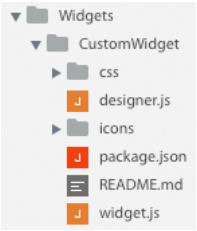
Creating a Custom Widget

In addition to Wakanda's built-in widgets, you also have the possibility to create your own custom widgets or add widgets created from other frameworks. Wakanda offers you the following features available for your custom widgets:

- Install a custom widget easily into your solution,
- Edit the code for your custom widget directly in Wakanda Studio via the Code Editor,
- Define properties (displayed in the **Properties** tab of the Prototyper) that can be datasources or static values for your custom widget,
- Customize events for your custom widget as well as those that can be intercepted by the user from the Prototyper's **Events** tab,
- Define which CSS style sections are available in the Prototyper's **Styles** tab, and
- Drag and drop the custom widget onto the Page Prototype to use it.

A custom widget has the following file structure:



Each custom widget's folder contains the following items:

- **css/widget.css**: The "widget.css" file, which is created by default for your widget, allows you to set your custom widget's default styles at runtime and design mode. The user can override them widget's CSS in the **Styles** tab if you give him/her access to these style settings.
- **designer.js**: JavaScript file allows you to customize your custom widget as it is displayed in the Prototyper (style settings to make available, default size of the widget, and customize properties defined in the widget.js file).
- **icons**: This folder contains the custom widget's .png file, by default named "widget.png", that is used in the Prototyper. The custom widget's icon must be 16x16 pixels.
- **package.json**: A JSON file in which you define the files necessary for your custom widget including any dependencies on other widgets. In this file, you also define the widget's display name, its icon, and its category as well as its meta data (like author, date, and version).
- **README.md**: A text file using Markdown syntax that you can use to describe your widget when distributing it.
- **widget.js**: Your custom widget's JavaScript file in which you code your custom widget and define its properties.

Creating a custom widget in Prototyper

You can create a custom widget from within Wakanda Studio by selecting **New Custom Widget** from the **File** menu or the Widgets folder's contextual menu.
When naming your custom widget, the first letter must be uppercase because each widget is a JavaScript class.

Location of custom widgets

You can install custom widgets either in the selected project's "Widgets" folder or the Widgets favorites folder. Refer to **Defining custom widgets as favorites**, for more information.
Your Favorites folder is in the following location:

- **On Macintosh**: /Users/*userName*/Documents/Wakanda/Widgets/
- **On Windows**: *diskName*:\Users*userName*\Documents\Wakanda\Widgets\

In Wakanda 8, this folder contained the custom widgets that you could use in all of your projects. Now, you can access only the custom widgets installed in your project's "Widgets" folder.
As of Wakanda 9, any custom widget in the favorites folder will be copied automatically into a new project.

Not saving a tag in the HTML file

If in your code, you have an HTML tag that you want to display, but do not want to save with your HTML page, you can use the **waf-studio-donotsave** class.
For example, you could write the following in your custom widget's code using the **doAfter( )** function:

```
CustomWidget.doAfter("init", function() {
        this.node.innerHTML += '<div class="waf-studio-donotsave">This information will not be saved in the HTML</d
    });
```

Adding an attribute to your widget's DOM node

If you want to add an attribute to your widget's DOM node, you must use the **setAttribute()** function. For example:

```
this.node.setAttribute('widgetAttribute', 'my value');
```

Your widget's DOM node would then be:

```
<div widgetAttribute="my value" />
```

Referencing images

Here is how you reference an image placed in your widget's "images" folder depending if you are doing so in a CSS or a JS file.

CSS file

For a CSS file, you use its relative path:

```
background-image: url(../images/background.png);
```

If your image is base64 encoded, you can do the following:

```
background-image: url(...);
```

JS file

In your JS file, you use its absolute path:

```
<img src="/widgets-custom/MyCustomWidget/images/background.png">
```

If your image is base64 encoded, you can do the following:

```
<img src="..." />
```

Instantiating a widget at runtime

Below is how you instantiate a custom widget in JavaScript:

```
var MyNewWidget = WAF.require('CustomWidget');
var instance = new MyNewWidget();
```

The new custom widget's ID will be similar to the one given in Wakanda Studio, i.e., its name starting with a lowercase letter and a number: "customWidget1".

