

## Widgets v2 Instance API

---

The functions in this API can be used for the instance of a widget that you created using the [Widgets v2 Class API](#) either at runtime (execution of the code) or in your custom widget's `widget.js` file.

All the functions can be used either at runtime or in your custom widget's code as shown below:

### At Runtime:

```
$$('customWidget1').addClass('mycssclass');
```

### Code for your widget:

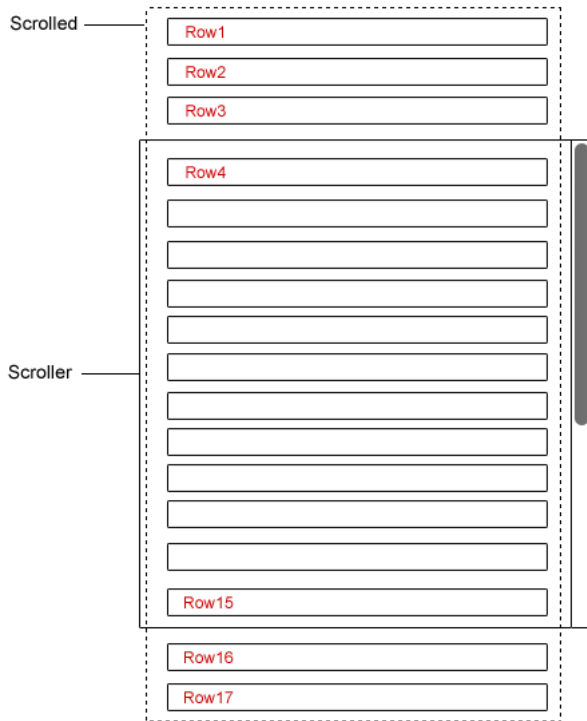
```
CustomWidget.prototype.init = function () {  
    this.addClass('mycssclass');  
};
```

## Repeater Live Scroll

You can enable the repeater live scroll of data for your custom widget by including the "waf-behavior/layout/repeater-livescroll" library into your "widget.js" file. This behavior uses the following terminology:

- **Scroller:** the DOM node that encompasses the scrollable area
- **Scrolled:** the scrollable area
- **Row:** the element in the scrollable area displayed for each entity

Graphically the live scroll appears as shown below:



### CSS definition

For the scroller to function properly, you must add the following three items to your custom widget's CSS file. In our example, the main widget is named "MainScroll" and the row widget is named "MainScrollRow".

The following code allows the scroller to display the scrollbars:

```
.waf-widget.waf-mainscroll {
  overflow-x: hidden;
  overflow-y: scroll;
  -webkit-overflow-scrolling: touch;
}
```

The scrolled portion of the custom widget must have its position be relative instead of absolute.

```
.waf-widget.waf-mainscroll ul {
  position: relative;
}
```

The row must also have a relative position:

```
.waf-widget.waf-mainscroll .waf-mainscrollrow {
  position: absolute;
}
```

### Mandatory callbacks

You must define the following properties that are retrieved by these functions in this behavior:

- **getRowHeight:** Height of a row in pixels that must also be defined in the custom widget's CSS file.
- **getScrolledNode:** The DOM node for the scrolled node.

For more information about the page size (the `pageSize` property), you can use refer to `{propertyName}.pageSize()` property for the property of type `Datasource`.

## Example

Here is an example of how the widget using this behavior must be constructed:

```
WAF.define('LiveScrollTest', ['waf-core/widget', 'Text'], function(widget, Text) {
    "use strict";

    var LiveScrollTest = widget.create('LiveScrollTest', {
        rows: widget.property({
            type: 'datasource',
            pageSize: 10,
            attributes: ['title', 'content']
        }),
        getScrolledNode: function() {
            if(!this.node.getElementsByTagName('ul')[0]) {
                this.node.innerHTML = '<ul></ul>';
            }
            return this.node.getElementsByTagName('ul')[0];
        },
        getRowHeight: function() {
            return 60;
        },
        init: function() {
            // initialization code
        }
    });
    LiveScrollTest.inherit('waf-behavior/layout/repeater-livescroll');
    LiveScrollTest.linkDatasourcePropertyToRepeater('rows');
    LiveScrollTest.repeatedWidget(Text);
    LiveScrollTest.mapAttributesToRepeatedWidgetProperties({
        title: 'value',
        content: 'content'
    });
});
```

*Note: The `pageSize` property must be defined for the property of type `Datasource` and should be equal to or greater than the first page of data.*

---

## getRowSize()

Number `getRowSize()`

Returns Number Row size in pixels

## Description

`getRowSize()` allows you to return the size of a row in pixels depending on if the rows are scrolling horizontally or vertically. For example, you can define the `getRowSize` property for your custom widget as shown below:

```
getRowSize: function() {
    return 40;
}
```

In the CSS file, you can define it in this way:

```
.waf-widget.waf-livescrolltest .waf-livescrolltestrow {
    position: absolute;
    width: 100%;
    height: 40px; /* for a vertical scroll the size (height) of the row is 40px*/
    overflow: hidden;
}
```

---

## getScrolledNode()

HTMLElement `getScrolledNode()`

Returns HTMLElement Scrolled DOM node

## Description

`getScrolledNode()` allows you to return the scrolled DOM node.

## Example

If you call the following line of code:

```
var scrolledNode = $$('liveScrollTest1').getScrolledNode();
```

In our case, the `scrolledNode` variable returns the following:

```
<ul style="height: 480px;">
```



## Composed

---

The following functions are in the `waf-behavior/layout/composed` behavior.

### `getPart()`

---

Widget <code>getPart</code> ( partName )	Type	Description
Parameter partName	String	Part name
Returns	Widget	Widget for partName

#### Description

`getPart()` allows you to get the widget for a part.

#### Example

The following example retrieves the widget that is passed as the part to get for the composed widget:

```
var myPart = $$('composedWidget1').getPart('part1');
```

### `getParts()`

---

Array <code>getParts()</code>	Type	Description
Returns	Array	Parts for the widget

#### Description

`getParts()` returns the parts of the widget in an array.

#### Example

For example, if you write the following:

```
var myParts = $$("composedWidget1").getParts();
```

The `myParts` array contains the names of all the composed widget's parts.

### `removePart()`

---

void <code>removePart</code> ( part )	Type	Description
Parameter part	String	Part name

#### Description

`removePart()` allows you to remove a part. This part will not be destroyed completely, so you could add it back to the composed widget using the `setPart()` function.

#### Example

In this example, we remove one of the parts:

```
$$('composedWidget1').removePart('part1');
```

### `setPart()`

---

void <code>setPart</code> ( String part [, Widget widget] )	Type	Description
Parameter part widget	String Widget	Part name Widget to set

#### Description

`setPart()` allows you to set the part for a widget by defining the part's name and an actual widget. If you do not specify a *widget* to set, the *part* will be removed.

#### Example

In the following example, we set a part defined by a name and an existing widget:

```
$$('composedWidget1').setPart('part1', $$('partWidget1'));
```

## Container

---

The following functions are in the `waf-behavior/layout/container` behavior.

You can create a widget and use this behavior for it by writing the following:

```
WAF.define('CustomWidget', ['waf-core/widget'], function(widget) {
  var CustomWidget = widget.create('CustomWidget', {
    init: function() { }
  });
  CustomWidget.inherit('waf-behavior/layout/container');

  return CustomWidget;
});
```

## detachWidget

---

### Description

`detachWidget` is an event that is triggered each time a widget is detached from a Container. You can detach a widget by using the `detachWidget()`, `detachAllWidgets()`, and `detachAndDestroyAllWidgets()` functions. You can also detach a widget from a Container by selecting a widget and dragging it out of the Container.

This event is also triggered when you use the `widget()` function to replace an attached widget in the Container.

In the `event.data` object, you can retrieve the following two attributes:

Attribute	Description
<code>widget</code>	The widget that was detached from the Container.
<code>index</code>	Index number of the detached widget.

### Example

Below is an example of how to intercept this event for the Container widget:

```
var MyContainer = widget.create('MyContainer', {
  init: function() {
    this.subscribe('detachWidget', function(event) {
      //do something here with the event.data object
    });
  }
});
```

## moveWidget

---

### Description

`moveWidget` is an event that is triggered when a widget is moved from one position to another in the index list of the Container's attached widgets using the `moveWidget()` function.

In the `event.data` object, you can retrieve the following attributes:

Attribute	Description
<code>widget</code>	The attached widget.
<code>from</code>	Previous index number of the attached widget.
<code>to</code>	New index number of the attached widget.

The `from` and `to` attributes are the values you passed as parameters to the `moveWidget()` function.

### Example

Below is an example of how to intercept this event for a Container widget:

```
var MyContainer = widget.create('MyContainer', {
  init: function() {
    this.subscribe('moveWidget', function(event) {
      //do something here with the event.data object
    });
  }
});
```

## insertWidget

---

### Description

`insertWidget` is an event that is triggered each time a widget is attached to a Container using either `insertWidget()` or `attachWidget()`. You can also attach a widget in the GUI Designer by dropping it into the Container.

This event also occurs if you replace an existing attached widget using the `widget()` function.

In the `event.data` object, you can retrieve the following two attributes:

Attribute	Description
-----------	-------------

Attribute	Description
widget	The widget that was attached from the Container.
index	Index number of the attached widget.

### Example

Below is an example of how to intercept this event for the Container widget:

```
var MyContainer = widget.create('MyContainer', {
  init: function() {
    this.subscribe('insertWidget',function(event) {
      //do something here with the event.data object
    });
  }
});
```

### attachWidget( )

Number <b>attachWidget</b> ( Widget <i>widget</i> )		
Parameter	Type	Description
widget	Widget	Widget to attach to the container
Returns	Number	Index of the widget attached

### Description

**attachWidget( )** allows you to attach a widget to a Container. The widget will be added to the end of the Container's list and its index number will be returned after you call this function.

