Widgets v2 Class API

To define how the widget behaves in Wakanda Studio's GUI Designer in the **designer.js** file, refer to the functions in the **Studio** category.
In the **widget.js** file, you develop your custom widget with functions in this API.
You can manipulate the instance of your widget (either in the code or at runtime) by using the functions in **Widgets v2 Instance API**.

Composed

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

addAliasProperty( )

void **addAliasProperty**(String *alias*, String *part*, String *property*)

Parameter	Type	Description
alias	String	Name of the alias property
part	String	Name of the part from which to copy the property
property	String	Name of the property to copy

Description

addAliasProperty( ) allows you to create an alias property of one part and attach it to the widget.

Example

The example below creates an alias property on a part:

```
//create the "titlePart" part from the Input widget
CustomWidget.setPart('titlePart', Input);

//create the "title" alias and copy it into the "value" property of the Input widget
CustomWidget.addAliasProperty('title', 'titlePart', 'value');
```

addProxiedEvent( )

void **addProxiedEvent**(String *event* , String *part* [, String *newEvent*])

Parameter	Type	Description
event	String	Event to listen to and refire
part	String	Part name to subscribe
newEvent	String	Change the event type

Description

addProxiedEvent( ) allows you to fire ***newEvent*** when ***event*** for ***part*** is called.

Example

In the following example, when an action is executed on the "closeButton" part, the "close" event will be called on CustomWidget.

```
CustomWidget.setPart('closeButton');
CustomWidget.addProxiedEvent('action', 'closeButton', 'close');
```

addProxiedMethods( )

void **addProxiedMethods**(String | Array *methods* [,Function *part*] [,String *prefix*] [,String *suffix*])

Parameter	Type	Description
methods	String, Array	List of method names or behavior
part	Function	Part name to proxy
prefix	String	String to prefix for all the functions created
suffix	String	String to append for all the functions created

Description

addProxiedMethods( ) allows you to add methods to access the methods in the list or in the behavior of the widget's part. The created methods can be prefixed and/or suffixed. Please note that the actual function name will begin with a capital letter if you add a prefix so that we can respect the lower camelcase rule for functions.

Example

In the example below, we take the **hide()** and **show()** methods and add a prefix to them so that we can use them for a part of the widget. Our new functions will be **partHide()** and **partShow()**.

```
CustomWidget.addProxiedMethods(['show',  'hide'], 'part1', 'part'); //add the prefix "part" to the show()  and hide() fun
```

We can then call the function that will be applied to the widget's part:

```
$$('customWidget1').partHide(); //hide the "part1" part in the custom widget
```

getParts( )

Widget **getParts**( )

Returns	Widget	Parts for the widget

Description

getParts( ) allows you to retrieve all the parts.

Example

For example, if you write the following:

```
var myParts = ComposedWidget.getParts();
```

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

removePart( )

void **removePart**(String *part*)

Parameter	Type	Description
part	String	Part name

Description

removePart( ) allows you to remove a part.

Example

In this example, we remove one of the parts:

```
ComposedWidget.removePart('part1');
```

setPart( )

void **setPart**(String *part* [, Widget *widget*])

Parameter	Type	Description
part	String	Part name
widget	Widget	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:

```
ComposedWidget.setPart('part',PartWidget);
```

Container

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

edihzedch
zxlkjjzex

grhrhrrhr

gegege

test test

test

Un petit test

Les applications de la gamme 4D v12 requièrent au minimum les configurations suivantes :

- **Windows :**

Processeur	Pentium IV
OS	Windows 7, Windows Vista, Windows XP
RAM	1 Go (2 Go recommandés)
Résolution écran	1280*1024

- **Mac OS :**

Processeur	Intel
OS	Mac OS version 10.5 ou ultérieure
RAM	1 Go (2 Go recommandés)
Résolution écran	1280*1024

Les applications de la gamme 4D v12 requièrent au minimum les configurations suivantes.

- **Windows**

Processeur	Pentium IV
OS	Windows 7, Windows Vista, Windows XP
RAM	1 Go (2 Go recommandés)
Résolution écran	1280*1024

- **Mac OS**

Processeur	Intel (R)
OS	Mac OS version 10.5 ou ultérieure
RAM	1 Go (2 Go recommandés)
Résolution écran	1280*1024

Les applications de la gamme 4D v12 requièrent au minimum les configurations suivantes.

- **Windows**

Processeur	Pentium IV
OS	Windows 7, Windows Vista, Windows XP
RAM	1 Go (2 Go recommandés)
Résolution écran	1280*1024

- **Mac OS**

Processeur	Intel (R)
OS	Mac OS version 10.5 ou ultérieure
RAM	1 Go (2 Go recommandés)
Résolution écran	1280*1024

Les applications de la gamme 4D v12 requièrent au minimum les configurations suivantes :

- Windows

Processeur	Pentium IV
OS	Windows 7, Windows Vista, Windows XP
RAM	1 Go (2 Go recommandés)
Résolution écran	1280*1024

- Mac OS

Processeur	Intel (R)
OS	Mac OS version 10.5 ou ultérieure
RAM	1 Go (2 Go recommandés)
Résolution écran	1280*1024

Remember that you cannot instantiate ***EventEmitter*** objects directly; they are instantiated through the emitter itself (for example, a ***socket***).
All ***EventEmitter*** objects emit the 'newListener' event when new listeners are added.

addIndexedEvent( )

void **addIndexedEvent**(String *event* [, String *newEvent*])

Parameter	Type	Description
event	String	Event (on the attached widget) to listen to that refires newEvent
newEvent	String	Event to fire on the parent widget

Description

addIndexedEvent( ) allows you to fire ***newEvent*** for the attached widget when ***event*** is fired for the parent widget. If you do not specify ***newEvent***, the parent's ***event*** event will be fired.
In the event's **event.data** object, the following information is returned:

Attribute	Description
parentEvent	Object defining the parent event
index	Index number of the attached widget
widget	Object defining the widget on which ***newEvent*** is executed

Example

In this example, we use this function for the parent widget, TabViewBar:

```
TabViewBar.addIndexedEvent('action', 'select');
```

When the user select's a TabViewTab (which is the attached widget), the TabViewBar's *select* event is fired.

addIndexedMethods( )

void **addIndexedMethods**(Array | Behavior *methods* [,String *prefix*] [,String *suffix*] [,Number | Function *defaultIndex*])

Parameter	Type	Description
methods	Array, Behavior	Array of method names or a behavior
prefix	String	String to prefix for all the functions created
suffix	String	String to append for all the functions created
defaultIndex	Number, Function	Attached widget's index to use or a function to get it

Description

addIndexedMethods( ) allows you to add methods to access the methods in the list or in the behavior of the attached widgets. The created methods can be prefixed and/or suffixed. Please note that the actual function name will begin with a capital letter if you add a prefix so that we can respect the lower camelcase rule for functions.
defaultIndex can be an integer or a function that returns an integer.
The method's argument are the same as the original method; however, if you have not specified ***defaultIndex*** in addIndexedMethods( ), you must pass it as the first parameter (see Example 1).

Example

In the example below, we take the **hide()** and **show()** methods and add a suffix in them so that we can use them for the attached widget. Our new functions are **hideSubwidget()** and **showSubwidget()**.

```
CustomWidget.addIndexedMethods(['show', 'hide'], '', 'Subwidget'); //add the suffix "Subwidget" to the show() and hide()
```

When we call our widget, we can pass the subwidget's index, which is the *defaultIndex*, so that the **hide()** function is applied to it:

```
$$('customWidget1').hideSubwidget(2); //hide the third subwidget attached to the Container
```

Example

In the example below, we take the **style()** method and add a suffix to it and apply it to the 0 index, which is the first attached widget. Our new function is **styleSubwidget()**.

```
CustomWidget.addIndexedMethods(['style'], '', 'Subwidget', 0); //add the suffix "Subwidget" to the style() function and a
```

When we call our widget, you no longer need to pass the subwidget's index and you pass just the parameters to the **styleSubwidget()** function:

```
$$('customWidget1').styleSubwidget("border-color","#c30"); //define the border for the attached widget
```

Example

If you create a behavior:

```
var MyBehavior = WAF.define('waf-core/behavior').create();
```

You can then use this function to create a set of methods with the specified prefix and/or suffix as shown below:

```
CustomWidget.addIndexedMethods(WAF.require('MyBehavior'), '', 'Subwidget');
```

containerChildrenAreSubWidgets( )

void **containerChildrenAreSubWidgets**( )

Description

containerChildrenAreSubWidgets( ) allows you to define all children containers are subwidgets. This distinction means that other widgets cannot be copied into the subwidget. A subwidget is a part of the widget and therefore cannot be detached.
By default, you cannot include any built-in Wakanda widgets (those developed using the Widgets v1 architecture) into the Container if you have called this function.

Example

In this Container widget, all the children containers are subwidgets. The code below is in the **designer.js** file:

```
(function(MyContainer) {
    MyContainer.containerChildrenAreSubWidgets();
});
```

containerDisableCustomNodes( )

void **containerDisableCustomNodes**( )

Description

containerDisableCustomNodes( ) allows you to disallow the inclusion of built-in Wakanda widgets (those developed using the Widgets v1 architecture) into the Container. By default, the built-in Wakanda widgets are allowed in the custom widget of type Container.