Defining a dependent widget to be installed

If your custom widget requires another widget for it to function properly, you must indicate it in the *externalWidgets* property in the custom widget's **package.json** file.You can also hide it from the Prototyper, by setting the category to "Hidden".

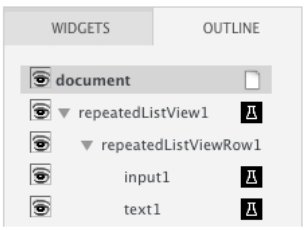
Any dependent widgets will be installed automatically for you. If they existed before, the old ones will be placed in a separate folder.

Widget IDs

The ID for a custom widget added to the Page Prototype will be the widget's name plus a sequential number. If your widget's name is "CustomWidget", the ID of the first widget you add to your Page Prototype will be "customWidget1". The number added at the end will be sequential, starting at 1.

In the case of multiple widgets created for a custom widget, the ID for each widget will be the widget's name plus a sequential number.

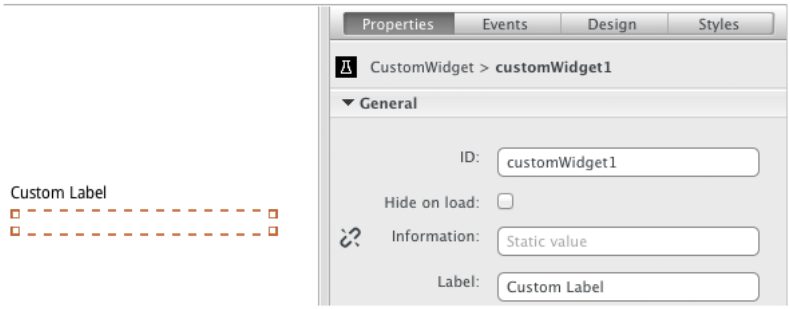
For example, if you have a widget named RepeatedListView widget that creates three widgets (RepeatedListViewRow, Input, and Text), the IDs for each widget will be "repeatedListView1", "repeatedListViewRow1", "input1", and "text1".



Defining a widget's properties

In this chapter, we explain how to add properties to your custom widget. For more information about properties for a widget are displayed, refer to **Widget v1 Properties**.

In our example below, we add one property (which creates two fields in the Prototyper) and a label property:



To add a property to your widget:

- Use the **addProperty( )** function to define the property in your "widget.js" file.
- Use the **customizeProperty( )** function in the "designer.js" file to customize how it appears in the **Properties** tab in the Prototyper.

However, if you want to add a label property to your widget, which is managed automatically by Wakanda Studio, refer to the **Adding a Label property** section below.

***Note:** If you create a property that has the name "myNewProperty", the title in Wakanda Studio by default will be "My New Property". Wakanda looks at the camelcase and adds a space before the capital letter when creating the title.You can always change it by using the **customizeProperty( )** function.*

Adding a property

To define a property, you can either:

1. Use the widget.create() function:

```
var CustomWidget = widget.create('CustomWidget', {
    info: widget.property()
});
```

Or

2. Use the addProperty( ) function in the "widget.js" file

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

Both functions have the same **options property**.

Regardless of the method you use to add a property (named *propertyName)* to your widget, two properties in the DOM node (and the Properties panel) will be created:

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

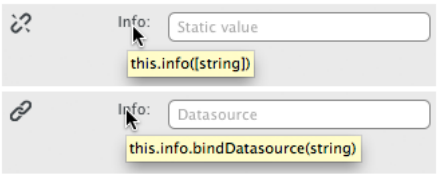
In the Prototyper, the two fields for this property appear as shown below:

***Note:** Click the icon to toggle between the static value and the datasource.*

In our example above, we have the "data-info" and the "data-binding-info" properties in the custom widget's DOM node:

```
<div id="customWidget1" data-type="CustomWidget"  data-package="CustomWidget" data-lib="WAF"
data-info="static value"  data-binding-info="company.url"
data-label-position="left" data-label="Custom Widget Label"
class=""  data-constraint-top="true"
data-constraint-left="true"></div>
```

By default, a tooltip appears when you hover the property's title and is different depending on if you have the static field or the datasource field displayed:



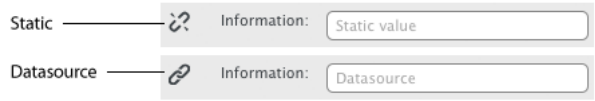
To add your own tooltip message before the syntax, refer to the **customizeProperty( )** function.