### Example

This example allows you to attach a widget to the Container widget:

```
var widgetIndex = $$("container1").attachWidget($$('customWidget1'));
```

### countWidgets( )

Number <b>countWidgets</b> ( )		
Returns	Number	Number of widgets attached to the container

### Description

**countWidgets( )** allows you to returns the number of widgets attached to the container.

### Example

In the following example, where there are four widgets and two label widgets:

A call to **countWidgets( )**, returns 6:

```
var attWidgets = $$("container1").countWidgets();
```

### detachAllWidgets( )

void **detachAllWidgets**( )

### Description

**detachAllWidgets( )** allows you to detach all the widgets attached to the container. The **detachWidget** event is executed for each detached widget. The widgets are detached from the Container, but still exist. To delete them as well, you can use the **detachAndDestroyAllWidgets( )** function.

### Example

This example detaches all the widgets previously attached to a Container:

```
$$("myContainer1").detachAllWidgets();
```

### detachAndDestroyAllWidgets( )

```
void detachAndDestroyAllWindows()
```

## Description

`detachAndDestroyAllWindows()` allows you to detach all the widgets attached to the container and destroy them. The `detachWidget` event is executed for each detached widget.

If you want to just detach the widgets from the Container widget, you can use the `detachAllWindows()` function.

## Example

This example detaches and destroys all the widgets previously attached to a Container:

```
$$('myContainer1').detachAndDestroyAllWindows();
```

## detachWidget()

---

Widget `detachWidget`( Number *index* )

Parameter	Type	Description
index	Number	Index of the widget to detach
Returns	Widget	Widget detached from Container

## Description

`detachWidget()` allows you to detach a widget from the container by passing either its *index*, which is the widget's index number defined by the `widgets()` function, or the widget itself. In both cases, the actual widget is returned and the `detachWidget` event is executed.

If you want to detach all the widgets, you can use the `detachAllWindows()` function.

## Example

In the following example, we detach a specific widget.

Either by passing the widget itself:

```
var detachedWidget = $$('myContainer1').detachWidget($$('customWidget1'));
```

Or by passing the widget's index number in the Container:

```
var detachedWidget = $$('myContainer1').detachWidget(2); //it's the third widget attached to the myContainer1 widget
```

## indexOfWidget()

---

Number `indexOfWidget`( Widget *widget* )

Parameter	Type	Description
widget	Widget	Widget attached to container
Returns	Number	Widget index or -1 if widget was not found

## Description

The `indexOfWidget()` function returns the attached widget's index number in the Container or -1 if the widget is not attached to the Container.

You can insert a widget into the Container or attach it to the Container by using `attachWidget()` or `insertWidget()`. In the GUI Designer, you attach a widget to the Container by dropping it inside of the Container.

## Example

In the following example, we retrieve the index of the widget inside the container:

```
var indexButton = $$('container1').indexOfWidget($$('button3'));
```

## insertWidget()

---

Number `insertWidget`( Number *index*, Widget *widget* )

Parameter	Type	Description
index	Number	Index number where to insert the widget in the container
widget	Widget	Widget to insert in the container
Returns	Number	Index number of the widget inserted in the container

## Description

`insertWidget()` allows you to insert a widget in the container in the *index* position.

## Example

The following example attaches a widget at *index* to the Container widget:

```
var attachedWidget = $$('myContainer1').insertWidget(2,$$('customWidget1'));
```

## invoke()

---

Array `invoke`( String *functionName* [, String *argument*,..., String *argumentN*])



Parameter	Type	Description
functionName argument	String String	Name of the function Arguments for functionName
Returns	Array	Results for the function invoked for each widget it was applied to

## Description

`invoke()` allows you to call a function on all the container's attached widgets. *functionName* returns an array with the results. *functionName* is applied to all the widgets that are returned by `widgets()` and not the actual widget for which you're calling it.

## Example

Our example applies the style function on the widgets inside of *the container1* widget (which are *customWidget2* and *customWidget3*):

```
var resultsInvoke = $$('container1').invoke('style', 'background-color', 'red')
```



Only widgets *customWidget2* and *customWidget3* have a red background. The result of the `invoke()` function is:

```
["red", "red"]
```

## lastWidget()

---

Widget `lastWidget()`

Returns	Widget	Last widget attached to the container
---------	--------	---------------------------------------

## Description

`lastWidget()` allows you to return the last widget attached to the Container either by dropping it into the Container or by adding it using either the `insertWidget()` or `attachWidget()` functions. The last widget attached can also occur when you replace an existing widget by using the `widget()` function.

The widget must still be attached to the Container when you call this function. Otherwise, an error will be returned.

## Example

In our example, we test for the last widget attached to the Container:

```
var MyContainer = widget.create('MyContainer', {
  init: function() {
    this.subscribe('insertWidget', function(event) {
      var thisLast = this.lastWidget();
      // do something with thisLast (which is the last widget attached to the Container)
    }, this);
  }
});
```

*Note: The `event.data.widget` attribute also returns the last attached widget.*

## moveWidget()

---

void `moveWidget( Number element, Number newPosition )`

Parameter	Type	Description
element	Number	Element to move
newPosition	Number	New position for element

## Description

`moveWidget()` allows you to move *element* to *newPosition*. If there is a widget in the *newPosition*, the other widgets are moved accordingly.

The `moveWidget` event is triggered after you call this function.

## Example

In the following example, we first obtain the widgets in a container widget:

```
var myWidgets = $$('container1').widgets(); // returns [$$('widget1'), $$('widget2'), $$('widget3'), $$('widget4')]
```

If we call the following code:

```
$$('container1').moveWidget(0,2);
```

If we call this line of code again, the results are different:

```
var myWidgets = $$('container1').widgets(); // returns [$$('widget2'), $$('widget3'), $$('widget1'), $$('widget4')]
```

---

## restrictWidget()

```
void restrictWidget(Widget widgetClass)
```

Parameter	Type	Description
widgetClass	Widget	Widget class

### Description

**restrictWidget()** allows you to restrict the widget class (v2 only) that can be attached to the container. Once you define the widget class to restrict, no other widget classes can be attached to the Container using either **insertWidget()** or **attachWidget()**, or by dropping it into the Container in the GUI Designer. The widget class you specify also applies to widgets that inherit that widget class.

*Note: You must also include the widgets you restrict in the as shown in the example for the **restrictWidget()** class function.*

---

## widget()

```
Widget widget( Number index [, Widget widgetToReplace] )
```

Parameter	Type	Description
index	Number	Index of the widget
widgetToReplace	Widget	If specified, the widget replaces the widget at index
Returns	Widget	Widget at the index position or widgetToReplace

### Description

**widget()** allows you to either retrieve the widget at the *index* position or replace it with the widget passed as the second parameter.

The widget that was in the *index* position is detached and destroyed by *widgetToReplace*, if passed. The **detachWidget** and **insertWidget** events occur if you replace one widget with another. The first event occurs for the widget detached from the Container and the second event is triggered for the widget attached to the Container.

### Example

In the example below, we replace the widget at the *index* position with another widget:

```
var newWidget = $$('container1').widget(1,$$('text1'));
```

### Example

In this example, we retrieve the widget at the first position in the Container where *index* is equal to 0:

```
var firstWidget = $$('container1').widget(0); //get first widget
```

To get the last widget in the index list attached to the Container:

```
var count = $$('container1').countWidgets();  
var lastWidget = $$('container1').widget(count-1); //get last widget
```

---

## widgets()

```
Array widgets()
```

Returns	Array	Array of the container's attached widgets
---------	-------	---

### Description

With the **widgets()** function, you retrieve an array of all the attached widgets. Each widget has its own *index* number that can be used by other functions.

*Note: For all v1 widgets, the "kind" is "OldWidget" instead of the widget's class name.*

### Example

In the following example, we first obtain the widgets in a container widget:

```
var myWidgets = $$('container1').widgets(); // returns [$$('widget1'), $$('widget2'), $$('widget3'), $$('widget4')]
```

If we call the following code:

```
$$('container1').moveWidget(0,2);
```

If we call this line of code again, the results are different:

```
var myWidgets = $$('container1').widgets(); // returns [$$('widget2'), $$('widget3'), $$('widget1'), $$('widget4')]
```

## MultiContainer

---

The following functions are in the `waf-behavior/layout/multicontainer` behavior.

You can create a widget and use this behavior for it by writing the following:

```
WAF.define('CustomWidget', ['waf-core/widget'], function(widget) {
  var CustomWidget = widget.create('CustomWidget', {
    init: function() { }
  });
  CustomWidget.inherit('waf-behavior/layout/multicontainer');

  return CustomWidget;
});
```

### Applying other functions

You can also apply the **Container** functions to a container in the MultiContainer widget, like `attachWidget()` and `widgets()`.

If you have a composed widget in the Container, you can use `children()` to retrieve all the widgets. Otherwise, you can use `widgets()` to retrieve just the main widgets.

For example, you can write the following to get all the widgets in the first container:

```
var allContainerWidgets = $$('customWidget').container(0).widgets();
```

## addContainer()

---

Number `addContainer()` ([Object *options*])

Parameter	Type	Description
<code>options</code>	Object	Options for the Container class to apply to the new container
Returns	Number	Index number of the container added

### Description

`addContainer()` allows you to add a new container to your multi-container widget. You can specify any *options* that you defined for your custom widget.

For more information about the *options*, refer to [options](#).

### Example

In this example, we create a new container:

```
var indexNewContainer = $$("customWidget1").addContainer({color:"red", title: "My Title"});
```

## container()

---

Widget `container()` (Number *index*)

Parameter	Type	Description
<code>index</code>	Number	Index number of the container
Returns	Widget	Container widget defined by index

### Description

`container()` allows you to get the container widget specified by *index*.

You can apply the **Container** functions to a container in the MultiContainer widget, like `attachWidget()` and `widgets()`.

