and also you can make the changes described in this article if they are needed:

You can grab those files by the next link

Data Components

(DataStore, Lists, Grid, Dataview, Chart)

The definition of sorting/grouping in the view configuration was changed. You need to use the scheme property and the field name without "#". For complex templates use functions

Old usage

```
dhx.ui({
    sort: "#name#"
});
```

New usage

```
webix.ui({
    scheme:{
        $sort: {
            by:"name"
        }
    }
});
```

Grouplist

"Back" items already contain the "<" icon by default. The **templateBack** property requires defining the text of a "back" item only

Old usage

```
templateBack:"&lt; #title#",
```

New usage

```
templateBack:"#title#",
```

DataProcessor

The onDBerror event is not supported any more. Use the onAfterSaveError event instead

Old usage

```
dp.attachEvent("onDBerror",function(response, state){
 ...
})
```

New usage

```
dp.attachEvent("onAfterSaveError",function(id,status,obj,details){
var state = dp.getItemState(id);
...
});
```

Window and Layout

Layout and related

Full screen layout considers the sizes of inner views and can have smaller sizes than document body. For example, if all rows of layout have fixed height, the layout's height will be calculated like a sum of these rows' heights. So, if you want your layout to occupy the whole page and to consist of rows/cols with fixed height, you should add a row/col without height/width specified.

Old usage

```
dhx.ui({
    rows:[
       {view: "text", ...}
    ]
});
```

New usage

```
webix.ui({
    rows:[
        {view: "text", ...},
        {}
    ]
});
```

Window

Now window is hidden by default. To show it, call the show method.

Old usage

```
dhx.ui({
```

```
    view: "window",...
});
```

New usage

```
webix.ui({
    view: "window",...
}).show();
```

Tabbar, Segmented,Radio, Richselect, Combo

We should use "id" to define options keys instead of "value" and "value" to define titles instead of "label"

Old usage

```
options: [
     {value: 1, label: "Option 1"}, ...
]
```

New usage

```
options: [
     {id: 1, value: "Option 1"}, ...
]
```

Tabbar

1) Tabbar has a new property **tabMinWidth**. It sets minimum width of tabbar buttons, the value is 100 by default. Options that do not fit will be put in a popup. This popup is displayed by clicking the "..." icon.

```
{view:"tabbar", tabMinWidth:90,...}
```

2) Parameters of the onBeforeTabClick and onAfterTabClick events were changed. Now event handlers take two paremeters: id of the clicked option and the native event object

Old usage

```
$$("tabbar").attachEvent("onAfterTabClick",
 function(tabbarId, optionId) {
..
});
```

New usage

```
$$("tabbar").attachEvent("onAfterTabClick",
function(optionId, event) {
..
```

```
});
```

Tabbar, Segmented

Use the "value" property instead of the "selected" one to set a button's value

Old usage

```
{ view: "tabbar" , selected: 2, ...}
```

New usage

```
{ view: "tabbar" , value: 2, ..}
```

Richselect, Combo

1) Use the "suggest" view instead of the "popup" one to define a custom popup in Richselect and Combo

Old usage

```
dhx.ui({
      view: "popup",
      id: "comboOptions",
      body:{ view:"list"}
});
dhx.ui({
      view: "combo",
      popup:"comboOptions"
});
```

New usage

```
webix.ui({
    view: "suggest",
    id: "comboOptions",
    body:{ view:"list",...}
});
webix.ui({
     view: "combo",
     suggest:"comboOptions"
});
```

2) Use **control.getPopup().getList()** instead of **$$(controlId+"_list")** to get list

Old usage

```
var list = $$("myCombo_list")
```

New usage

```
var list = $$("myCombo").getPopup().getList();
```

Form and Toolbar

1) The method setValues sets values to elements by their "name" properties. The getValues method returns hash of values by elements' names. The previous version accepted "id" as well as "name". Now ids are ignored.