Example

In this Container widget, we disallow the developer to include any built-in Wakanda widgets (those developed using the Widgets v1 architecture).
The code below is in the **designer.js** file:

```
(function(MyContainer) {
    MyContainer.containerDisableCustomNodes();
});
```

restrictWidget( )

void **restrictWidget**(Widget *widgetClass*)

Parameter	Type	Description
widgetClass	Widget	Widget class

Description

restrictWidget( ) allows you to restrict one 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 the **insertWidget( )** or **attachWidget( )** functions, or by dropping it into the Container in the GUI Designer.
The widget class you specify also extends to widgets that inherit the same widget class.
***Note**: You must also include the widgets you restrict in the **WAF.define()** as shown in our example below.*

Example

In the following example, we restrict the widget classes to Button:

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

    CustomWidget.restrictWidget(Button);

    return CustomWidget;
});
```

Container Properties

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

The **linkListPropertyToContainer( )** function allows you to create a subwidget for each List item in your Container widget.

linkListPropertyToContainer( )

void **linkListPropertyToContainer**(String *property* [, Object *options*])

Parameter	Type	Description
property	String	Name of the property of type List
options	Object	Options for the Container

Description

linkListPropertyToContainer( ) allows you to link a property of type List to a Container so that each item will create a subwidget. By doing so, you can create a widget for each List item that is added to the Container widget.

The default subwidget that is created for each List item must be defined in your widget's **WAF.define()** and included in your widget's **package.json** file. For more information, refer to the **Adding another custom widget** paragraph.

You define the Widget class by either passing it to the **restrictWidget( )** function or by defining it in the *defaultWidgetClass* property in this function's *options* property.

options property

In the *options* property, you can specify the following property:

Property	Type	Description
defaultWidgetClass	Widget Class	Default widget class for new items, which will be subwidgets. You can also define the *defaultWidgetClass* by using the **restrictWidget( )** function on the Container widget directly.

Example

In this example, the *MyWidget* widget will create a *MyButton* widget for each "items" List item defined in the GUI Designer.

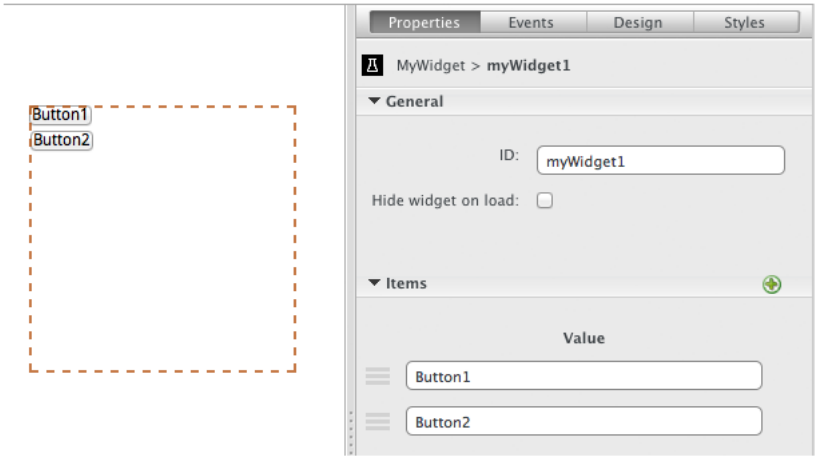
Here is the *MyWidget*'s "widget.js" file:

```
WAF.define('MyWidget', ['waf-core/widget', 'MyButton'], function(widget, MyButton) {

    var MyWidget = widget.create('MyWidget');

    MyWidget.inherit('waf-behavior/layout/container');
    MyWidget.inherit('waf-behavior/layout/properties-container');

    MyWidget.addProperty('items', {
        type: 'list',
        attributes: [{
            name: 'value'
        }]
    });

    MyWidget.linkListPropertyToContainer('items', {
        defaultWidgetClass: MyButton
    });

    return MyWidget;

});
```

The "MyButton" widget's "widget.js" file is the following:

```
WAF.define('MyButton', ['waf-core/widget'], function(widget) {

    var MyButton = widget.create('MyButton', {
        init: function() {},
        value: widget.property({
            onChange: function() {
                this.node.innerHTML = this.value();
            },
            defaultValueCallback: function() {
                return this.node.innerHTML;
            }
        })
    });

    MyButton.tagName = 'button';

    return MyButton;

});
```

When you create an item in the List property, a subwidget is created as shown:


Button1
Button2
Properties
Events
Design
Styles
MyWidget > myWidget1
General
ID:
myWidget1
Hide widget on load:
Items
Value
Button1
Button2

DOM Helpers

The **mapDomEvents( )** function in this category allows you to map a DOM event to either a DOM event or a custom event you created by using the **fire( )** function.

mapDomEvents( )

void **mapDomEvents**(Object *map* [, String *selector*])

Parameter	Type	Description
map	Object	DOM event(s) mapped to one or more events
selector	String	A CSS selector that restricts the DOM event to a specific sub node

Description

mapDomEvents( ) allows you to define one or more events to execute for a DOM event defined in *map*. The event can either be a DOM event or a custom event that you create using the **fire( )** function.

By calling this function, *event* is added automatically to the **Events** tab in the GUI Designer in the "General Events" category. See the below. You can modify the display name and the category by calling the **addEvent( )** or **addEvents( )** functions.

For more information regarding DOM events, refer to **DOM events**.

map property

In the *map* parameter, you define the DOM event to map to one or more events.

Attribute	Description
domEvent	DOM event (i.e., "click","dblclick")
event	Custom event created using the **fire( )** function

domEvent and *event* must begin with a lowercase letter, can contain only letters and numbers, and must not include any spaces.

Example

To enable a DOM event for your custom widget, you pass the same event as the domEvent:

```
CustomWidget.mapDomEvents({
    'click': 'click',
    'dblclick': 'dblclick'
});
```

To assign multiple DOM events to an event:

```
CustomWidget.mapDomEvents({
    'mouseover mousedown': 'myEvent'
});
```

To assign multiple events to a DOM event:

```
CustomWidget.mapDomEvents({
    'click': ['action1','action2']
});
```

To affect an event to a DOM event for a particular CSS selector:

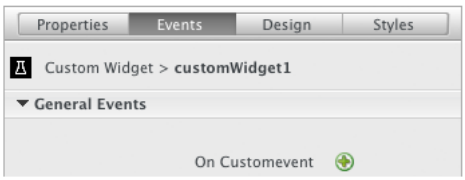
```
CustomWidget.mapDomEvents({
    'click': 'action1'
}, 'span.waf-selected'));
```

Example

The following code placed in the "widget.js" file binds the "customevent" to the standard DOM click event so that "customevent" will be fired when you click on the custom widget:

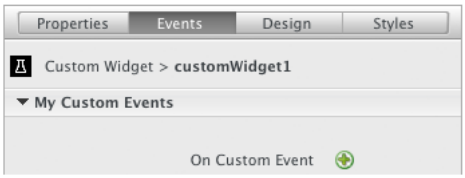
```
CustomWidget.mapDomEvents( { 'click': 'customevent' } );
```

Your "customevent" appears as shown below in the GUI Designer:



If you wan to customize the display of the event, you can write the following in your "designer.js" file:

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


Properties
Events
Design
Styles
Custom Widget > customWidget1
My Custom Events
On Custom Event

Formatters

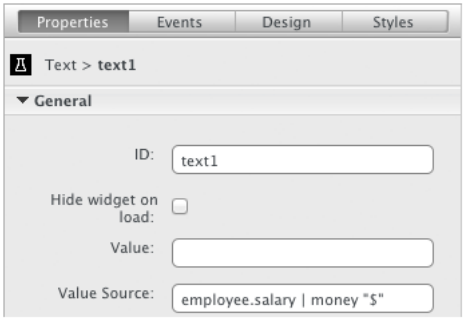
This class allows you to create formatters for your data. To do so, you must define two methods that allow you to return the *formatted* and the *unformatted* version of your data.

You must include the "waf-core/formatters" library to create a formatter. In our example below, we create a "lowercase" format:

```
WAF.require('waf-core/formatters').create('lowercase', {
    format: function(value) { return value.toLowerCase(); },
    unformat: function(value) { return value; },
});
```

Example

The following example shows you how you can create a formatter to format money. For the attribute you assign to a property, you can pass both the datasource and the currency symbol in the GUI Designer:



Or you can pass it directly to the **{propertyName}.bindDatasource( )** function:

```
$$('customWidget1').value.bindDatasource("employee.salary | money \"$\"");
```

You can create the "money" formatter as shown below:

```
Formatter.create('money', {
        unformat: function(value, devise) {
            value = value.replace(' ', '');
            value = value.replace(',', '.');
            return parseFloat(value);
        },
        format: function(value, devise) {
            var v = String(Math.floor(value));
            return [].map.call(v, function(s, i) {
                        if((v.length - i) % 3 === 0 && i) {
                            s = ' ' + s;
                        }
                        return s;
                    }).join('') +
                    ',' +
                    Math.round(100 * (value % 1)) + devise;
        }
    });
```

Formatter arguments

Formatters accept one or more arguments. You define the format by specifying it after the "|" character:

```
employee.birthDate|formatDate 'MM/DD/YYYY'
```

If you want to pass two arguments, you can do so:

```
employee.type|translate employee.language|truncate 31 '...'
```

You can also nest formatters:

```
 employee.type|concat [employee.function|translate employee.language]|truncate 31 '...'
```

The arguments can be of the following types :

- **string**: Text enclosed in single or double quotes, e.g., 'MM/DD/YYYY'
- **number**: An actual number not enclosed in quotes, e.g., 24.
- **databinding expression**: Expression containing datastore class and attribute, e.g., employee.language

create( )

void **create** (format , formatMethods)

Parameter	Type	Description
format	String	Name of the format to create
formatMethods	Object	Object containing two properties (format and unformat) in which to define the functions to receive the unformatted data and return the formatted unformatted

Description

create( ) allows you to create a formatter that returns the data either formatted or unformatted based on the functions you define in the second parameter.

Example

The following example creates a "lowercase" format:

```
WAF.require('waf-core/formatters').create('lowercase', {
    format: function(value) { return value.toLowerCase(); },
    unformat: function(value) { return value; },
});
```

Example

The following example shows you how you can create a formatter to format money. You pass two arguments: datasource and currency symbol:

```
employee.salary | money "$"
```

You can create the "money" formatter as shown below:

```
Formatter.create('money', {
        unformat: function(value, devise) {
            value = value.replace(' ', '');
            value = value.replace(',', '.');
            return parseFloat(value);
        },
        format: function(value, devise) {
            var v = String(Math.floor(value));
            return [].map.call(v, function(s, i) {
                        if((v.length - i) % 3 === 0 && i) {
                            s = ' ' + s;
                        }
                        return s;
                    }).join('') +
                    ',' +
                    Math.round(100 * (value % 1)) + devise;
        }
    });
```

Methods Helper

The functions in the Methods Helper category allow you to create class or instance methods for your widget as well as defining callbacks at different times. The *init* instance method exists by default for your custom widget, which means that you can pass it as the method name to functions like **doAfter( )**.

addClassMethod( )

void **addClassMethod**(String *name*, Function *function*)

Parameter	Type	Description
name	String	Class method name
function	Function	Function for the class method

Description

addClassMethod( ) allows you to create a class method for your custom widget.

```
CustomWidget.addClassMethod('testClassMethod', function(that){
  // do something
});
```

addClassMethods( )

void **addClassMethods**(Object *object*)

Parameter	Type	Description
object	Object	An object for each class method { name: function() }

Description

addClassMethods( ) allows you to create multiple class methods for your custom widget.

```
CustomWidget.addClassMethods({
        testClassMethod1: function(that){
              //do something
        },
        testClassMethod2: function(that){
              //do something
        }
  });
```

addMethod( )

void **addMethod**(String *name*, Function *function*)

Parameter	Type	Description
name	String	Instance method name
function	Function	Function for the instance method

Description

addMethod( ) allows you to create an instance method for your custom widget.

```
CustomWidget.addMethod('testClassMethod', function(that){
  // do something
});
```

addMethods( )

void **addMethods**(Object *object*)

Parameter	Type	Description
object	Object	An object for each instance method { name: function() }

Description

addMethods( ) allows you to create multiple instance methods for your custom widget.

```
CustomWidget.addMethods({
        testClassMethod1: function(that){
              //do something
        },
        testClassMethod2: function(that){
              //do something
        }
  });
```

doAfter( )

void **doAfter**(String *name*, Function *callback*)

Parameter	Type	Description
name	String	Name of the instance method
callback	Function	Function to call after

Description

doAfter( ) allows you to execute the ***callback*** function after the ***name*** instance method is executed with the same arguments. The return value is the one returned by the ***name*** instance method.

doAfterClassMethod( )

void **doAfterClassMethod**(String *name*, Function *callback*)

Parameter	Type	Description
name	String	Name of the class method
callback	Function	Function to call after

Description

doAfterClassMethod( ) allows you to execute the ***callback*** function after the ***name*** class method is executed with the same arguments. The return value is the one returned by the ***name*** class method.

doBefore( )

void **doBefore**(String *name*, Function *callback*)

Parameter	Type	Description
name	String	Name of the instance method
callback	Function	Function to call before

Description

doBefore( ) allows you to execute the ***callback*** function before the ***name*** instance method is executed with the same arguments. The return value is the one returned by the ***name*** instance method.

doBeforeClassMethod( )

void **doBeforeClassMethod**(String *name*, Function *callback*)

Parameter	Type	Description
name	String	Name of the class method
callback	Function	Function to call before

Description

doBeforeClassMethod( ) allows you to execute the ***callback*** function before the ***name*** class method is executed with the same arguments. The return value is the one returned by the ***name*** class method.

wrap( )

void **wrap**(String *name*, Function *wrapper*)

Parameter	Type	Description
name	String	Function to wrap
wrapper	Function	Wrapper function callback

Description

wrap( ) allows you to wrap the ***name*** instance method so it is available as the first argument of the ***wrapper*** function. The ***name*** instance method is not executed until the ***wrapper*** function calls it.

wrapClassMethod( )

void **wrapClassMethod**(String *name*, Function *wrapper*)

Parameter	Type	Description
name	String	Function to wrap
wrapper	Function	Wrapper function callback

Description

wrapClassMethod( ) allows you to wrap the ***name*** class method so it is available as the first argument of the ***wrapper*** function. The ***name*** class method is not executed until the ***wrapper*** function calls it.

Modules

A module is JavaScript code that is completely encapsulated. A widget is a module in that its code is encapsulated and you can define files that are required for your widget.

For this reason, to create a custom widget, you use the **define( )** function:

```
WAF.define('CustomWidget', ['waf-core/widget'], function(widget) {
        // more code here
        return CustomWidget;
});
```

You can load a module with your custom widget by using the **require( )** function:

```
WAF.require('waf-core/formatters');
```

define( )

void **define** (moduleName , fileDependencies , function)

Parameter	Type	Description
moduleName	String	Name of the module/custom widget
fileDependencies	Array	Files that are dependent for your module
function	Function	Function that returns the module

Description

define( ) allows you to define a module (which can be a custom widget).

Example

Here is how to create a custom widget:

```
WAF.define('CustomWidget', ['waf-core/widget'], function(widget) {
        // more code here
        return CustomWidget;
});
```

require( )

void **require** (module)

Parameter	Type	Description
module	String	Path to the library to include

Description

require( ) allows you to load a module with your custom widget.

Properties

The functions in this category allow you to create properties for your widget, retrieve an array of them, and remove one if necessary.

{propertyName}.bindDatasource( )

void **{propertyName}.bindDatasource**(DataSource *datasource*)

Parameter	Type	Description
datasource	DataSource	Datasource to bind to the property

Description

{propertyName}.bindDatasource( ) allows you to bind a datasource to a property. This function does not work with properties of type Datasource.

After you call this function, the **datasourceBindingChange** event is triggered.

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 {propertyName}.bindDatasource( ).

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 *property* parameter is specified)
getCallback	Get callback function (not to be used if the *property* 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.*

Example

The following example changes the datasource for a property:

```
$$('customWidget1').strProperty.bindDatasource("company.name");
```

Example

The following example changes the datasource and attribute for a property:

```
$$('customWidget1').strProperty.bindDatasource({
   datasource: sources.company,
   attribute: 'name'
   });
```

{propertyName}.boundDatasource( )

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

Returns	Object	Object containing details about the datasource bound to the property

Description

{propertyName}.boundDatasource( ) allows you to retrieve information about the datasource bound to the property.

datasource object

The object returned by {propertyName}.boundDatasource( ) contains the following properties:

Property	Description
datasourceName	Name of the datasource
datasource	Datasource object
attribute	Datasource attribute (for properties of all types except datasource)
attributes	Datasource attributes (for a property of type datasource)
formatters	Array of formatters defined for the datasource

{propertyName}.unbindDatasource( )

void **{propertyName}.unbindDatasource**(DataSource *datasource*)

Parameter	Type	Description
datasource	DataSource	Datasource to unbind from the property

Description

{propertyName}.unbindDatasource( ) allows you to remove the bind between the property and the datasource. This function does not work with properties of type Datasource.

If no datasource is bound to the property, this function returns null.

addProperty( )

void **addProperty**(String *name* [, Object *options*])

Parameter	Type	Description
name	String	Property name
options	Object	Object defining the attributes for a property

Description

addProperty( ) allows you to create a property for your widget by defining its *name* and *options*.

To add a Label property, which is an automated feature in Wakanda Studio, to a custom property, you use the **addLabel( )** function in your custom widget's "designer.js" file.

You can also add a property directly in the widget.create() function. For more details, refer to **Adding a property**.

name property

You define the name of your property in the *name* property.

All property names must be in all lowercase letters because at runtime all property names are returned in lowercase letters.

The property's value can be retrieved and set using the following syntax: **{propertyName}( )**. The value that comes from the datasource bound to the widget will be returned even if a static value was entered.

***Note**: To make sure that you do not use a reserved keyword as the name of your property, refer to **Reserved Keywords for Widgets v2 API**.*

options property

In the *options* property, you can define the following options:

Property	Type	Description
type	String	Widget type
defaultValue		Default value of the widget
defaultValueCallback	Function	Callback to define the widget's default value
onChange	Function	Function to be called during the widget's onChange event, which occurs when the value is changed
attributes	Array	Array of objects in which each attribute is defined in an object (name: "attributeName") for properties of type "datasource" and "list"
values	Object	Object to define the value and its display value for properties of type "enum"
bindable	Boolean	True/False: define if the property is bindable to a datasource (valid for all types except "datasource" and "list")
pageSize	Number	Page size of data to display for a property of type "datasource" (Number or Infinity)
datasourceProperty	String	Property of type Datasource that the template will use for the attributes defined for a template (Defined for a "template" property)

***Note**: The pageSize and datasourceProperty properties were added in the Dev Branch.*

type property

The *type* property can be one of the following values:

Type	Description
string	If no property type is defined, it is by default string. (For more information, refer to **Property of type string**.)
integer	This property returns an integer value. (For more information, refer to **Property of type integer**.)
boolean	Returns either true or false. In the GUI Designer, this property is displayed as a checkbox. (For more information, refer to **Property of type boolean**.)
datasource	Multiple *attributes* can be defined and in the GUI Designer they can either be static values or datasources. (For more information, refer to **Property of type datasource**.)
list	For this property, you can create multiple attributes in which you define one or more elements that are all static values. (For more information, refer to **Property of type list**.)
enum	A list of *values* displayed as a dropdown in the GUI Designer. (For more information, refer to **Property of type enum**.)
template	Define templates to create for which the user can define properties (For more information, refer to **Property of type template**.)

***Note**: The template property was added in the Dev Branch.*

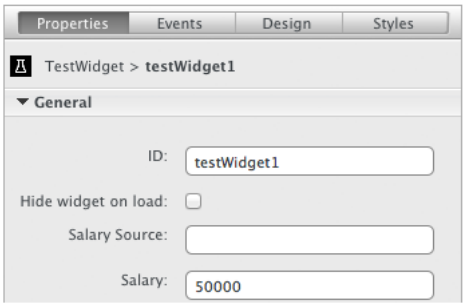
defaultValue property

You define the property's default value in the *defaultValue* property. The type passed to this attribute depends on the property's *type*.

If you have a property of type "integer", you would pass a numeric value as shown below:

```
TestWidget.addProperty('salary', {
    type: "integer",
    defaultValue: 50000
});
```

The value you define appears for the static field for the property:



defaultValueCallback property

In the *defaultValueCallback* property, you can define a function that will be called to set the default value of your property.

```
TestWidget.addProperty('title', {
    defaultValueCallback: function(){
           //do something here
    }
});
```

In the callback function, *this* is the instance of the widget.

onChange property

You pass a callback function to the *onChange* property. Each time the property is changed, this callback is called. The callback's parameter is the changed value of the property.
In the following example, the widget's value is updated at runtime when it changes. The custom widget's value is in the test() function (whose name comes from the property's name).

```
TestWidget.addProperty('test', {
        onChange: function(newValue) {
            this.node.innerHTML = this.test(); //test() contains the widget's value at runtime
        }
    });
```

In the callback function, *this* is the instance of the widget.

attributes property

The *attributes* property, which is used for properties of type "datasource" and "list", is an array of objects in which you define in an array element an attribute name to the *name* attribute. For example:

```
attributes: [{
    name: 'value'
    }, {
    name: 'label'
}]
```

values property

The *values* property, which is used for a property of type "enum", is an object in which you create an attribute for each value (value: "display value"). For example:

```
values: {
    value1: "display value 1",
    value2: "display value 2"
   }
```

bindable property

The *bindable* property defines if you want both the "static" field and the "datasource" field to be displayed for a property. By default, both appear.
If you set *bindable* to false, the property's "datasource" field will not be displayed.

pageSize property

The pageSize property allows you to define the size of the page of data to display for a property of type Datasource.
The possible values are either a specific number or Infinity, which fetches all the entities in the datasource.

```
myItems: widget.property({
     type: 'datasource',
     pageSize: 30
});
```

```
myItems: widget.property({
    type: 'datasource',
    pageSize: Infinity
}),
```

You can also manage the pagination of your property of type datasource by using the **Pagination** functions.

datasourceProperty property

The *datasourceProperty* property defines the property of type datasource that will be used for the attributes defined in the template.This property is only valid for properties of type "template".

HTML properties created for a property

Generally, this function creates two properties for your custom widget for each *propertyName* that you define in ***name***:

- **data-{*propertyName*}**: default (static) value for this property
- **data-binding-{*propertyName*}**: name of the attribute bound to this property (datasource)

For all other property types, refer to the paragraphs below.

datasource property

If you have defined a property of type "datasource", the properties are:

- **data-static-{*propertyName*}**: default (static) values for this property
- **data-{*propertyName*}**: name of the datasource bound to this property
- **data-{*propertyName*}-attribute-{*attributeName*}:** datasources bound to each attribute

list property

If you have defined a property of type "list", the property created is:

- **data-{*propertyName*}**: default (static) value for this property

template property

If you have defined a property of type "template" and have specified a *datasourceProperty* property, each attribute defined for the template will have the following property created:

- **data-{*datasourcePropertyName*}-attribute-{*attributeName*}**: binding value for the attribute

Otherwise, if you have not specified a *datasourceProperty* property, the property of type "template" adds the following property for an attribute in the template:

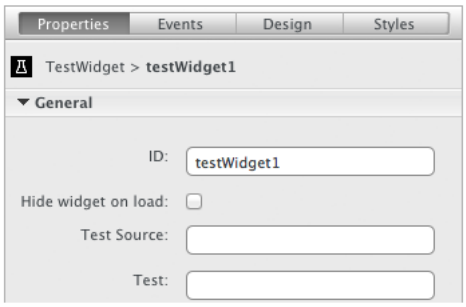
- **data-{*propertyName*}-binding-{*attributeName*}**: binding value for the attribute

Example

In the example below, we define the "test" property, which is by default of type "string".

```
TestWidget.addProperty('test');
```

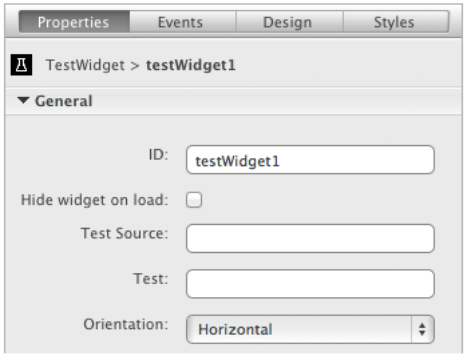
The property appears as two fields "Test Source" and "Test". You can insert a datasource in the "Test Source" field and a static value in the "Test" field.



Example

In the following example, we define a property that will be displayed as a dropdown and does not have a "source" field attached to it:

```
CustomWidget.addProperty('orientation', {
    type: "enum",
    "values": {
        horizontal: "Horizontal",
        vertical: "Vertical"
    },
    bindable: false
});
```

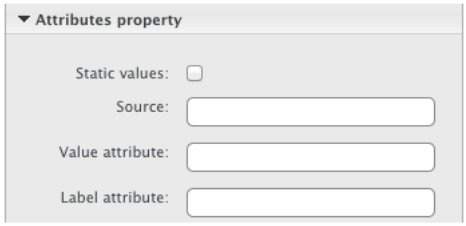


Example

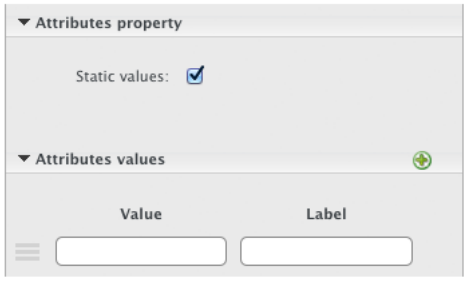
In this example, we define the "attributes" property of type "datasource":

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

In this section on the **Properties** tab, you can define the datasources for the attributes defined:



If you click on the **Static binding** checkbox, you can define static values for the attributes:



getProperties( )

Array Function **getProperties** ( )

Returns	Array	Array of the widget's defined properties

Description

getProperties( ) returns in an array the widget's properties that were added using **addProperty( )**.

removeProperty( )

void **removeProperty**(String *name*)

Parameter	Type	Description
name	String	Property name

Description

removeProperty( ) allows you to remove a property from your widget by passing its *name*.

Example

In the example below, we remove the "test" property.