If you have a composed widget in the Container, you can use `children()` to retrieve all the widgets. Otherwise, you can use `widgets()` to retrieve just the main widgets.

### Example

For example, you can write the following to get all the widgets in the first container:

```
var allFirstContainerWidgets = $$('customWidget').container(0).widgets();
```

## containers()

---

Array `containers()`

Returns	Array	All the container widgets
---------	-------	---------------------------

### Description

`containers()` allows you to retrieve all the containers in the multi-container widget.

### Example

In this example, retrieve all the container widgets for your multi-container widget:

```
var allContainers = $$('customWidget').containers();
```

## countContainers( )

---

Number **countContainers()**

Returns Number Total number of containers

### Description

**countContainers( )** allows you to retrieve the number of containers in your multi-container widget.

## currentContainer( )

---

Widget **currentContainer()**

Returns Widget Current container widget

### Description

**currentContainer( )** returns the current Container widget.

## currentContainerIndex( )

---

Number **currentContainerIndex( [Number index] )**

Parameter	Type	Description
index	Number	Select the container specified by its index
Returns	Number	Currently selected container's index

### Description

**currentContainerIndex( )** allows you to select the current container and retrieve the index for the currently selected container. The "waf-state-active" CSS class is assigned to the current container.

## defaultContainerClass( )

---

String **defaultContainerClass( [String containerClass] )**

Parameter	Type	Description
containerClass	String	Default Container class
Returns	String	Default Container class

### Description

**defaultContainerClass( )** allows you to get or set the default Container's class when creating new containers.

### Example

In the following example, when a container is added to the widget, its class will be "MyContainerClass":

```
init: function() {  
    this.defaultContainerClass(MyContainerClass);  
}
```

## insertContainer( )

---

Number **insertContainer( Number index [, Object options] )**

Parameter	Type	Description
index	Number	Index number where the new Container is inserted
options	Object	Options for the Container class to apply to the new container
Returns	Number	Index number of the container inserted

### Description

**insertContainer( )** allows you to add a new container at *index* to your multi-container widget. You can specify any *options* that you defined for your custom widget.

For more information about the *options*, refer to [options](#).

### Example

In this example, we insert a new container at the third position:

```
var indexNewContainer = $$("customWidget1").insertContainer(2, {color:"red", title: "My Title"});
```

## lastContainer( )

---

Widget **lastContainer()**

Returns Widget The last widget container added/inserted

## Description

`lastContainer()` allows you to retrieve the last container *widget* added or inserted into the multi-container widget.

## Example

In this example, we insert a new container at the third position:

```
var indexNewContainer = $$("customWidget1").insertContainer(2, {color:"red", title: "My Title"});
```

Then, we call `lastContainer()` to retrieve it:

```
var lastContainer = $$("customWidget1").lastContainer();
```

---

## `moveContainer()`

```
void moveContainer( Number position, Number newPosition )
```

Parameter	Type	Description
position	Number	Position of the container to move
newPosition	Number	New position of the container

## Description

`moveContainer()` allows you to move the Container in *position* to *newPosition*. If there is a container in the *newPosition*, the other containers are moved accordingly.

## Example

If we call the following code:

```
$$('customWidget1').moveContainer(0,2);
```

The first container will be moved to the third position.

---

## `removeAllContainers()`

```
void removeAllContainers()
```

## Description

`removeAllContainers()` allows you to remove all the containers in the multi-container widget. All the widgets inside each container are also removed.

---

## `removeContainer()`

```
void removeContainer( Number index )
```

Parameter	Type	Description
index	Number	Index number of the container to remove

## Description

`removeContainer()` allows you to remove a specific container that you specify by its *index*. All the widgets inside the container are also removed.

---

## `setLastContainerAsCurrent()`

```
void setLastContainerAsCurrent()
```

## Description

`setLastContainerAsCurrent()` allows you to set the last container as the current one.

## Example

In this example, we create a new container and then select it immediately afterwards:

```
var indexNewContainer = $$("customWidget1").addContainer({color:"green"});  
$$("customWidget1").setLastContainerAsCurrent();
```

## Observable

---

These functions allow you to define or execute an event as well as subscribe to and unsubscribe from an event.

### fire()

---

```
void fire( String event [,String target] [,Object data] [,Object options] )
```

Parameter	Type	Description
event	String	Event name
target	String	Target expressed in RegEx
data	Object	Optional data
options	Object	Event options

#### Description

`fire()` allows you to declare or fire an event.

#### options

The `options` parameter contains the following attributes:

Attribute	Description
deferred	timeout to launch the callback
animationFrame	launch the callback during an animation frame
once	discard previous pending events if true
onlyRealEvent	launch callback only when the fired event is the subscribed event

#### Example

In the following example, we create a custom event attached to the standard DOM "click" event. In the widget.js file, we write the following:

```
CustomWidget.prototype.init = function() {
    this.fire('myEvent');
};
CustomWidget.mapDomEvents( { 'click': 'myEvent' } );
```

You can customize how the event appears in the GUI Designer even though it's not necessary, by writing:

```
CustomWidget.addEvent({
    'name': 'myEvent',
    'description': 'On My Event',
    'category': 'My Custom Events'
});
```

When you click on your custom widget, the "myEvent" event will be launched.

### removeSubscriber()

---

```
void removeSubscriber( Object subscriber )
```

Parameter	Type	Description
subscriber	Object	Subscriber object

#### Description

With `removeSubscriber()`, you can remove a subscriber that was created with `subscribe()`.

#### Example

If you subscribe to an event, as shown below:

```
var mySubscriber = $$('customWidget1').subscribe('click', function() {
    //do something
});
```

You can remove the subscriber from it in the following manner:

```
$$('customWidget1').removeSubscriber(mySubscriber);
```

### subscribe()

---

```
Object subscribe( String event[, String target], Function callback[, String observer][, Object userData] )
```

Parameter	Type	Description
event	String	Event to which to subscribe
target	String	Target filtering for the event
callback	Function	Callback after the event is fired
observer	String	Object subscribing to the event
userData	Object	Data passed to the callback
Returns	Object	Subscriber object

#### Description

`subscribe()` allows you to subscribe to an event and execute a *callback* asynchronously. The event must have already been defined for the custom widget by using `mapDomEvents()`.

#### Example

If you subscribe to an event, as shown below:

```
var mySubscriber = $$('customWidget1').subscribe('click', function() {
  //do something
}, this);
```

**Note:** The *this* variable added as a parameter to the `subscribe()` function means that *this* will be the custom widget and not the subscriber.

You can unsubscribe from it in the following manner:

```
$$('customWidget1').unsubscribe({
  event: 'click'
});
```

#### unsubscribe()

---

void `unsubscribe`(Object *subscriber*)

Parameter	Type	Description
<code>subscriber</code>	Object	Object defining the event, target, callback, and observer

#### Description

`unsubscribe()` allows you to unsubscribe a subscriber from an event.

#### Example

If you subscribe to an event, as shown below:

```
var mySubscriber = $$('customWidget1').subscribe('click', function() {
  //do something
}, this);
```

**Note:** The *this* variable added as a parameter to the `subscribe()` function means that *this* will be the custom widget and not the subscriber.

You can unsubscribe from it in the following manner:

```
$$('customWidget1').unsubscribe({
  event: 'click'
});
```

## Pagination

---

You can enable the pagination of data for your custom widget by including the "waf-behavior/navigationSource" library into your "widget.js" file.

You include this library with your custom widget, you can write the following:

```
WAF.define('CustomWidget', ['waf-core/widget'], function(widget) {
  var sourceNavigation = WAF.require('waf-behavior/navigationSource');
  var CustomWidget = widget.create('CustomWidget', {
    //place the rest of the code for your widget here
  });
  CustomWidget.inherit(sourceNavigation);
  return CustomWidget;
});
```

Otherwise, you can include this information directly in the `define()` function when you include the library:

```
WAF.define('CustomWidget', ['waf-core/widget', 'waf-behavior/source-navigation'], function(widget, sourceNavigation) {
  var CustomWidget = widget.create('CustomWidget', {
    //place the rest of the code for your widget here
  });
  CustomWidget.inherit(sourceNavigation);
  return CustomWidget;
});
```

You can define the default `pageSize` in the `pageSize` property when declaring the property of type `datasource`.

### Properties

You can define the following properties, which are also defined as functions below:

- **start**: The number of the element displayed at the top of the page.
- **currentPage**: The current page of the paginated data displayed.
- **navigationMode**: The navigation mode that is either 'loadmore' or 'pagination'.
- **totalPages**: Total number of pages of the paginated data.
- **pageSize**: Number of elements currently displayed on the page.

### currentPage()

---

void **currentPage**(Number *page*)

Parameter	Type	Description
page	Number	Page number of the page to display

### Description

`currentPage()` allows you to either set the page of the paginated data to display or get the number of the currently displayed page.

### linkDatasourcePropertyToNavigation()

---

void **linkDatasourcePropertyToNavigation**(String *datasourceProperty*)

Parameter	Type	Description
datasourceProperty	String	Custom widget's property of type <code>Datasource</code>

### Description

With `linkDatasourcePropertyToNavigation()`, you define the custom widget's property of type `Datasource`, which contains the data to be paginated.

### Example

In the example below, we create the property of type `Datasource` and assign it to our custom widget so that we can page the data.

```
var CustomWidget = widget.create('CustomWidget', {
  dsProperty: widget.property({
    type: 'datasource'
  }),
  init: function() {
    this.linkDatasourcePropertyToNavigation('dsProperty');
  }
});
```

### linkParentElementToNavigation()

---

void **linkParentElementToNavigation**(Object *domElement*)

Parameter	Type	Description
domElement	Object	Parent DOM element

### Description

`linkParentElementToNavigation()` allows you to define the parent DOM event in which to return the data (which will be in a separate DOM node for each



element).