Customizing a property

If you want to customize it in the Prototyper, you can do so by using the **customizeProperty( )** function in the "designer.js" file:

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

The "info" property appears as shown below in the **Properties** tab. Click the icon to toggle between the static value and the datasource:



For more information, refer to the **addProperty( )** and **customizeProperty( )** functions.

Adding a Label property

To add a label property for your widget, which Wakanda manages automatically for you, use the **addLabel( )** function. For more information about the Label widget, refer to the **Label Property** section.

Ordering the properties

The order in which the properties are defined in the "widget.js" file is the same order in which they are displayed in the **Properties** tab.
For our CustomWidget example above, the **Label** property is displayed at the bottom of the list.

Property of type boolean

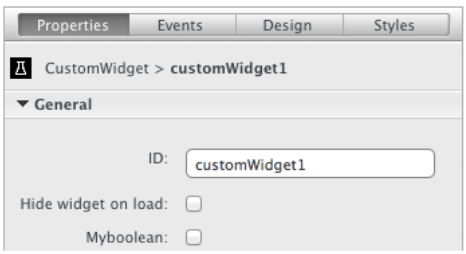
You can define a property of type "boolean" in the "widget.js" file:

```
CustomWidget.addProperty('myboolean', {
    type: 'boolean',
    defaultValue: false,
    bindable: false
});
```

The values for this property type are *true* and *false*.

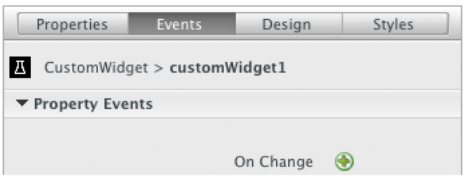
Properties tab

In the **Properties** tab, this property appears as a checkbox in the Prototyper:



Events tab

For this property, you can intercept the **On Change** event in the Prototyper's **Events** tab:



This event is fired when the value for the property has been modified.

event object for onChange event

In the event object are the following properties:

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

Property of type datasource

For a property of type "datasource", you define the values in the "widget.js" file:

```
CustomWidget.addProperty('mysource', {
    type: 'datasource',
    attributes: ['att1', 'att2']
});
```

You can also define the attributes as shown below:

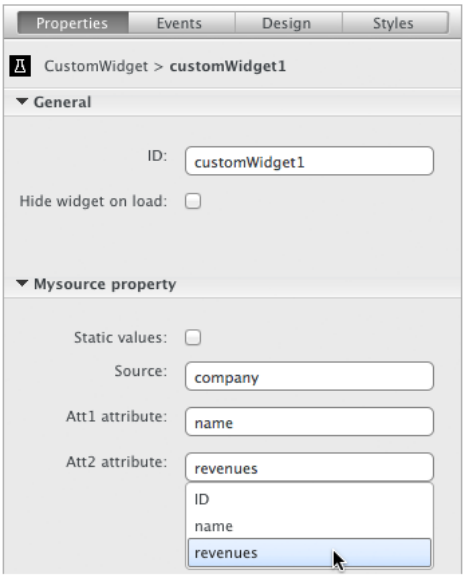
```
CustomWidget.addProperty('mysource', {
    type: 'datasource',
    attributes: [{
        name: 'att1'
    }, {
        name: 'att2'
    }]
```

```
});
```

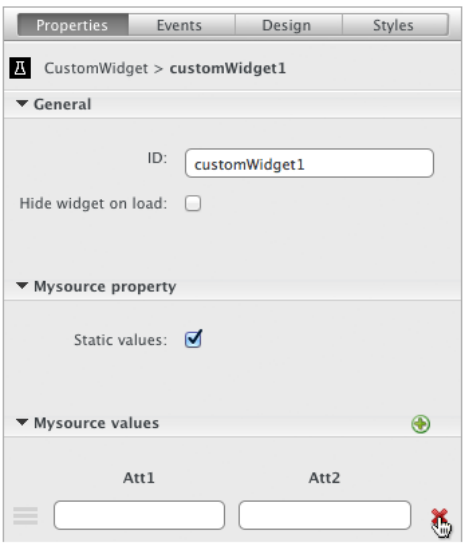
Properties tab

In the **Properties** tab, you define the values for each attribute.