```
CustomWidget.removeProperty('test');
```

Repeater

The following functions are in the **waf-behavior/layout/repeater** behavior.
For the repeater behavior to work, you must do the following:

1. Create a widget with a property of type Datasource.
2. Link the attribute from the property of type Datasource using **linkDatasourcePropertyToRepeater( )**.
3. Define the widget to repeat using **repeatedWidget( )**.
4. Map the attribute to the repeated widget's property using **mapAttributesToRepeatedWidgetProperties( )**.

If you want to repeat multiple widgets, you can create a Container in which you can place the widgets.
For more detailed information, refer to the example for **linkDatasourcePropertyToRepeater( )**.

linkDatasourcePropertyToRepeater( )

void **linkDatasourcePropertyToRepeater**(String *property*)

Parameter	Type	Description
property	String	Name of the property of type Datasource

Description

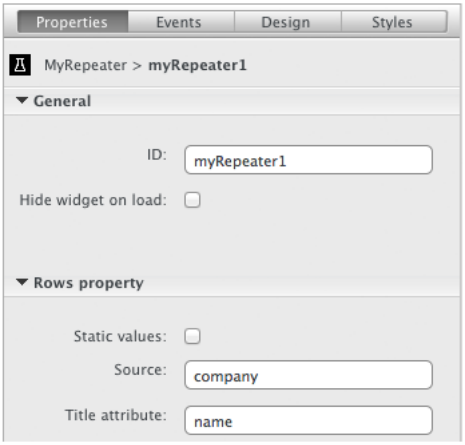
linkDatasourcePropertyToRepeater( ) allows you to link a property of type Datasource to the widget.

Example

The following example links the "title" property from "rows" property of type Datasource with the "value" property of the Text widget:

```
WAF.define('MyRepeater', ['waf-core/widget','Text'], function(widget,Text) {

    var MyRepeater = widget.create('MyRepeater', {
        rows: widget.property({
            type: 'datasource',
            attributes: ['title']
        })
    });

    MyRepeater.inherit('waf-behavior/layout/repeater');
    MyRepeater.linkDatasourcePropertyToRepeater('rows'); //widget's property of type Datasource
    MyRepeater.repeatedWidget(Text);
    MyRepeater.mapAttributesToRepeatedWidgetProperties({
        title: 'value' //'title' attribute from the MyRepeater's 'rows' property to map to the 'value' property of the Te
    });

    return MyRepeater;

});
```

***Note**: The Text widget is also included in the widget's **package.json** file. For more information, refer to the **Adding another custom widget** paragraph.*
Below is the MyRepeater widget in the GUI Designer:



mapAttributesToRepeatedWidgetProperties( )

void **mapAttributesToRepeatedWidgetProperties**(Object *attributesMap*)

Parameter	Type	Description
attributesMap	Object	Object defining the attributes to map to repeated widget's properties

Description

mapAttributesToRepeatedWidgetProperties( ) allows you to map the attributes of the property of type Datasource with the properties of the repeated widget.
Refer to the example for **linkDatasourcePropertyToRepeater( )**.

repeatedWidget( )

void **repeatedWidget**(Widget *widget*)

Parameter	Type	Description
widget	Widget	Set the widget class to repeat

Description

With **repeatedWidget( )**, you can either get or set the repeated widget class.
Refer to the example for **linkDatasourcePropertyToRepeater( )**.

Studio

When creating a new widget, use the functions below in the **designer.js** file to define how your custom widget will be displayed and used in Wakanda Studio's GUI Designer:

```
(function(CustomWidget) {
      // "CustomWidget" is the name of your custom widget
      // define how your widget will appear/interact in the GUI Designer
});
```

_studioOn( ) **NOT PUBLIC**

void _**studioOn **NOT PUBLIC****(String *event*, Function *callback*)

Parameter	Type	Description
event	String	Event to run the callback function
callback	Function	Callback function to run for specified event

Description

_studioOn( ) **NOT PUBLIC** allows you to execute *callback* for an ***event***. The following events are allowed:

Event	Description
Display	When the custom widget is displayed in the GUI Designer
DSDrop	When the custom widget receives a datasource that has been dropped on it
Move	When the custom widget is moved
Resize	When the custom widget is resized
WidgetDrop	When a widget is dropped onto the custom widget
WidgetStartDrag	At the beginning of when a widget is dragged
WidgetDrag	When a widget is being dragged (its position is relative to waf-body)

event object

Depending on the ***event*** you intercept with your ***callback***, the event object varies.

event object for Display

For the Display event, the event object contains the widget's properties as defined in the "designer.js" file.

event object for DSDrop

The **event.source.name** property contains the name of the datasource dropped onto the widget. The value is the exact same one that appears in the **Properties** list.

event object for Move

For the Move event, only the following properties are available:

Property	Description
originalPosition	An object containing the widget's position before the Move event. Its properties are left and top.
position	An object containing the widget's position after the Move event. Its properties are left and top.

event object for Resize

For the Resize event, only the following properties are available:

Property	Description
originalPosition	An object containing the widget's position before the Resize event. Its properties are left and top.
originalSize	An object containing the widget's position before the Resize event. Its properties are height and width.
position	An object containing the widget's position after the Resize event. Its properties are left and top.
size	An object containing the widget's position after the Resize event. Its properties are height and width.

event object for WidgetDrag

For the WidgetDrag event, only the following properties are available:

Property	Description
originalPosition	An object containing the widget's position for the WidgetDrag event. Its properties are left and top.
parent	Either "document" if the widget is on the Page and not in another widget, or the widget.
position	An object containing the widget's position for the WidgetDrag event. Its properties are left and top.

event object for WidgetDrop

For the WidgetDrop event, only the following properties are available:

Property	Description
draggableTag	The widget that was dropped into your widget.

event object for WidgetStartDrag

For the WidgetStartDrag event, only the following properties are available:

Property	Description
parent	Either "document" if the widget is on the Page and not in another widget, or the widget.
position	An object containing the widget's position for the WidgetStartDrag event. Its properties are left and top.

Getting information about parent widget

You can obtain information about the widget (which is called its "parent") in which your custom widget is located.
For example, to get the widget's ID, you can write:

```
var parentID = event.parent.tag.getId(); // for example, container1
```

To get its type, you can write:

```
var parentType = event.parent.tag.getType(); // for example, container
```

this

The *event* parameter is an object containing the widget as it was defined in "designer.js".
Inside *callback*, **this** is an object that represents the widget at runtime.

Example

The following example inserts the custom widget's "data-binding" attribute when displaying the widget in the GUI Designer:

```
widget.on('Display', function(event) {
    $('#' + this.id).html(event['data-binding']); // insert the name of the datasource inside of the widget's main DOM no
});
```

{property}.hide( )

void **{property}.hide**( )

Description

{property}.hide( ) allows you to hide a property in the **Properties** tab. To display it again, you use the **{property}.show( )** function.
You must first add the property to your widget by using the **addProperty( )** function.

{property}.show( )

void **{property}.show**( )

Description

{property}.show( ) allows you to display a property in the **Properties** tab if it has been hidden using the **{property}.hide( )** function.
You must first add the property to your widget by using the **addProperty( )** function.

addEvent( )

void **addEvent**(Object *event*)

Parameter	Type	Description
event	Object	Widget event

Description

addEvent( ) allows you to define an event for a widget to be displayed in the **Events** tab of the GUI Designer. The *event* is defined in an object. You must, however, create the event in the "widget.js" file by using the **fire( )** or **mapDomEvents( )** functions.
For more information regarding DOM events, refer to **DOM events**.

event object

Here are the properties to define an event:

Property	Description
name	Either one of the predefined events in Wakanda or a custom internal name. *name* must begin with a lowercase letter, can contain only letters and numbers, and must not include any spaces.
description	The title of the event to display
category	The category of the event (either as defined by Wakanda or your own)

Example

Below is an example to add an event to a widget:

```
widget.addEvent({
    'name':'eventName',
    'description':'Event Name',
    'category':'Category Name'
});
```

addEvents( )

void **addEvents**(Array *events*)

Parameter	Type	Description
events	Array	Widget events

Description

addEvents( ) allows you to define multiple events for a widget to be displayed in the **Events** tab of the GUI Designer. The ***events*** are defined in an array of objects in which each event is defined in an object. You must, however, create the event in the "widget.js" file by using the **fire( )** or **mapDomEvents( )** functions.

For more information regarding DOM events, refer to **DOM events**.

event object

Here are the properties to define an event:

Property	Description
name	Either one of the predefined events in Wakanda or a custom internal name. *name* must begin with a lowercase letter, can contain only letters and numbers, and must not include any spaces.
description	The title of the event to display
category	The category of the event (either as defined by Wakanda or your own)

Example

Here is an example of how to define multiple events for a widget:

```
widget.addEvents([{
    'name': 'click',
    'description': 'On Click',
    'category': 'Mouse Events'
},{
    'name': 'mouseover',
    'description': 'On Mouse Over',
    'category': 'Mouse Events'
}]);
```

addLabel( )

void **addLabel** (labelDefinition)

Parameter	Type	Description
labelDefinition	Object	Object defining Label widget's default value and position

Description

addLabel( ) allows you to add a Label widget to your custom widget. You can then define it's title in the **Properties** tab as well as its default position in the ***labelDefinition*** object that you pass to this function. The **Label** property will be the last one in the list; however, it will appear before any properties of type "list" or "datasource".

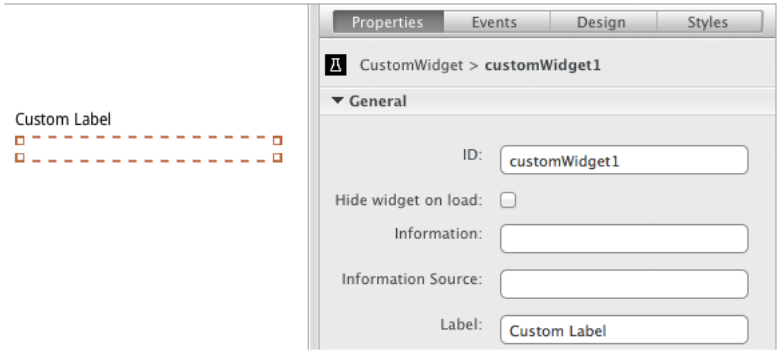
Property	Description
defaultValue	Default value for the Label property
position	Label's default position (top, left, bottom, or right). By default, the position is left.

Example

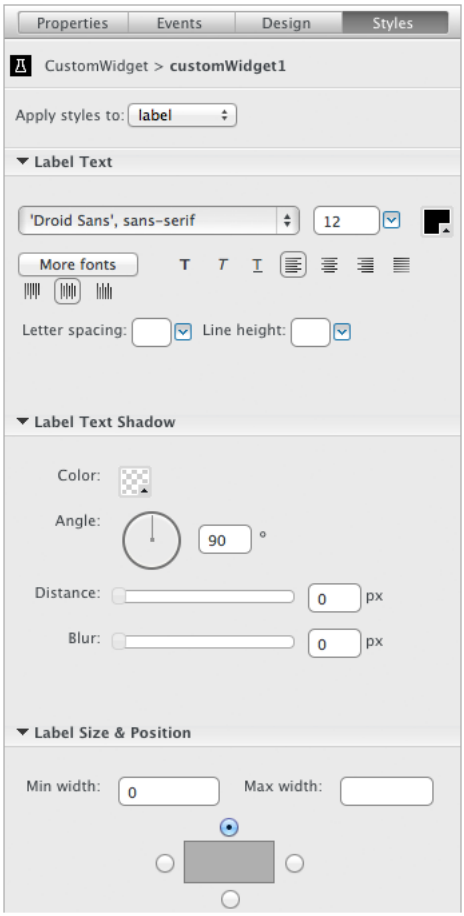
In this example, we create a Label property for our custom widget:

```
CustomWidget.addLabel({
  'defaultValue': 'Custom Label',
  'position': 'top'
});
```

In the GUI Designer, the **Label** property appears on the **Properties** tab. The ***defaultValue*** is entered by default for the Label property and a Label widget is created and attached to the custom widget automatically.



The **Styles** tab also contains the following sections to customize for your custom widget's **Label** property:



addState( )**NOT PUBLIC**

void **addState**NOT PUBLIC**(Object *state*)

Parameter	Type	Description
state	Object	Object defining a particular state in the Styles tab

Description

addState( )NOT PUBLIC**** allows you to define a state in the **Styles** tab. For each state, the same sections are displayed as those defined in the **setPanelStyle( )** function.

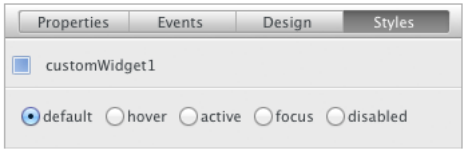
state object properties

Below is a table describing the different properties that you can define for a state:

Property	Description
label	Label of the state to display
cssClass	Name of the CSS class to add for the state.
find	Pass a CSS class to apply the CSS class defined in the cssClass property.
mobile	True/False = display the state for mobile pages.

Example

The following example sets four states in the **Styles** tab:




```
customWidget.addState({
    label: 'hover',
    cssClass: 'waf-state-hover',
    find: '',
    mobile: false
});
widget.addState({
    label: 'active',
    cssClass: 'waf-state-active',
    find: '',
    mobile: false
});
widget.addState({
    label: 'focus',
    cssClass: 'waf-state-focus',
    find: '',
    mobile: false
});
widget.addState({
    label: 'disabled',
    cssClass: 'waf-state-disabled',
    find: '',
    mobile: false
});
```

addStates( )**NOT PUBLIC**

void **addStates**NOT PUBLIC**** (Array *states*)

Parameter	Type	Description
states	Array	Array of objects defining states in the Styles tab

Description

addStates( )NOT PUBLIC**** allows you to define multiple states in the **Styles** tab. For each state, the same sections are displayed as those defined in the **setPanelStyle( )** function.

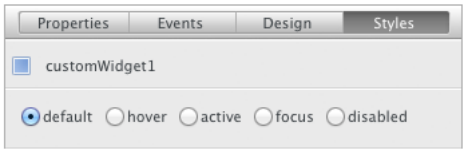
state object properties

Below is a table describing the different properties that you can define for a state:

Property	Description
label	Label of the state to display
cssClass	Name of the CSS class to add for the state.
find	Pass a CSS class to apply the CSS class defined in the cssClass property.
mobile	True/False = display the state for mobile pages.

Example

The following example sets four states in the **Styles** tab:




```
customWidget.addStates([{
    label: 'hover',
    cssClass: 'waf-state-hover',
    find: '',
    mobile: false
},{
    label: 'active',
    cssClass: 'waf-state-active',
    find: '',
    mobile: false
},{
    label: 'focus',
    cssClass: 'waf-state-focus',
    find: '',
    mobile: false
},{
    label: 'disabled',
    cssClass: 'waf-state-disabled',
    find: '',
```

```
    mobile: false
}]);
```

addStructure( ) **NOT PUBLIC**

void **addStructure **NOT PUBLIC****(Object *structure*)

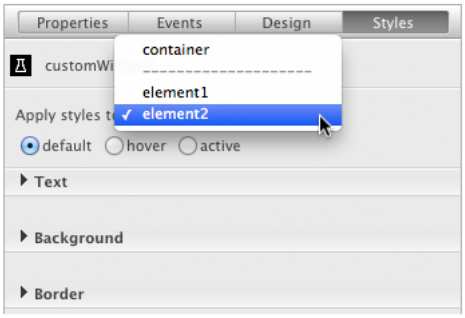
Parameter	Type	Description
structure	Object	Definition of a structure for an element of the widget and its states in the GUI Designer's Styles tab

Description

The **addStructure( ) **NOT PUBLIC**** method allows you to define a more complex structure defining states and elements for your widget in the **Styles** tab.

structure object

You can define the subelements of your widget so that they appear in the dropdown menu in the GUI Designer's **Styles** tab to then be able to modify the styles for each CSS selector:



The “container” option is displayed by default if you have a "label" element or if you define a structure with this function.
You define the following elements in the *structure* array of objects:

Property	Type	Description
description	String	Name of the widget’s sub element displayed in the dropdown in the Styles tab
selector	String	CSS selector of the sub element
style	Object	Define which sections in the Styles tab to display
state	Array	An array of objects defining the label and CSS class for each state

style properties

Below is a table describing the different properties that you can show or hide in the **Skins** and **Styles** tab:

Tag	Style Property	Description
fClass	CSS Classes and Widget Role	True/False = show the “General” section on the **Design** tab, which contains the CSS Classes and Widget Role properties.
text	Text	True/False = show show the “Text” section on the **Styles** tab.
background	Background	True/False = show show the “Background” section on the **Styles** tab.
border	Border	True/False = show to show the “Border” section on the **Styles** tab.
sizePosition	Size & Position	True/False = show show the “Size & Position” section on the **Styles** tab.
textShadow	Shadow	True/False = show show the “Text Shadow” section on the **Styles** tab.
dropShadow	Drop Shadow	True/False = show show the “Drop Shadow” section on the **Styles** tab.
innerShadow	Inner Shadow	True/False = show show the “Inner Shadow” section on the **Styles** tab.
disabled	Disabled	An array of attributes to disable specific properties in certain sections on the **Styles** tab.

disabled Property in the style Array

In this property, you define which of the individual items in the Styles tab you’d like to disable/hide. The sections that are not accessible are the “X” and “Y” properties in the “Size & Position” section as well as the “Label Text” and “Label Size & Position” sections, which are managed internally as mentioned above.

Section	Style Property	Tag
Text	Font	font-family
	Size	font-size
	Color	color
	Bold	font-weight
	Italic	font-style
	Underline	text-decoration
	Text Align	text-align
	Letter-Spacing	letter-spacing
Background	Background	background
	Background Image	background-image
	Background Repeat	background-repeat
	Gradient	background-gradient
Border	Color	border-color
	Style	border-style

	Size	border-size
	Radius	border-radius
Size & Position	Width	width
	Height	height
	z-index	z-index
	Left	left
	Top	top

The *disabled* property is an array in which you define the style properties to hide per section:

```
disabled: ['border-radius','background-image','background-repeat']
```

***Note**: Even if an individual element cannot be hidden in the Styles tab (i.e., "X" and "Y" properties in the "Size & Position" section), you can still hide the entire section in the style property by setting sizePosition to false.*

***Note**: The "letter-spacing" tag hides the "Letter Spacing" field in both the "Text" and "Label Text" sections on the Styles tab.*

state array

Each object in the *state* array has the following properties:

Property	Type	Description
label	String	Label of the subelement in the **Styles** tab
cssClass	String	CSS selector of the sub element
find	String	Define which CSS selector to apply the cssClass property
mobile	Boolean	True/False = only availalable for the mobile platform

Example

The code below defines the subelements as shown in the screen shot above:

```
widget.addStructure({
    description: 'element1',
    selector: '.classElement1',
    style: {
        'text': true,
        'background': true,
        'border': true
    }
});
widget.addStructure({
    description: 'element2',
    selector: '.classElement2',
    style: {
        'text': true,
        'background': true,
        'border': true
    },
    state: [{
        label: 'hover',
        cssClass: 'element2-state-hover',
        find: '.classElement2',
        mobile: false
    }, {
        label: 'active',
        cssClass: 'element2-state-active',
        find: '.classElement2',
        mobile: false
    }]
});
```

addStructures( ) **NOT PUBLIC**

void **addStructures **NOT PUBLIC****(Array *structures*)

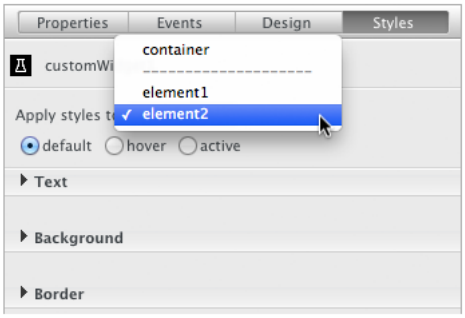
Parameter	Type	Description
structures	Array	Definition of structures for a widget's elements and their states in the GUI Designer's Styles tab

Description

The **addStructures( ) **NOT PUBLIC**** method allows you to define a more complex structure defining states and elements for your widget in the **Styles** tab.

structure object

You can define the subelements of your widget so that they appear in the dropdown menu in the GUI Designer's **Styles** tab to then be able to modify the styles for each CSS selector:



The “container” option is displayed by default if you have a "label" element or if you define a structure with this function.
You define the following elements in the *structure* array of objects:

Property	Type	Description
description	String	Name of the widget’s sub element displayed in the dropdown in the Styles tab
selector	String	CSS selector of the sub element
style	Object	Define which sections in the Styles tab to display
state	Array	An array of objects defining the label and CSS class for each state

style properties

Below is a table describing the different properties that you can show or hide in the **Skins** and **Styles** tab:

Tag	Style Property	Description
fClass	CSS Classes and Widget Role	True/False = show the “General” section on the **Design** tab, which contains the CSS Classes and Widget Role properties.
text	Text	True/False = show show the “Text” section on the **Styles** tab.
background	Background	True/False = show show the “Background” section on the **Styles** tab.
border	Border	True/False = show to show the “Border” section on the **Styles** tab.
sizePosition	Size & Position	True/False = show show the “Size & Position” section on the **Styles** tab.
textShadow	Shadow	True/False = show show the “Text Shadow” section on the **Styles** tab.
dropShadow	Drop Shadow	True/False = show show the “Drop Shadow” section on the **Styles** tab.
innerShadow	Inner Shadow	True/False = show show the “Inner Shadow” section on the **Styles** tab.
disabled	Disabled	An array of attributes to disable specific properties in certain sections on the **Styles** tab.

disabled Property in the style Array

In this property, you define which of the individual items in the Styles tab you’d like to disable/hide. The sections that are not accessible are the “X” and “Y” properties in the “Size & Position” section as well as the “Label Text” and “Label Size & Position” sections, which are managed internally as mentioned above.

Section	Style Property	Tag
Text	Font	font-family
	Size	font-size
	Color	color
	Bold	font-weight
	Italic	font-style
	Underline	text-decoration
	Text Align	text-align
	Letter-Spacing	letter-spacing
Background	Background	background
	Background Image	background-image
	Background Repeat	background-repeat
	Gradient	background-gradient
Border	Color	border-color
	Style	border-style
	Size	border-size
	Radius	border-radius
Size & Position	Width	width
	Height	height
	z-index	z-index
	Left	left
	Top	top

The *disabled* property is an array in which you define the style properties to hide per section:

```
disabled: ['border-radius','background-image','background-repeat']
```

***Note**: Even if an individual element cannot be hidden in the Styles tab (i.e., “X” and “Y” properties in the “Size & Position” section), you can still hide the entire section in the style property by setting sizePosition to false.*

***Note**: The “letter-spacing” tag hides the “Letter Spacing” field in both the “Text” and “Label Text” sections on the Styles tab.*

state array

Each object in the *state* array has the following properties:

Property	Type	Description
label	String	Label of the subelement in the **Styles** tab
cssClass	String	CSS selector of the sub element
find	String	Define which CSS selector to apply the cssClass property
mobile	Boolean	True/False = only availalable for the mobile platform

Example

The code below defines the subelements as shown in the screen shot above:

```
widget.addStructures([{
    description: 'element1',
    selector: '.classElement1',
    style: {
        'text': true,
        'background': true,
        'border': true
    }
},{
    description: 'element2',
    selector: '.classElement2',
    style: {
        'text': true,
        'background': true,
        'border': true
    },
    state: [{
        label: 'hover',
        cssClass: 'element2-state-hover',
        find: '.classElement2',
        mobile: false
    }, {
        label: 'active',
        cssClass: 'element2-state-active',
        find: '.classElement2',
        mobile: false
    }]
}]);
```

customizeProperty( )

void **customizeProperty**(String *property*, Object *options*)

Parameter	Type	Description
property	String	Property name as defined in the widget.js file
options	Object	Options to customize for the property

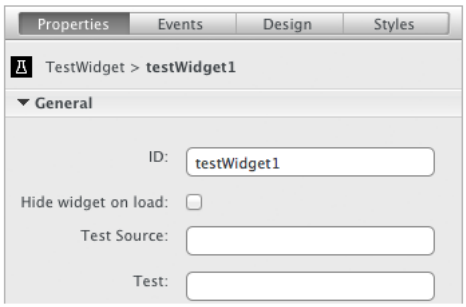
Description

customizeProperty( ) allows you to customize a widget's property in the GUI Designer. Before calling this function, you must first create *property* in the "widget.js" file by using the **addProperty( )** function.

By default, when you define a property in the "widget.js" file its actual name appears in the **Properties** tab along with another field whose name is the property name plus "Source" to show that it's a datasource. In our example, we created a property named "test" in the "widget.js" file:

```
TestWidget.addProperty('test');
```

It appears by default as shown below in the **Properties** tab for our widget:



options parameter

You can use the **customizeProperty( )** function by defining the following attributes in the *options* object:

Property	Description
title	Title of the static field for the property
sourceTitle	Title of the source field for the property
display	True/False = display the static field for the property
sourceDisplay	True/False = display the source field for the property

multiline	True/False = display property of type "string" as a text area
radio	True/False = display property of type "enum" as a group of radio buttons

Note: *You must specify both the display and sourceDisplay property if you set either one to false. Otherwise, both are defined as true.*

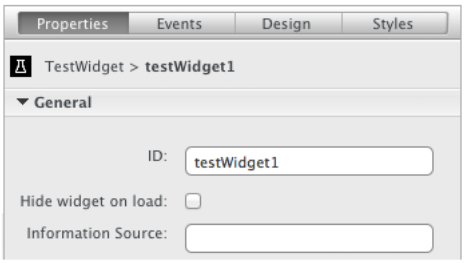
Example

In the following example, we create a property for our widget in the "widget.js" file:

```
TestWidget.addProperty('info');
```

Then, we change the "Info Source" property's title and do not display the static "Info" property in the GUI Designer:

```
TestWidget.customizeProperty('info', {
    sourceTitle: 'Information Source',
    display: false,
    sourceDisplay: true
});
```

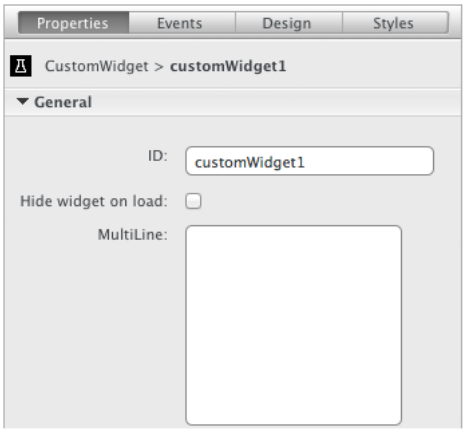


Example

After the "multitext" property was created in the custom widget's "widget.js" file, we change the field type:

```
CustomWidget.customizeProperty('multitext', {
    title: 'MultiLine',
    multiline: true,
    display: true,
    sourceDisplay: false
});
```

The field appears in this way in the GUI Designer:

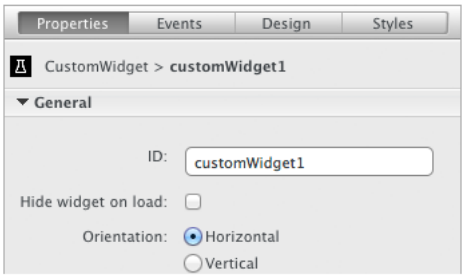


Example

After the "orientation" property (of type "enum") was added to the custom widget's "widget.js" file, the following code changes how it is displayed:

```
CustomWidget.customizeProperty('orientation', {
    title: 'Orientation',
    radio: true,
    display: true,
    sourceDisplay: false
});
```

This field appears in this way in the GUI Designer:



orderEvents( )

void **orderEvents** (events)

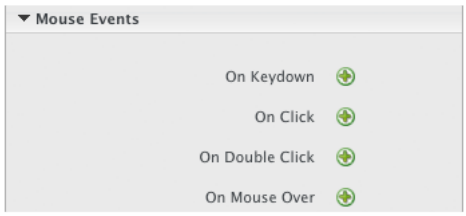
Parameter	Type	Description
events	Array	An array defining the order of the events

Description

orderEvents( ) allows you to order the events to display in the **Events** tab in the GUI Designer.
If the events are in different categories, the categories will also be reordered.
You can also change the order of the "On Change" event in the "Properties" category, by passing its name "change" to this function.

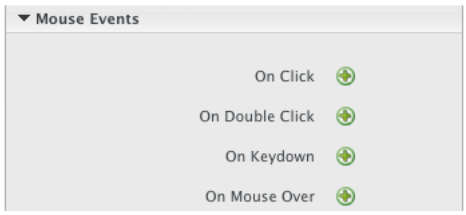
Example

By default, the events appear in the **Events** tab in the order they are created:



You can reorder them by writing the following code:

```
CustomWidget.orderEvents(['click','dblclick','keydown','mouseover']);
```



removeEvent( )

void **removeEvent** (event)

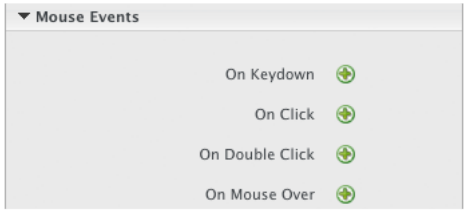
Parameter	Type	Description
event	String	Event to remove from the Events tab

Description

removeEvent( ) allows you to remove an event from the **Events** tab in the GUI Designer. It still exists, but the event is just not available through the GUI Designer.

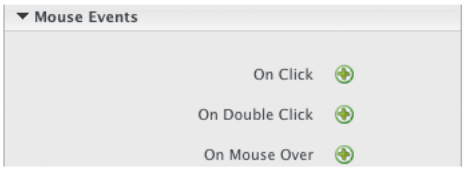
Example

In the following example, we have the following events in the **Events** tab:



To remove one of the events, you can write the following code:

```
CustomWidget.removeEvent('keydown');
```



setHeight( )

void **setHeight**(String *height*)

Parameter	Type	Description
height	String	Default height of the widget

Description

setHeight( ) allows you to set the widget's default height when added to a Page in the GUI Designer. If you do not define a default height with this function, it is set to 300 pixels.

setPanelStyle( )

void **setPanelStyle**(Object *styles*)

Parameter	Type	Description
styles	Object	Object defining the sections to display in the Styles tab and the properties in the Design tab

Description

setPanelStyle( ) allows you to define the sections to display in the **Styles** tab and if the **CSS Classes** and **Widget Role** properties in the Design tab are visible.
By default, the following sections in the **Styles** panel are available:

- Background
- Border
- Size & Position

style properties

Below is a table describing the different properties that you can show or hide in the **Skins** and **Styles** tab:

Tag	Style Property	Description
fClass	CSS Classes and Widget Role	True/False = show the “General” section on the **Design** tab, which contains the CSS Classes and Widget Role properties.
text	Text	True/False = show show the “Text” section on the **Styles** tab.
background	Background	True/False = show show the “Background” section on the **Styles** tab.
border	Border	True/False = show to show the “Border” section on the **Styles** tab.
sizePosition	Size & Position	True/False = show show the “Size & Position” section on the **Styles** tab.
textShadow	Shadow	True/False = show show the “Text Shadow” section on the **Styles** tab.
dropShadow	Drop Shadow	True/False = show show the “Drop Shadow” section on the **Styles** tab.
innerShadow	Inner Shadow	True/False = show show the “Inner Shadow” section on the **Styles** tab.
disabled	Disabled	An array of attributes to disable specific properties in certain sections on the **Styles** tab.

disabled Property in the style Array

In this property, you define which of the individual items in the Styles tab you’d like to disable/hide. The sections that are not accessible are the “X” and “Y” properties in the “Size & Position” section as well as the “Label Text” and “Label Size & Position” sections, which are managed internally as mentioned above.

Section	Style Property	Tag
Text	Font	font-family
	Size	font-size
	Color	color
	Bold	font-weight
	Italic	font-style
	Underline	text-decoration
	Text Align	text-align
	Letter-Spacing	letter-spacing
Background	Background	background
	Background Image	background-image
	Background Repeat	background-repeat
	Gradient	background-gradient
Border	Color	border-color
	Style	border-style
	Size	border-size
	Radius	border-radius
Size & Position	Width	width
	Height	height
	z-index	z-index

Left	left
Top	top

The *disabled* property is an array in which you define the style properties to hide per section:

```
disabled: ['border-radius','background-image','background-repeat']
```

***Note**: Even if an individual element cannot be hidden in the Styles tab (i.e., "X" and "Y" properties in the "Size & Position" section), you can still hide the entire section in the style property by setting sizePosition to false.*

***Note**: The "letter-spacing" tag hides the "Letter Spacing" field in both the "Text" and "Label Text" sections on the Styles tab.*

Example

The following example sets a few sections in the **Styles** tab and enables the **CSS Classes** property in the **Design** tab:

```
widget.setPanelStyle({
        'fClass': true,
        'text': true,
        'background': true,
        'border': true,
        'sizePosition': true,
        'label': true,
        'disabled': ['border-radius']
    });
```

setWidth( )

void **setWidth**(String *width*)

Parameter	Type	Description
width	String	Default width of the widget

Description

setWidth( ) allows you to set the widget's default width when added to a Page in the GUI Designer. If you do not define a default width with this function, it is set to 300 pixels.

studioOnResize( )

void **studioOnResize**

Description

studioOnResize( ) allows you to execute code when a widget is resized in the GUI Designer.
You can resize the widget either by dragging a corner of it or by editing the values in the **Width** and **Height** fields in the **Styles** tab.
You can use this function as shown below in the **designer.js** file:

```
CustomWidget.studioOnResize(function() {
   // the widget was resized, do something here
});
```

Style

These functions allow you to define CSS classes for your 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
cssClass	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">
```

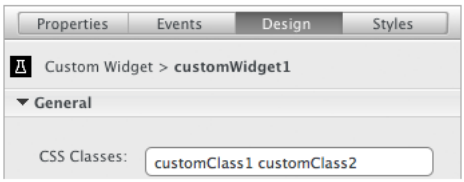
For information about Wakanda's generic CSS classes, refer to **Using Wakanda's generic CSS classes**.

Example

If you write the following code:

```
CustomWidget.addClass("customClass1");
CustomWidget.addClass("customClass2");
```

The **CSS Classes** field in the **Design** tab appears as shown below for your custom widget:



hasClass( )

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

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

Description

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

removeClass( )

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

Parameter	Type	Description
cssClass	String	CSS class to remove

Description

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

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.

Widget

This section of the API allows you to:

- define the tag for the custom widget by using **tagName**.
- create a function for your custom widget, by using **prototype.{method}**.
- define how to initialize your custom widget, use **prototype.init**.

tagName

Description

tagName allows you to define the tag for the widget. By default, the tag is "div".

Example

In the "widget.js" file, you can define the tag to be "input" instead of "div":

```
WAF.define('CustomWidget', function() {
    var widget = WAF.require('waf-core/widget');
    var CustomWidget = widget.create('CustomWidget');

    CustomWidget.tagName = "input";

    //continue the definition of the custom widget...

    return CustomWidget;
});
```

Once the custom widget is added to your Page, its definition in HTML is:

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

prototype.{method}

void **prototype.{method}**

Description

With **prototype.{method}**, you can create a function for the custom widget. For the function to be considered "private", you can prefix it with an underscore.

Example

In the code below, we create a private function (prefixed by an underscore) and a public one:

```
WAF.define('CustomWidget', function() {
    var widget = WAF.require('waf-core/widget');
    var CustomWidget = widget.create('CustomWidget');

    CustomWidget.prototype._privateFunction = function() {
        //do something here in the private function
    };

    CustomWidget.prototype.publicFunction = function() {
        //do something here in the public function
    };

  return CustomWidget;
});
```

prototype.init

void **prototype.init**

Description

In the **prototype.init** function, you initialize your custom widget in which you can do the following:

- define the HTML of the widget and
- declare events

Example

In the example below, we define a custom event:

```
TestWidget.prototype.init = function() {

    /* Define a custom event */
    this.fire('myEvent', {
        message: 'Hello'
    });
};
```

Appendix

In this appendix, we define some of the terminology we use throughout this manual.

DOM events

Here are a few of the standard DOM events that already exist in Wakanda:

Event Name	Description
blur	On Blur
click	On Click
dblclick	On Double Click
focus	On Focus
keydown	On Key Down
keyup	On Key Up
mousedown	On Mouse Down
mousemove	On Mouse Move
mouseout	On Mouse Out
mouseover	On Mouse Over
mouseup	On Mouse Up
touchcancel	On Touch Cancel
touchend	On Touch End
touchmove	On Touch Move
touchstart	On Touch Start

For a complete list of DOM events, refer to the Events section (Mozilla).

DOM event properties and methods

For more information regarding DOM event properties, refer to https://developer.mozilla.org/en/docs/Web/API/Event#Properties.
For more information regarding DOM event methods, refer to https://developer.mozilla.org/en/docs/Web/API/Event#Methods.