## Example

The following example defines the parent DOM element to be a <ul>:

```
init: function() {
  this.linkDataSourcePropertyToNavigation('dsProperty');
  var ulElement = document.createElement('ul');
  this.node.appendChild(ulElement);
  this.linkParentElementToNavigation(ulElement);
}
```

---

## loadMore( )

void **loadMore**( )

### Description

**loadMore( )** allows you to load another page of data as defined by the *pageSize* property.

---

## navigationMode( )

void **navigationMode**( String *mode* )

Parameter	Type	Description
mode	String	Paging mode: "pagination" or "loadmore"

### Description

**navigationMode( )** allows you to either get or set the pagination mode (either "pagination" or "loadmore"). The two modes are:

- **pagination**: Change pages with the data.
- **loadmore**: Allows you to load more data that is appended to the data already displayed on the page.

---

## nextPage( )

void **nextPage**( )

### Description

**nextPage( )** allows you to retrieve the next page of the paginated data. The number of entities displayed depends on what is currently being displayed.

---

## pageSize( )

void **pageSize**( Number *size* )

Parameter	Type	Description
size	Number	Number of elements to display on the page

### Description

**pageSize( )** allows you to set the number of elements to display on the page or get the number of elements currently displayed on the page.

---

## prevPage( )

void **prevPage**( )

### Description

**prevPage( )** allows you to retrieve the previous page of the paginated data. The number of entities displayed depends on what is currently being displayed.

---

## renderElement( )

Object **renderElement**( )

Returns	Object	Each entity in the paginated data
---------	--------	-----------------------------------

### Description

**renderElement( )** allows you to define a function in which the current entity is returned. The parameters received by this function are explained below.

### Parameters

In the function you create for **renderElement( )**, you receive two parameters:

Parameter	Description
element	Datasource entity. You can use the <b>getAttributeValue( )</b> function to access data from a specific attribute in the datasource element.

position Position of the entity in the entity collection.

---

### Example

In the following example, the `renderElement()` function returns one of the attributes, in our case "name", from the current entity in the `<li>` DOM element after we defined the parent DOM element with `linkParentElementToNavigation()`:

```
var CustomWidget = widget.create('CustomWidget', {  
  
  dsProperty: widget.property({  
    type: 'datasource',  
    pageSize: Infinity  
  } ),  
  
  init: function() {  
    this.buildContainer();  
    this.linkDatasourcePropertyToNavigation('dsProperty');  
  },  
  
  buildContainer: function() {  
    var ulElement = document.createElement('ul');  
    this.node.appendChild(ulElement);  
    this.linkParentElementToNavigation(ulElement);  
  },  
  
  renderElement: function(element, position) {  
    return '<li>' + position + " - " + element.getAttributeValue('name') + '</li>';  
  }  
  
});
```

---

### start()

void start

### Description

`start()` allows you to either set or get the entity number at the beginning of the paginated data.

---

### totalPages()

Number totalPages()

Returns                      Number                      Total number of pages of paginated data

### Description

`totalPages()` allows you to get the total number of pages of paginated data.



`fitToLeft()` allows you to fit the widget to the left of the Page or Container in which it is located. Consequently, the width and height of the widget will be modified.

---

## `fitToRight()`

void `fitToRight()`

### Description

`fitToRight()` allows you to fit the widget to the right of the Page or Container in which it is located. Consequently, the width and height of the widget will be modified.

---

## `fitToTop()`

void `fitToTop()`

### Description

`fitToTop()` allows you to fit the widget to the top of the Page or Container in which it is located. Consequently, the width and height of the widget will be modified.

---

## `left()`

void `left()` ( position )

Parameter	Type	Description
position	Number	Widget's left offset in pixels

### Description

`left()` allows you to set the widget's left offset in pixels.

---

## `position()`

Object `position()`

Returns	Object	Position of the widget
---------	--------	------------------------

### Description

`position()` returns the widget's position depending on its location (if it's in a Container). In the object that it returns, you can retrieve the top and left coordinates:

```
var position = $$('customWidget1').position();
```

The two properties below define the left and top coordinates of the widget's position:

Property	Description
left	Left offset
top	Top offset

---

## `right()`

void `right()` ( position )

Parameter	Type	Description
position	Number	Widget's right offset in pixels

### Description

`right()` allows you to set the widget's right offset in pixels.

---

## `top()`

void `top()` ( position )

Parameter	Type	Description
position	Number	Widget's top offset in pixels

### Description

`top()` allows you to set the widget's top offset in pixels.

## Properties

---

The functions in the Properties category allow you to set or get the value of your widget's property, define a callback when the property changes or bind a datasource attribute to it.

### change

---

#### Description

`change` is an event that is triggered when the property's value changes.

#### Example

The following example shows you how to subscribe to this event:

```
this.subscribe('change', 'test', function() {
  //do something
}, this);
```

### datasourceBindingChange

---

#### Description

`datasourceBindingChange` is an event that is triggered when the property's datasource changes. This event is valid for all properties except those of type List. The `datasourceBindingChange` event is triggered when you:

1. Modify the datasource in the **Property** tab in the GUI Designer,
2. Change the datasource binding using the `{propertyName}.bindDatasource( )` function for a property (except of type Datasource), or
3. Modify the datasource binding using the `{propertyName}.mapping( )` function for **Properties of type Datasource**.

#### Example

The following example shows how to intercept the `datasourceBindingChange` event:

```
var CustomWidget = widget.create('CustomWidget', {
  init: function() {
    this.subscribe('datasourceBindingChange', 'test', function() {
      //do something
    }, this);
  },
  test: widget.property({
    onChange: function(newValue) {
      this.node.innerHTML = this.test();
    }
  })
});
```

### {propertyName}.onChange( )

---

void `{propertyName}.onChange( Function callback )`

Parameter	Type	Description
<code>callback</code>	Function	Callback function to call

#### Description

`{propertyName}.onChange( )` allows you to set a callback when the value of the property changes.

#### event object for onChange event

In the event object are the following properties:

Name	Description
<code>data.oldValue</code>	Previous value for the property
<code>data.value</code>	Actual value for the property
<code>kind</code>	Event type
<code>parentEvent</code>	Event that triggered this event
<code>target</code>	Property name

#### Example

In the following example, we set a callback function to occur when the value in the `test` property changes:

```
documentEvent.onLoad = function documentEvent_onLoad (event)
{
  $$('customWidget1').test.onChange(function(newValue) {
    //do something here
  });
};
```

The *newValue* parameter contains the new value of the property.

## `{propertyName}()`

---

```
void {propertyName}(String | Number | Boolean value)
```

Parameter	Type	Description
value	String, Number, Boolean	Value to set for the property

### Description

`{propertyName}()` allows you to get or set the value in the property whose name is the function's name.

### Getting the property's value

To retrieve the value in a property you defined as "titleProp", you write the following:

```
var propValue = this.titleProp(); //propValue contains the current value of the titleProp property
```

### Setting the property's value

If you want to set the value of the "titleProp" property, you write the following:

```
this.titleProp("New value");
```

## `bindDatasourceAttribute()`

---

```
void bindDatasourceAttribute(DataSource datasource, String attribute [, String property])
```

Parameter	Type	Description
datasource	DataSource	Datasource in which the attribute is located (e.g., sources.company)
attribute	String	Attribute name as defined in the datastore class
property	String	Custom widget's property name to bind to the datasource attribute

### Description

With `bindDatasourceAttribute()`, you bind a datasource attribute to a property and define specific options regarding events, callbacks, etc.

There are two ways in which you can use this function: either by defining the custom widget's property or by defining an event for one or more callbacks.

The syntax in which you pass the *datasource* as a string is the only one you can use in the GUI Designer because the actual datasource does not exist until the Page is run. The property will be modified in the **Properties** panel after you call `bindDatasourceAttribute()`.

### options parameter

The options parameter is an object in which you can specify the following properties:

Property	Description
datasource	Datasource (i.e., sources.company)
attribute	Attribute name
setCallback	Set callback function (not to be used if the <i>property</i> parameter is specified)
getCallback	Get callback function (not to be used if the <i>property</i> parameter is specified)
event	Event name
callback	Callback function

*Note: This syntax cannot be used to change the datasource and attribute when working in the GUI Designer. You must use the other syntax to do so.*

### property parameter

The *property* parameter is the custom widget's property. You cannot use the `setCallback` or `getCallback` parameters in the *options* property if you specify the custom widget's property in the *property* parameter.

### Example

In our example below, we modify the datasource attribute bound to our custom widget's "widgetPropName" property so that it's now bound to the "name" attribute in the "company" datasource:

```
this.bindDatasourceAttribute({
  datasource: sources.company,
  attribute: 'name'
},
'widgetPropName'
);
```

### Example

The example below allows you to define a callback function that sets the value for the "myProp" property defined for your custom widget:

```
this.bindDataAttribute({
  datasource: sources.company,
  attribute: 'name',
  setCallback: function(value) { this.myProp(value); }
});
```

## Example

This example shows you how to bind a datasource to an attribute when working in the GUI Designer:

```
$$("customWidget1").bindDatasourceAttribute("company.name", "nameProp");
```

*Note: Since the datasource does not yet exist, you cannot use the other syntax in which the datasource is defined in the options parameter.*

## **bindDatastourceElement()** **\*\*NOT PUBLIC\*\***

---

```
void bindDatastourceElement **NOT PUBLIC**
```

## Description

By default binding is done on the current element. This function allows you to choose which element (based on its number) to bind the datasource to.

## Properties of type Datasource

---

All the functions in this category can be used on a property of type "datasource". For example, if the syntax is `{propertyName}.attributes( )`, you'd write the following if your property's name of type "datasource" is "testProperty":

```
var myAttributes = $$('customWidget1').testProperty.attributes( )
```

or, when in the "widget.js" file:

```
var myAttributes = this.testProperty.testProperty.attributes( )
```

### datasourceBindingChange

---

#### Description

`datasourceBindingChange` is an event that is triggered when the property's datasource changes. This event is valid for all properties except those of type List. The `datasourceBindingChange` event is triggered when you:

1. Modify the datasource in the **Property** tab in the GUI Designer,
2. Change the datasource binding using the `{propertyName}.bindDatasource( )` function for a property (except of type Datasource), or
3. Modify the datasource binding using the `{propertyName}.mapping( )` function for **Properties of type Datasource**.