In the property's "property" section, you can bind datasources to the attributes by selecting the datastore class (or array) in the **Source** field and the individual attributes in the attributes that you defined for your property:



To define static values, check the **Static values** checkbox.The property's "values" section appears:

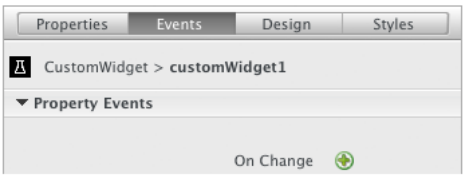


To add an element, click on the icon at the top of this section.

To delete an element, click on the icon.

Events tab

For this property, you can intercept the **On Change** event in the Prototyper's **Events** tab:



This event is fired when the value for the property has been modified.

event object for onChange event

In the event object are the following properties:

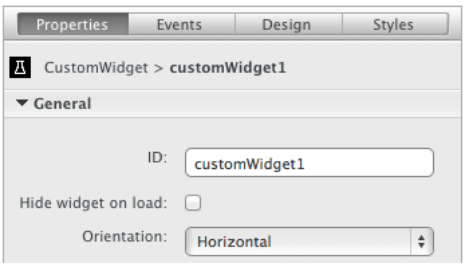
Name	Description
data.oldValue	Previous value for the property
data.value	Actual value for the property
kind	Event type
parentEvent	Event that triggered this event
target	Property name

Property of type enum

For a property of type "enum", you define the values in the "widget.js" file:

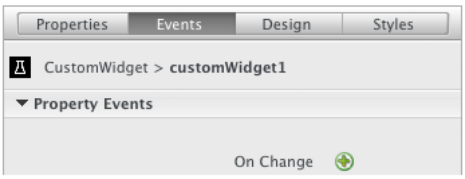
```
CustomWidget.addProperty('orientation', {
    type: "enum",
    "values": {
        horizontal: "Horizontal",
        vertical: "Vertical"
    },
    bindable: false
});
```

In the Prototyper, it appears as shown below:



Events tab

For this property, you can intercept the **On Change** event in the Prototyper's **Events** tab:



This event is fired when the value for the property has been modified.

event object for onChange event

In the event object are the following properties:

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

Property of type file

For a property of type "file", you define it in the "widget.js" file:

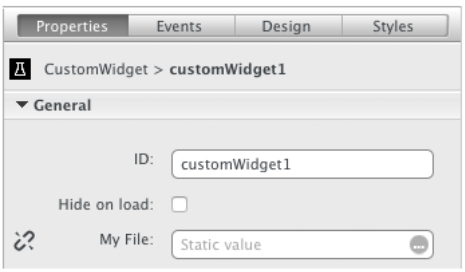
```
CustomWidget.addProperty('myFile', {
     type: 'file',
     folder: 'myfolder',
     accept: 'image/*',
     description: "File property tooltip message"
});
```

Besides the *type* and *description* properties, you can define two optional properties:

Property	Description
folder	(optional) Relative path (relative to the project's web folder) for the folder where the file will be uploaded
accept	(optional) File type to accept, e.g., "image/png" for only accepting "png" files. By default, all types are allowed (which is equal to "*"). For more information regarding media types, refer to the Internet media types page on Wikipedia.

Properties tab

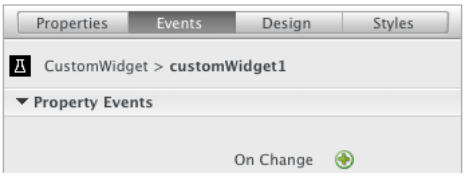
In the **Properties** tab, this property appears as one field in the Prototyper:



You can either select a static file or bind the property to an attribute from a datasource.