Old usage

```
dhx.ui({
    view: "form",
    id: "myform",
    elements:[
        { view: "text", name: "valueA", value: 1},
        { view: "text", id: "valueB", value: 2}
    ]
});


$$("myForm").getValues();
//same returns {valueA: 1, valueB:2}
```

New usage

```
webix.ui({
    view: "form",
    id: "myform",
    elements:[
        { view: "text", name: "valueA", value: 1},
        { view: "text", id: "valueB", value: 2}
    ]
});

$$("myForm").getValues(); // returns {valueA: 1}
```

2) If you set an event handler (for example, onItemClick, onBeforeTabClick) for controls to their parent Toolbar or Form, the handler will be called only for those elements that have the "name" property.

Old usage

```
dhx.ui({
    view: "toolbar",
    id: "myToolbar",
    elements:[
       { view: "button", id: "btn1", ...},
       { view: "button", id: "btn2", ...},
       ...
```

```
    ]
});

$$("myToolbar").attachEvent("onItemClick", function(id){
       alert("Clicked item is "+id);
});
```

New usage

```
webix.ui({
    view: "toolbar",
    id: "myToolbar",
    elements:[
       { view: "button", name: "btn1", ...},
       { view: "button", name: "btn2", ...},
       ...
    ]
});

$$("myToolbar").attachEvent("onItemClick", function(name){
       alert("Clicked item is "+name);
});
```

Toolbar

The "type" property is not supported

Old usage

```
{ view: "toolbar", type: "SubBar" ...}
```

New usage

```
{ view: "toolbar", ...}
```

Slider

Options are not supported. Use the **min**, **step** and **max** properties to define the range

Old usage

```
{ view:"slider", ...,
   options:[
   {value: 1, label: 1},
   {value: 2, label: 2},
   ....
]}
```

New usage

```
{ view:"slider", min: 1, step: 1, max: 10}
```

Autogenerated ids

If you did not set the id for a certain view, it will get the id in the following format:

"$"+ViewName+ViewIndex

Old usage

```
// we create a list without id definition
dhx.ui({view: "list",....});
// the id is set automatically - list1 (list2,...)
$$("list1").select(id);
```

New usage

```
// we create a list without id definition
webix.ui({view: "list",....});
// the id is set automatically - $list1 ($list2,...)
$$("$list1").select(id);
```

webix.message()

(dhx.notice() in old version)

Use the "type" property instead of the "css" one to customize style of the message container

Old usage

```
dhx.notice(css: "error",...)
```

New usage

```
webix.message(type: "error",...)
```

Controls

Toggle

Toggle now looks and behaves like a two-state button. It is one button that changes its style and label on click. If you want to use a control that is similar to the toggle used in the previous version, you should choose the segmented button instead.

Old usage

```
{ view:"toggle", id:'trip', options: ["Roundtrip","Oneway"] }
```

New usage

```
{ view:"segmented", name:'trip', options: ["Roundtrip","Oneway"]}
```

Round button

Button with the type "round" is not supported any more

Old usage

```
{view:"button",type: "round",...}
```

New usage

```
{view:"button",...}
```

History

We need to use the track(id) method instead of the start() one and pass the id of the tracked element (id of tabbar for example)

Old usage

```
dhx.history.start()
```

New usage

```
webix.history.track('tabbarId')
```

webix.proxy

The new version requires a new proxy definition

Old usage

```
var proxy= new dhx.proxy({
    url: "data.php",
    storage: dhx.storage.local
});
```

New usage

```
```

```
var proxy = new webix.proxy( "local", "data.php" );
```

webix.Date

Some Date methods do not create a copy by default. If the copy parameter is not set to true, the passed Date object will be modified.

Old usage

```
var next = dhx.Date.add(d,1,"day");
```

New usage

```
var next = webix.Date.add(d,1,"day",true);
```

Additional Migration Changes

In this article you'll find the migration issues that are solved by including the **touchui_migration.js** file.