#### Example

The following example shows how to intercept the `datasourceBindingChange` event:

```
var CustomWidget = widget.create('CustomWidget', {
  init: function() {
    this.subscribe('datasourceBindingChange', 'test', function() {
      //do something
    }, this);
  },
  test: widget.property({
    onChange: function(newValue) {
      this.node.innerHTML = this.test();
    }
  })
});
```

### {propertyName}.attributeFor( )

---

String `{propertyName}.attributeFor( String attribute )`

Parameter	Type	Description
attribute	String	Datasource defined for a property in the "attributes" property
Returns	String	Datasource attribute bound to attribute

#### Description

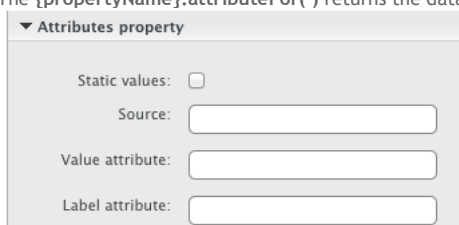
`{propertyName}.attributeFor( )` allows you to retrieve the datasource bound to a property's attribute. For example, if you have created a property of type "datasource":

```
CustomWidget.addProperty('attributes', {
  type: 'datasource',
  attributes: [{
    name: 'value'
  }, {
    name: 'label'
  }]
});
```

If you write:

```
var myds = $$('customWidget1').myAtts.attributeFor('value');
```

The `{propertyName}.attributeFor( )` returns the datasource affected in the "Attribute value" property:



▼ Attributes property

Static values:

Source:

Value attribute:

Label attribute:

### {propertyName}.attributes( )

---

Array `{propertyName}.attributes( )`

Returns	Array	Attributes defined for the property
---------	-------	-------------------------------------



## Description

`{propertyName}.attributes( )` allows you to retrieve the property's attributes.

If you have created your property of type "datasource":

```
CustomWidget.addProperty('myAtts', {
  type: 'datasource',
  attributes: [{
    name: 'value'
  }, {
    name: 'label'
  }]
});
```

In our example above, `{propertyName}.attributes( )` returns an array in which each object is an attribute:

```
[ { name = "value" }, { name = "label" } ]
```

## `{propertyName}.fetch( )`

---

void `{propertyName}.fetch( options )`

Parameter	Type	Description
options	Object	Object containing the properties to define the start and pageSize

## Description

`{propertyName}.fetch( )` allows you to fetch a page of data defined by the *start* and *pageSize* properties.

Property	Description
start	Index of the entity to start the page with
pageSize	Size of the page of entities

## `{propertyName}.getCollection( )`

---

void `{propertyName}.getCollection( Function callback [, Function errorCallback] )`

Parameter	Type	Description
callback	Function	Callback function
errorCallback	Function	Error callback

## `{propertyName}.getPage( )`

---

void `{propertyName}.getPage( start , pageSize , callback , errorCallback )`

Parameter	Type	Description
start	Number	Index number of the entity to begin with. By default it is 0.
pageSize	Number	Number of entities to retrieve
callback	Function	Callback function
errorCallback	Function	Error callback

## Description

`{propertyName}.getPage( )` allows you to retrieve the current collection of the datasource mapped to the property.

```
this.source.getPage(0, 50, function(elements) {
  // do something...
});
```

By default *start* is 0 if it has not been defined. The *pageSize*'s default value is the one defined in the **pageSize** property.

In the callback function's parameter *elements*, we retrieve an array of objects containing the current values for the attributes:

```
[ { company="Apple", url="http://www.apple.com"}, { company="4D", url="http://www.4d.com"} ]
```

## `{propertyName}.mapElement( )`

---

Object `{propertyName}.mapElement( Object map )`

Parameter	Type	Description
map	Object	Map the datasource to the value for each attribute
Returns	Object	Object containing a property for each attribute and its value

## Description

`{propertyName}.mapElement( )` allows you to map an element in the datasource.

In the *map* object, you pass the datasource bound to the property with its value:

```
{ datasourceProperty: "value" }
```

## Example

If you have the following property:

```
CustomWidget.addProperty('myAtts', {
```

```

    type: "datasource",
    attributes: [{
      name: 'coName'
    }, {
      name: 'coUrl'
    }]
  });

```

The "coName" property's datasource is "name" and the "coUrl" property's datasource is "url":

```
{ coName: "name", coUrl: "url" }
```

If you call the following code:

```
var myMap = $$('customWidget1').myAtts.mapElement({ name: "4D", url: "http://www.4D.com/" });
```

*myMap* will return the following:

```
{ coName: "4D", coUrl: "http://www.4D.com/" }
```

---

## **{propertyName}.mapping( )**

void **{propertyName}.mapping**( Object *map* )

Parameter	Type	Description
map	Object	Map for the property's attributes

### Description

**{propertyName}.mapping( )** allows you to get or set the map of the attributes in the datastore class *datasource* defined in the "Source" field. If you want to change the datastore class *datasource*, you must use the **{propertyName}( )** function.

The object you send contains the property and its new *datasource* attribute. The property (or properties) will be modified in the **Properties** panel after you call **{propertyName}.mapping( )**. You must set all the attributes at the same time because if not, the omitted attributes will be set to empty string.

If your property does not have any attributes defined, this function will return *undefined*.

The **datasourceBindingChange** event is triggered after calling this function.

### Example

For example, if we have two attributes in the *myAtts* property: "title" and "link", we assign two *datasource* attributes using this function:

```
var myAttsObject = $$('customWidget1').myAtts.mapping({ title:"fullName", link:"email" });
```

*myAttsObject* returns the mapping of the attributes after execution.

### Example

The following example modifies the datastore class *datasource* and its attributes:

```
$$('customWidget1').myAtts("company");
var myAttsObject = $$('customWidget1').myAtts.mapping({ title:"name", link:"url" });
```

---

## **{propertyName}.onChange( )**

void **{propertyName}.onChange**( Function *callback* )

Parameter	Type	Description
callback	Function	Callback function

### Description

**{propertyName}.onChange( )** allows you to define a callback that will be executed when the property changes.

```
this.myAtts.onChange(function(event) {
  // do something...
});
```

---

## **{propertyName}.onCollectionChange( )**

void **{propertyName}.onCollectionChange**( Function *callback* [, Function *errorCallback*] )

Parameter	Type	Description
callback	Function	Callback function
errorCallback	Function	Error callback

---

## **{propertyName}.onPageChange( )**

void **{propertyName}.onPageChange**( Function *callback* [, Function *errorCallback*] )

Parameter	Type	Description
callback	Function	Callback function
errorCallback	Function	Error callback

### Description

**{propertyName}.onPageChange( )** allows you to install a callback to get the mapped collection if the *datasource* changes.

```
this.myAtts.onPageChange(function(elements) {
    // do something...
});
```

This callback will be called each time:

- data in the current page changes,
- an attribute is changed in the current page,
- each time the current page is changed,
- the first time fetch() is called.

In the callback function's parameter *elements*, we retrieve an array of objects containing the values for the attributes:

```
[ { company="Apple", url="http://www.apple.com"}, { company="4D", url="http://www.4d.com"} ]
```

## **{propertyName}.pageSize( )**

---

Number **{propertyName}.pageSize( size )**

Parameter	Type	Description
size	Number	Page size to define
Returns	Number	Page size can either be a number or Infinity

### **Description**

With **{propertyName}.pageSize( )**, you can set or get the page size for the Datasource property. You can also define this *pageSize* property in the **{propertyName}.fetch( )** function.

## **{propertyName}.setMapping( )**

---

void **{propertyName}.setMapping( Object map )**

Parameter	Type	Description
map	Object	Map for the property's attributes

## **{propertyName}.start( )**

---

Number **{propertyName}.start( )**

Returns	Number	Start index number
---------	--------	--------------------

### **Description**

With **{propertyName}.start( )**, you can retrieve the start for the datasource property. You can define this *start* property in the **{propertyName}.fetch( )** function.

## **{propertyName}()**

---

DataSource **{propertyName}()**

Returns	DataSource	Datasource bound to this property
---------	------------	-----------------------------------

### **Description**

**{propertyName}()** allows you to retrieve the datasource bound to this property. A property for each attribute that you define is included in this datasource object.

For example, if you have defined the "name" attribute from your Company datastore class, the value of the current entity will be in the *mydsName* variable:

```
var mydsName = $$('customWidget1').myds().name;
```

If you want to modify the datasource bound to the property of type DataSource:

```
$$('customWidget1').myds("company");
```

## Properties of type List

---

All the functions in this category can be used on a property of type "list". For example, if the syntax is `{propertyName}.count( )`, you'd write the following if your property's name of type "list" is "listProperty":

```
var myCount = $$('customWidget1').listProperty.count( )
```

or, when in the "widget.js" file:

```
var myCount = this.listProperty.count( )
```

You define **attributes** for the property of type "list". An **element** is the set of values you specify for each set of attributes.

### Events fired by function

The following events are fired after each function is called:

#### `{propertyName}.concat( )`

- onInsert event for each added attribute
- onChange event

#### `{propertyName}.insert( )`

- onInsert event
- onChange event

#### `{propertyName}.move( )`

- onRemove event
- onInsert event
- onMove event
- onChange event

#### `{propertyName}.pop( )`

- onRemove event
- onChange event

#### `{propertyName}.push( )`

- onChange event

#### `{propertyName}.remove( )`

- onRemove event
- onChange event

#### `{propertyName}.removeAll( )`

- onRemove event for each removed attribute
- onChange event

#### `{propertyName}.shift( )`

- onRemove event
- onChange event

## insert

---

### Description

**insert** is an event that is triggered when an element is inserted in the property of type List.

For more information, refer to `{propertyName}.onInsert( )`. To subscribe to this event, use the `subscribe( )` function.