Events tab

For this property, you can intercept the **On Change** event in the Prototyper's **Events** tab:



This event is fired when the value for the property has been modified.

event object for onChange event

In the event object are the following properties:

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

Property of type integer

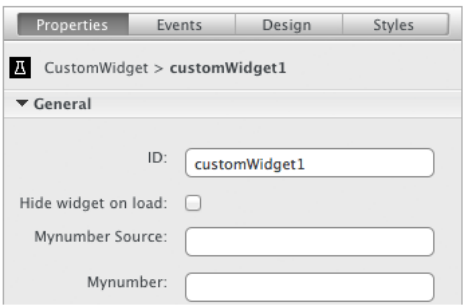
For a property of type "integer", you define the values in the "widget.js" file:

```
CustomWidget.addProperty('mynumber', {
    type: 'integer',
    defaultValue: 0
});
```

The value returned by this property is always a number.

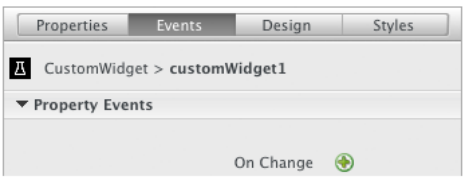
Properties tab

In the **Properties** tab, you either bind a datasource to this property or define a static value for it.



Events tab

For this property, you can intercept the **On Change** event in the Prototyper's **Events** tab:



This event is fired when the value for the property has been modified.

event object for onChange event

In the event object are the following properties:

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

Property of type list

For a property of type "list", you define the attributes in the "widget.js" file.

```
CustomWidget.addProperty('listproperty', {
    type: "list",
    attributes: [{
        name: 'value'
    }, {
        name: 'label'
```

```
    }]
});
```

To create an element, you can:

- Define it in the custom widget's **Properties** tab (in the Prototyper),
- Pass an object containing the values for each attribute to the **{propertyName}.insert( )** function, or
- Pass the element number (where to insert or which one to modify) plus an object containing the values for each attribute to the **{propertyName}( )** function.

attributes property

The *attributes* property allows you to define the following properties:

Property	Description
name	Property name and title displayed in the Prototyper
type	Type of the property (attribute, boolean, enum, integer, or string)
secondary	True = display separately and not in the list; False = display in the list (by default)
datasourceProperty	Datasource to use for the property of type attribute

type property

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

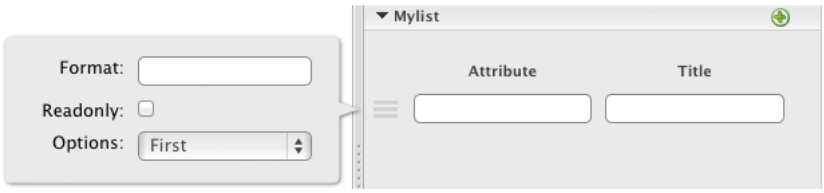
Type	Description
attribute	An attribute in the datasource
boolean	The value will either be True or False and will be displayed as a checkbox
enum	A list in which the values are defined in the values property (which is of type Array)
integer	An integer
string	A string value

If you use "attribute" as the type, you must first define a datasource property for the widget and specify it in *datasourceProperty* for the property as shown below:

```
myDatasource: widget.property({ type: 'datasource' }),
myList: widget.property({
    type: 'list',
    attributes: [
        {
            name: 'attributeProperty',
            type: 'attribute',
            datasourceProperty: 'myDatasource'
        }
    ]
})
```

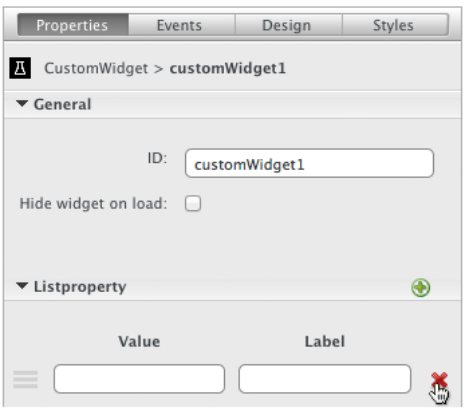
secondary property

The *secondary* property allows you to insert a property that will be displayed separately when the user is modifying a list element:



Properties

In the **Properties** tab, you define an element in the list property's section by specifying a value for each attribute:

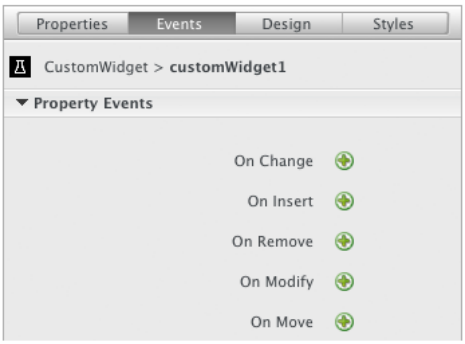


To add an element, click on the icon at the top of the list property's section.