Data Components

(DataStore, Lists, Grid, Dataview, Chart)

Some datastore methods and methods of views that based on DataStore (list,grid,dataview and others) were renamed:

Old usage	New usage
var count = list.dataCount();	var count = list.count();
var item = list.item(id);	var item = list.getItem(id);
list.update(id,data);	list.updateItem(id,data);
var index = list.indexById(id);	var index = list.getIndexById(id);
var id = list.idByIndex(index);	var id = list.getIdById(index);
var nextId = list.next(id,step);	var nextId = list.getNextId(id,step);
var firstId = list.first();	var firstId = list.getFirstId();
var lastid = list.last();	var lastId = list.getLastId();
var prevId = list.prev(id,step);	var prevId = list.getPrevId(id,step);
var id = list.getSelected();	var id = list.getSelectedId();

Dataview

Selection is not enabled by default

Old usage

```
{view: "dataview", ...}
```

New usage

```
{view: "dataview", select: true,...}
```

Grouplist

In case of **json datasource** you should use the **item** property in it to define child collection (there was "data" before)

Old usage

```
//JSON data source:
{ id:"root", value:"top item", data:[ ... ]}
```

New usage

```
//JSON data source:
{ id:"root", value:"top item", item:[ ... ]}
```

Pagelist

Use the getActiveId() method instead of the getActive() one to get the id of the first visible item

Old usage

```
var id = $$("list").getActive();
```

New usage

```
var id = $$("list").getActiveId();
```

DataProcessor

If you are using connector on the server side, you should start definition of the url property of DataProcessor with "connector->"

Old usage

```
var dp = new dhx.DataProcessor({
     url:"update.php",
     ...
});
```

New usage

```
var dp = new webix.DataProcessor({
     url:"connector->update.php",
     ..
});
```

Views of Layout and Window

Use the getParentView() method instead of the getParent() one

Old usage

```
var parentView =  view.getParent()
```

New usage

```
var parentView = view.getParentView()
```

Window

By default window is hidden and doesn't have fixed size. Its size is adjusted to the size of its content. Window view appears in the **left:0, top:0** position.

Old usage

Library sets by default for this view: **top:300, left:100, width:300, height:200**

New usage

Library sets by default for this view: **top:0, left:0, autofit:true**

Scrollview

Use the "body" property instead of the "content" one to define the inner view of Scrollview

Old usage

```
{
    view: "scrolview",...,
    content:{ view: "template",...}
}
```

New usage

```
{
    view: "scrolview",...,
    body:{ view: "template",...}
}
```

Tabbar

The default type of tabbar was changed. You should set the type:"bottom" if you want to show a tabbar below a multiview

Old usage

```
{ view:"tabbar", ...}
```

New usage

```
```

```
{ view:"tabbar", type: "bottom",...}
```

Multiview

Use the getActiveId() method instead of the getActive() one to get the id of an active view

Old usage

```
var id = $$("views").getActive();
```

New usage

```
var id = $$("views").getActiveId();
```

Datepicker and other cases

1) The default date format is set by the **dhx.i18n.parseFormat**. It is **"%Y-%m-%d"** by default.

Old usage

The default date format "%d.%m.%Y".

New usage

The default date format is "%Y-%m-%d".

2) Use the "value" property instead of the "date" one to define the control's value

Old usage

```
{ view:"datepicker", date:new Date(2014,7,1)}
```

New usage

```
{ view:"datepicker", value:new Date(2014,7,1)}
```

Message Boxes

notice

Use **webix.message** instead of **dhx.notice**

Old usage

```
dhx.notice(text)
```

New usage

```
webix.message(text)
```

webix.alert, webix.confirm, webix.message

Use the **text** property instead of the **message** one to define the popup text

Old usage

```
dhx.alert({message:'Incorrect password!"})
```

New usage

```
webix.alert({text:'Incorrect password!"})
```

Controls

Imagebutton

Use the view "button" with the type "image"

Old usage

```
dhx.ui({ view: "imagebutton", ...});
```

New usage

```
webix.ui({ view: "button", type: "image", ...});
```

Button

By default, a button's alignment is "left" ("center" in the previous version)

Old usage

```
dhx.ui({view:"button",inputWidth:100,..})
```

New usage

```
webix.ui({
view:"button",inputWidth:100, align:"center",...})
```

webix.KeyEvents

This module does not exist any more. The onTimedKeyPress event listening is enabled by default.

Old usage

```
dhx.extend($$('filter'),dhx.KeyEvents);
```