### Example

The following example shows you how to subscribe to this event:

```
this.subscribe('insert', function() {
  //do something
}, this);
```

## modify

---

### Description

**modify** is an event that is triggered when the value of an element is modified in the property of type List.

For more information, refer to `{propertyName}.onModify( )`. To subscribe to this event, use the `subscribe( )` function.

### Example

The following example shows you how to subscribe to this event:

```
this.subscribe('modify', function() {
  //do something
}, this);
```

---

## move

### Description

`move` is an event that is triggered when an element is moved in the property of type `List`.  
For more information, refer to `{propertyName}.onMove()`. To subscribe to this event, use the `subscribe()` function.

---

## remove

### Description

`remove` is an event that is triggered when an element is removed from the property of type `List`.  
For more information, refer to `{propertyName}.onRemove()`. To subscribe to this event, use the `subscribe()` function.

---

## {propertyName}()

Array `{propertyName}()`( Number *element*, Object *listElement* )

Parameter	Type	Description
<code>element</code>	Number	Element to insert or modify
<code>listElement</code>	Object	An object defining an element
Returns	Array	Array of objects defining the elements in the list property

### Description

`{propertyName}()` allows you to return an array of objects in which each object is an element in the list property.  
You can also update an existing element or add a new one by passing the *element* and *listElement* parameters to `{propertyName}()`. Afterwards, the array of objects is returned as well. In this case, the following events are fired: *onRemove* (if an element is removed), *onInsert*, *onModify*, and *onChange*.

### Example

The following example modifies the first element in the list property:

```
var listElements = $$("customWidget1").listproperty(0, {value: "modifiedValue", label:"Modified Value" });  
listElements contains an array defining the elements in the list property with the first one modified as specified.
```

---

## {propertyName}.concat()

void `{propertyName}.concat()`( Array *listElements* )

Parameter	Type	Description
<code>listElements</code>	Array	An array containing multiple objects that define an element

### Description

`{propertyName}.concat()` allows you to add an array of elements, *listElements*, in the property.  
The *onInsert* event is fired for each attribute added and then the *onChange* event is fired.

### Example

The following example adds two elements in the listProp property:

```
$$("customWidget1").listProp.concat( [ { value: "value1", label: "label1" }, { value: "value2", label: "label2" } ] );
```

---

## {propertyName}.count()

Number `{propertyName}.count()`

Returns	Number	Number of elements defined in the list property
---------	--------	---

### Description

`{propertyName}.count()` returns the number of elements in the list property.

---

## {propertyName}.first()

void `{propertyName}.first()`

### Description

`{propertyName}.first()` allows you to select the first element in the property of type "list".

---

## {propertyName}.insert()

Number `{propertyName}.insert()`( element , *listElement* )

Parameter	Type	Description
<code>element</code>	Number	Element where <i>listElement</i> is inserted

listElement	Object	An object defining an element
Returns	Number	Index of listElement inserted in the property

## Description

`{propertyName}.insert( )` allows you to insert a *listElement* in the property at *element*. This function returns the added element's index in the property. The *onInsert* event is fired and then afterwards the *onChange* event.

## Example

The following example inserts *listElement* at the first position:

```
var index = $$("customWidget1").listProp.insert(0, { value: "value1", label: "label1" }); //returns 0
```

---

## `{propertyName}.last( )`

```
void {propertyName}.last( )
```

## Description

`{propertyName}.last( )` allows you to select the last element in the property of type "list".

---

## `{propertyName}.move( )`

```
void {propertyName}.move( Number element, Number newPosition )
```

Parameter	Type	Description
element	Number	Element to move
newPosition	Number	New position for element

## Description

`{propertyName}.move( )` allows you to move *element* to *newPosition*.

The *onRemove*, *onInsert*, and *onMove* events are fired after calling `{propertyName}.move( )`. As usual, the *onChange* event is fired afterwards.

## Example

This example moves the 7th element to the 1st position:

```
$$('customWidget1').listProp.move(6,0);
```

---

## `{propertyName}.onChange( )`

```
void {propertyName}.onChange( Function callback )
```

Parameter	Type	Description
callback	Function	Callback function

## Description

`{propertyName}.onChange( )` allows you to define a *callback* that will be executed when the property changes.

```
this.listproperty.onChange(function(event) {
    // do something...
});
```

This *callback* is executed after other events occur, like an object being inserted, removed, and moved.

## event object

The *event* object returns two properties:

Property	Type	Description
index	Number	Element number
value	Array	An array of objects (defining the elements) in the List property that were affected

---

## `{propertyName}.onInsert( )`

```
void {propertyName}.onInsert ( callback )
```

Parameter	Type	Description
callback	Function	Callback function

## Description

`{propertyName}.onInsert( )` allows you to define a callback function when an element is inserted in the property of type List.

## event object

The *event* object returns two properties:

Property	Type	Description
index	Number	Element number
value	Array	An array of objects (defining the elements) in the List property that were affected

---

### `{propertyName}.onModify( )`

void `{propertyName}.onModify` ( callback )

Parameter	Type	Description
callback	Function	Callback function

#### Description

`{propertyName}.onModify( )` allows you to define a callback function when the value of an element is modified in the property of type List. You can modify an element by passing the element number and its new values to the `{propertyName}( )` function.

#### event object

The *event* object returns two properties:

Property	Type	Description
index	Number	Element number
value	Array	An array of objects (defining the elements) in the List property that were affected

---

### `{propertyName}.onMove( )`

void `{propertyName}.onMove` ( callback )

Parameter	Type	Description
callback	Function	Callback function

#### Description

`{propertyName}.onMove( )` allows you to define a callback function when an element is moved in the property of type List.

#### event object

The *event* object returns two properties:

Property	Type	Description
index	Number	Element number
value	Array	An array of objects (defining the elements) in the List property that were affected

---

### `{propertyName}.onRemove( )`

void `{propertyName}.onRemove`( Function *callback* )

Parameter	Type	Description
callback	Function	Callback function

#### Description

`{propertyName}.onRemove( )` allows you to define a callback function when an element is removed from the property of type List. If, for example, you call `{propertyName}.removeAll( )`, the callback defined for `{propertyName}.onRemove( )` is called for each object removed.

#### event object

The *event* object returns two properties:

Property	Type	Description
index	Number	Element number
value	Array	An array of objects (defining the elements) in the List property that were affected

---

### `{propertyName}.pop( )`

Object `{propertyName}.pop`( )

Returns	Object	Removed element
---------	--------	-----------------

#### Description

`{propertyName}.pop( )` allows you to remove the last element in the property of type "list". The removed element is returned by this function. The *onRemove* event is fired after calling `{propertyName}.pop( )`. As usual, the *onChange* event is fired afterwards.

---

### `{propertyName}.push( )`

Number **{propertyName}.push( Object *listElement* )**

Parameter	Type	Description
<code>listElement</code>	Object	An object defining an element
Returns	Number	Index of the <code>listElement</code> added

#### Description

**{propertyName}.push( )** allows you to append *listElement* to the property. This function returns the added element's index in the property. The *onChange* event is fired after calling this function.

#### Example

In the following example, we add *listElement* to the end of the elements in the `listProp` property:

```
var index = $$("customWidget1").listProp.push({ value: "value1", label: "label1" });
```

### **{propertyName}.remove( )**

---

Object **{propertyName}.remove( Number *element* )**

Parameter	Type	Description
<code>element</code>	Number	Define which object to remove (first element is 0)
Returns	Object	Removed element

#### Description

**{propertyName}.remove( )** allows you to remove a specific object defined by *element* in the property. The removed object is returned by this function. The *onRemove* and *onChange* events are fired after calling this function.

### **{propertyName}.removeAll( )**

---

void **{propertyName}.removeAll( )**

#### Description

**{propertyName}.removeAll( )** allows you to remove all the elements in the property of type "list". The *onRemove* is fired after the removal of each element. Afterwards, the *onChange* events is fired.

### **{propertyName}.shift( )**

---

Object **{propertyName}.shift( )**

Returns	Object	Removed element
---------	--------	-----------------

#### Description

**{propertyName}.shift( )** allows you to remove the first element defined for the property. The removed element is returned by this function. The *onRemove* and *onChange* events are fired after calling this function.



## Properties of type Template

---

All the functions in this category can be used on a property of type "template". For example, if the syntax is `{propertyName}.templates( )`, you'd write the following if your property's name of type "list" is "listProperty":

```
var myTemplates = $$('customWidget1').myTemplateProperty.templates( )
```

or, when in the "widget.js" file:

```
var myTemplates = this.myTemplateProperty.templates( )
```

### Attributes

All attributes in your template must be in all lowercase letters even if you have included an uppercase letter. For example, if you have named your template's attribute "myAttribute", you must refer to it as "myattribute".

### `{propertyName}()`

---

```
void {propertyName}(String template)
```

Parameter	Type	Description
template	String	Template to define for the property

### Description

`{propertyName}()` allows you to get or set the definition of the current template. The template definition is specified in this property's .

### Example

The following example returns the template string:

```
var templateString=$$("customWidget1").templateProperty(); //returns "<h1>{{name}}</h1><p><b>{{revenues}}</b></p>"
```

### Example

This example sets the template:

```
$$('customWidget1').templateProperty("<h1>name</h1>");
```

### `{propertyName}.attributes()`

---

```
Array {propertyName}.attributes()
```

Returns	Array	List of attributes defined for selected template
---------	-------	--

### Description

`{propertyName}.attributes()` allows you to get the attributes defined in the current template.

### Example

In this example, we retrieve the attributes defined for our template:

```
var templateAttributes = $$("customWidget1").templateProperty.attributes();
```

### `{propertyName}.onChange()`

---

```
void {propertyName}.onChange(Function callback)
```

Parameter	Type	Description
callback	Function	Callback function

### Description

`{propertyName}.onChange()` allows you to define a callback that will be executed when the template or attributes for the template property changes.

```
this.templateProperty.onChange(function(event) {  
    // do something...  
});
```

### event object

If you have not specified the `datasourceProperty` property for the template property, the event object returns only the new template.

If you have specified the `datasourceProperty` property, the event object returns the following properties:

Name	Description
data.oldValue	Previous value for the property
data.value	Actual value for the property

### `{propertyName}.onDataChange()`

---

void **{propertyName}.onDataChange()**( Function *callback* )

Parameter	Type	Description
callback	Function	Callback function

## Description

**{propertyName}.onDataChange()** allows you to define a *callback* that will be executed when the data for the property changes.

```
this.templateProperty.onDataChange(function(rows) {  
    // do something...  
});
```

## rows object

The *rows* object returns an array in which each element contains the data for each template entry. For example, if you define a template like this:

```
{  
    name: 'Template 2',  
    template: '<h1>{{name}}</h1><p><b>{{revenues}}</b></p>'  
}
```

The *rows* object will contain each entity formatted according to the selected template. For our template above, a sample row would be:

```
"<h1>4D</h1><p><b>800000</b></p>"
```

If you have not specified the *datasourceProperty* property for the template property, the *rows* object returns only the current entity and not all the entities in the collection.

## Example

In the following example, we retrieve the data for the *templateProperty* in the parameter for the **{propertyName}.onDataChange()** callback:

```
var MyTemplate = widget.create('MyTemplate', {  
    init: function() {  
        this.templateProperty.onDataChange(function(rows) {  
            this.node.innerHTML = rows.join(""); //display data  
        });  
    }  
    // define the other properties here  
});
```

In the *rows* parameter, the data appears as shown below for our template, which is "<h1>{{name}}</h1><p><b>{{revenues}}</b></p>":

```
["<h1>4D</h1><p><b>900000</b></p>", "<h1>Apple</h1><p><b>800000</b></p>", "<h1>Microsoft</h1><p><b>500000</b></p>",  
"<h1>Google</h1><p><b>600000</b></p>", "<h1>FaceBook</h1><p><b>700000</b></p>", "<h1>Twitter</h1><p><b>300000</b></p>"]
```

*Note:* The "name" attribute was replaced by the company name and "revenues" attribute was replaced by the company's revenues.

## Example

In the following example, we retrieve the current entity for the *templateProperty* in the parameter for the **{propertyName}.onDataChange()** callback:

```
var MyTemplate = widget.create('MyTemplate', {  
    init: function() {  
        this.templateProperty.onDataChange(function(rows) {  
            this.node.innerHTML = rows; //display current entity  
        });  
    }  
    // define the other properties here  
});
```

In the *rows* parameter, the current entity appears as shown below for our template, which is "<h1>{{name}}</h1><p><b>{{revenues}}</b></p>":

```
"<h1>4D</h1><p><b>900000</b></p>"
```

## **{propertyName}.render()**

String **{propertyName}.render()**( Object *data* )

Parameter	Type	Description
data	Object	Attributes with their values for the template
Returns	String	Data formatted based on template

## Description

**{propertyName}.render()** allows you to define data to be applied to the template and get the formatted data based on the template.

## Example

If we have the following format in a template:

```
"<h1>{{name}}</h1><p><b>{{revenues}}</b></p>"
```

You can use the **{propertyName}.render()** function to input the data to format it according to the template:

```
var formattedData = $$("customWidget1").templateProperty.render({ name:"New Company", revenues:"990000" });
```

The data returned in *formattedData* will be:

```
"<h1>New Company</h1><p><b>990000</b></p>"
```

---

## **{propertyName}.select( )**

void **{propertyName}.select( String template )**

Parameter	Type	Description
template	String	Select a predefined template

### Description

**{propertyName}.select( )** allows you to get or set the name of the selected template. The template name is specified in this property's .

If a template's attribute wasn't defined in the widget's definition, the format will appear although the data will be empty. For example, if your template is "`<h1>{{name}}</h1><p>{{revenues}}</p>`" and you have not defined the revenues attribute, the data returned would be "`<h1>Company Name</h1><p></p>`".

If you modify the current template by using the **{propertyName}( )** function, this function returns null.

### Example

In this example, we retrieve the name of the selected template:

```
var templateName = $$("customWidget1").templateProperty.select(); //returns "Template 1"
```

### Example

In this example, we select one of the predefined templates ("Template 1", "Template 2" or "Template 3").

```
$$("customWidget1").templateProperty.select("Template 2");
```

---

## **{propertyName}.templates( )**

Array **{propertyName}.templates( )**

Returns	Array	Predefined templates for the property
---------	-------	---------------------------------------

### Description

**{propertyName}.templates( )** allows you to retrieve an array of objects in which each object defines the predefined template. Each template is defined in this property's as shown below.

#### templates object

In this array of objects, you define two properties in the object for each template:

Property	Description
name	Display name for your template
template	String that defines the template with each attribute defined within <code>{{</code> and <code>}}</code> characters. No spaces are allowed in the attribute name.

If you have defined a *datasource* property for your widget and specified it in the template property's *datasourceProperty* property, each attribute will be displayed in the section for the *datasourceProperty* property.

### Example

In this example, retrieve the predefined templates:

```
var templates = $$("customWidget1").templateProperty.templates(); //returns an array of objects (one object for each ter
```

## Repeater

---

The following functions are in the `waf-behavior/layout/repeater` behavior.

### getPosition( )

---

Number **getPosition**( Widget | HTMLElement | Number *widget* )

Parameter	Type	Description
widget	Widget, HTMLElement, Number	The widget, the widget's HTML element, or the widget's index
Returns	Number	Position in the datasource corresponding to widget

#### Description

`getPosition( )` allows you to get the position in the datasource corresponding to *widget*.

#### Example

In the following example, we retrieve the position in the datasource of the currently selected row. We then remove the "waf-state-selected" CSS class from the previously selected widget and then get the selected widget corresponding to the position and add the "waf-state-selected" CSS class to it:

```
this.rows.subscribe('currentElementChange', function() {
    var position = this.rows().getPosition();
    this.invoke('removeClass', 'waf-state-selected');
    var widget = this.widgetByPosition(position);
    if(widget) {
        widget.addClass('waf-state-selected');
    }
}, this);
```

### repeatedWidget( )

---

Widget **repeatedWidget**( [Widget *widget*] )

Parameter	Type	Description
widget	Widget	Set the widget to repeat
Returns	Widget	Get the repeated widget

#### Description

With `repeatedWidget( )`, you can either get or set the repeated widget.

For example, you can retrieve the widget that is set to be repeated:

```
var widgetToBeRepeated = $$('mainRepeatingWidget1').repeatedWidget();
```

To set the widget to repeat, you can write:

```
$$('mainRepeatingWidget1').repeatedWidget($$('widgetToRepeat1'));
```

### repeaterAttachWidget( )

---

void **repeaterAttachWidget**( Widget *widget*, Number *position* )

Parameter	Type	Description
widget	Widget	Widget instance to attach
position	Number	Position in the datasource

#### Description

`repeaterAttachWidget( )` allows you to attach a cloned widget to the repeater.

This function can be overridden to insert the cloned widget in a different way, or to modify the widget's state according to its position.

### widgetByPosition( )

---

Widget **widgetByPosition**( Number *position* )

Parameter	Type	Description
position	Number	Position of the widget
Returns	Widget	Widget in position or null

#### Description

`widgetByPosition( )` allows you to return the widget that corresponds to a specific *position*. If no widget is found, this function returns null.

For more information, refer to the `getPosition( )` example.

### widgetsToRemoveOnUpdate( )

---

Array **widgetsToRemoveOnUpdate**( Number *start*, Number *end* )

Parameter	Type	Description
start	Number	Start position of the new page to display
end	Number	End position of the new page to display
Returns	Array	List of widgets

## Description

`widgetsToRemoveOnUpdate( )` allows you to return the list of widgets to remove when updating the widget. Generally, all the widgets are removed when the Page is updated. If you want to retrieve them in an array beforehand, you call this method.

## Example

In this example, we retrieve the array of widgets that will be removed on update:

```
proto.widgetsToRemoveOnUpdate = function(start, end) {  
    return [];  
};
```



## Style

---

The functions in the Style category allow you to define the CSS style(s) for the instance of your custom widget.  
For information about Wakanda's generic CSS classes, refer to [Using Wakanda's generic CSS classes](#).

### addClass( )

---

void **addClass**(String *cssClass* )

Parameter	Type	Description
<i>cssClass</i>	String	CSS class to add to the widget

#### Description

**addClass( )** allows you to define your widget's CSS class by passing it to *cssClass*.  
The *cssClass* is added to the widget's DOM node's *class* property:

```
<div id="customWidget1" data-type="CustomWidget" data-lib="WAF" data-package="CustomWidget"
class="waf-widget waf-customwidget customClass1 customClass2" data-constraint-left="true" data-constraint-top="true">
```

For information about Wakanda's generic CSS classes, refer to [Using Wakanda's generic CSS classes](#).

### bindDatasourceAttributeCSS( )

---

Object **bindDatasourceAttributeCSS**( *datasource* , *attribute* , *cssProperty* )

Parameter	Type	Description
<i>datasource</i>	DataSource	Datasource to bind to <i>cssProperty</i>
<i>attribute</i>	String	Attribute name
<i>cssProperty</i>	String	CSS property to bind to <i>datasource</i>
Returns	Object	Subscriber object

#### Description

**bindDatasourceAttributeCSS( )** allows you to bind a *datasource* attribute to a CSS property.  
The subscriber object that is returned can be passed to **removeSubscriber( )** to remove it.

#### Example

In the example below, we bind the *sources.styles.bgcolor* *datasource* attribute with the "background" property in the OnLoad event of our Page:

```
documentEvent.onLoad = function documentEvent_onLoad (event)
{
    var cssSubscriber = $$('customWidget1').bindDatasourceAttributeCSS(sources.styles, "bgcolor", "background");
};
```

Each time the *sources.styles.bgcolor* *datasource* changes, the background color of our widget changes the value defined in the *datasource* attribute.

### hasClass( )

---

Boolean **hasClass**(String *cssClass* )

Parameter	Type	Description
<i>cssClass</i>	String	CSS class
Returns	Boolean	True/False = custom widget has <i>cssClass</i>

#### Description

**hasClass( )** allows you to check if *cssClass* exists for the custom widget.

### hide( )

---

void **hide**( )

#### Description

Call the **hide( )** function to hide the widget on the Page.

### removeClass( )

---

void **removeClass**(String *cssClass* )

Parameter	Type	Description
<i>cssClass</i>	String	CSS class to remove

#### Description

**removeClass( )** allows you to remove a CSS class from the custom widget.

## show( )

---

void **show**( )

### Description

With the **show( )** function, you can show the widget on the Page.

## style( )

---

void **style**( String *cssProperty*, String *value* )

Parameter	Type	Description
cssProperty value	String String	CSS property, e.g., "background", "font-size", "border" Value for the cssProperty

### Description

**style( )** allows you to define a value for a CSS property.

For example, you can write the following to set the background of the widget to grey at runtime:

```
$$ ( 'customWidget1' ).style( 'background', '#ccc' );
```

*Important: The **style( )** function cannot be used on custom widgets in Wakanda Studio. If you want to modify the CSS styles of a widget, you can modify them directly in the widget's CSS file.*

## toggleClass( )

---

void **toggleClass**( String *cssClass* )

Parameter	Type	Description
cssClass	String	CSS class to toggle

### Description

**toggleClass( )** allows you to toggle a CSS class for the custom widget. If it exists in the custom widget's "class" property, it will be removed and if it does not exist, it will be added.



## Subscriber

---

The functions in this category allow you to interact with a subscriber that you defined for a specific event by using the functions in the **Observable** category.

### isPaused()

---

Boolean **isPaused()**

Returns Boolean True/False = event is paused

#### Description

**isPaused()** allows you to discover if the event is paused.

### pause()

---

void **pause()**

#### Description

**pause()** allows you to temporarily pause the event.

### resume()

---

void **resume()**

#### Description

**resume()** allows you to resume a paused event.

### unsubscribe()

---

void **unsubscribe()**

#### Description

**unsubscribe()** allows you to unsubscribe a subscriber.

#### Example

If you subscribe to an event, as shown below:

```
var mySubscriber = $$('customWidget1').subscribe('click', function() {  
  //do something  
});
```

You can also unsubscribe from it in the following manner:

```
$$('customWidget1').unsubscribe({  
  event: 'click'  
});
```

or

```
mySubscriber.unsubscribe(); //the subscriber object was created with the subscribe() function
```

## Widget

---

These functions allow you to interact with your custom widget's instance.

By default, you can either set or get the value of a property by using the `{propertyName}()` function.

### Widget properties

The following properties are available at runtime for the custom widget:

Property	Description
id	Widget ID
kind	Widget name

## options

---

### Description

The `options` property contains an object that contains all the properties that begin with "data-".

This property can also be used when instantiating a widget. For more information, refer to [Instantiating a widget at runtime](#).

### Example

For the custom widget whose DOM node is the following:

```
<div id="customWidget1" class="waf-widget waf-widget waf-customwidget" data-constraint-right="false" data-constraint-bottom="false" data-binding-test="person.lastName" data-constraint-left="true" data-constraint-top="true" data-lib="WAF" data-package="CustomWidget" data-type="CustomWidget">
```

The `options` property is the following:

```
{
  binding-test: "person.lastName",
  constraint-bottom: "false",
  constraint-left: "true",
  constraint-right: "false",
  constraint-top: "true",
  lib: "WAF",
  package: "CustomWidget",
  type: "CustomWidget"
}
```

## parentWidget

---

### Description

The `parentWidget` property returns an object of the widget's parent widget (in which it is contained).

The object for a widget is returned in `$$("customWidget1")`. If you place `customWidget1` inside of `customContainer1` and you write the following:

```
var parentWidgetObj = $$("customWidget1").parentWidget;
```

`parentWidgetObj` will contain the same object returned by `$$("customContainer1")`.

## allChildren()

---

Array `allChildren()`

Returns                      Array                      Children widgets (along with children's children widgets)

### Description

With `allChildren()`, you can retrieve all the "children" widgets. The "children" widgets are all those that are included in the "parent" widget.

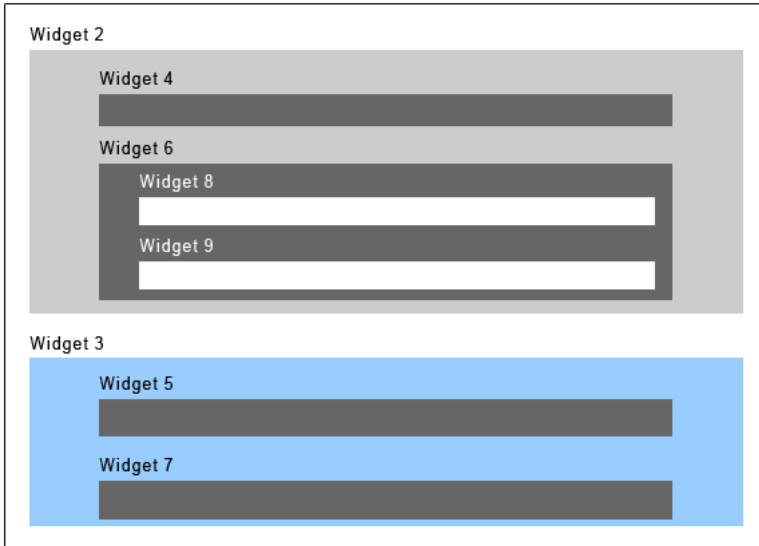
If you call the `allChildren()` function on `widget2` (see screenshot below), an array containing all the children widgets is returned:

```
var myChildren = $$('widget2').allChildren(); //returns [$$('widget4'), $$('widget6'), $$('widget8'), $$('widget9')]
```

If you call the `children()` function on `widget1`, an array containing the children widgets is returned.

```
var myChildren = $$('widget1').children(); //returns [$$('widget2'), $$('widget3')]
```

## Widget 1



*Important Note: This function only returns the widgets that were created with the Widgets v2 architecture.*

## children( )

---

Array children()

Returns                      Array                      Children widgets (only first level)

### Description

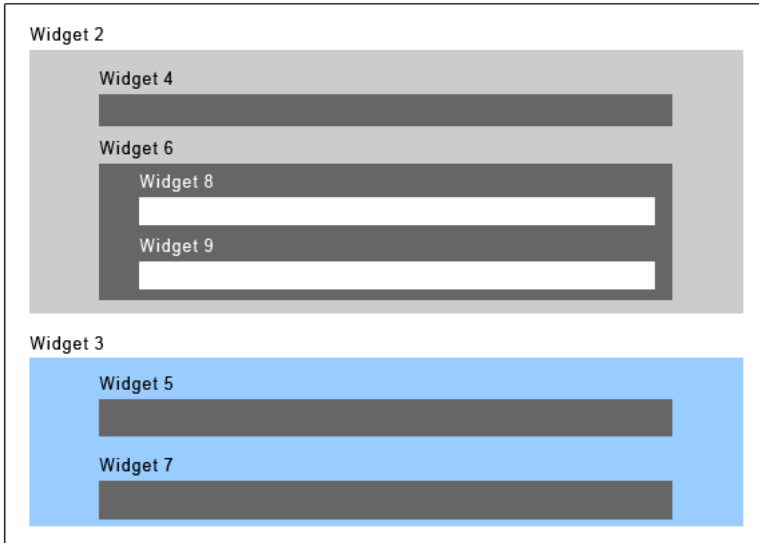
With `children( )`, you can retrieve the "children" widgets for the "parent" widget. The "children" widgets are all those included in the "parent" widget. If you call the `allChildren()` function on `widget2` (see screenshot below), an array containing all the children widgets is returned:

```
var myChildren = $$('widget2').allChildren(); //returns [$$('widget4'), $$('widget6'), $$('widget8'), $$('widget9')]
```

If you call the `children()` function on `widget1`, an array containing the children widgets is returned.

```
var myChildren = $$('widget1').children(); //returns [$$('widget2'), $$('widget3')]
```

## Widget 1



*Important Note: This function only returns the widgets that were created with the Widgets v2 architecture.*

## destroy( )

---

void destroy()

### Description

`destroy( )` allows you to delete the widget (along with any of its "children" widgets) from the Page and removes all the listeners associated to it.

## disable( )

---

void **disable()**

## Description

**disable()** allows you to disable the widget.

The "waf-state-disabled" CSS class is added to the widget's "class" property:

```
<div id="customWidget1" class="waf-widget waf-customwidget waf-state-disabled"... >
```

## **disabled()**

---

Boolean **disabled()**

Returns Boolean True/False = widget is disabled

## Description

With **disabled()**, you can test if the widget has been disabled or not.

## **enable()**

---

void **enable()**

## Description

**enable()** allows you to enable the widget if it had previously been disabled.

## **getNode()**

---

String **getNode()**

Returns String Widget's DOM node

## Description

With **getNode()**, you can retrieve the widget's DOM node. The widget's DOM node is also in the *node* property.

```
var myNode = $$('customWidget1').getNode();
```

The node returned looks something like this:

```
<div id="customWidget1" data-type="customWidget" data-constraint-left="true" data-constraint-top="true" data-label="Label" data-label-position="top" data-lib="WAF" data-package="customWidget" class="waf-widget waf-customwidget" >
```