To delete an element, click on the icon.

Events

In the **Events** tab, the property of type "list" activates multiple events besides the On Change event, which appears by default if your widget has at least one property.



Here are the events proposed by the list property:

- **On Change**: When the list property changes.
- **On Insert**: When a new element is added to the list property.
- **On Remove**: When an element is removed from the list property.
- **On Modify**: When one or more values of an existing element are modfiied.
- **On Move**: When an element is moved.

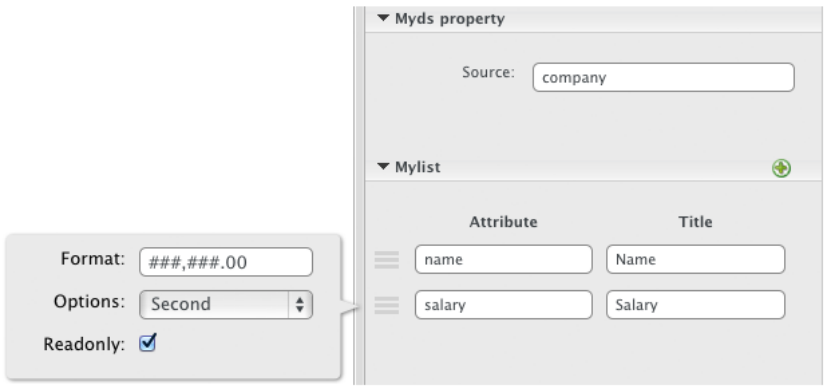
For more information about the events that are fired, refer to **Events fired by function**.

Example

If you want to have a datasource property and then a list with attributes, you can do so as shown below:

```
var CustomWidget = widget.create('CustomWidget', {
    myds: widget.property({ type: 'datasource'}),
    mylist: widget.property({
        type: 'list',
        attributes: [
            { name: 'attribute', type: 'attribute', datasourceProperty: 'myds' },
            { name: 'title', type: 'string' },
            { name: 'format', type: 'string', secondary: true },
            { name: 'options', type: 'enum', values: ['First','Second','Third'], secondary: true },
            { name: 'readonly', type: 'boolean', secondary: true }
        ]
    })
});
```

In the **Properties** panel, the properties are displayed:



The three properties (format, options, and readonly) are not in the list, but rather the window that appears to the left when you are modifying the list properties.

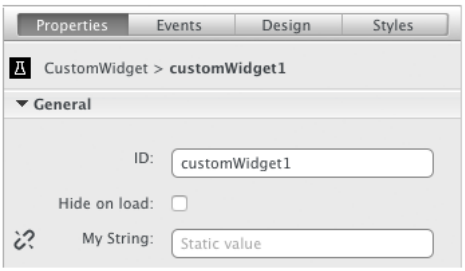
Property of type string

For a property of type "string", which is the type by default, you define the values in the "widget.js" file:

```
CustomWidget.addProperty('mystring', {
    type: 'string'
});
```

Properties tab

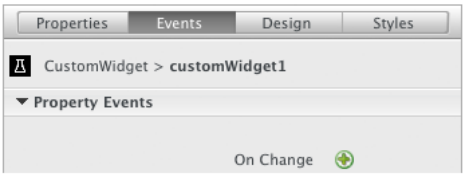
In the **Properties** tab, this property appears as one field in the Prototyper:



You can either enter a static text or bind the property to an attribute from a datasource.

Events tab

For this property, you can intercept the **On Change** event in the Prototyper's **Events** tab:



This event is fired when the value for the property has been modified.

event object for onChange event

In the event object are the following properties:

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

Property of type template

For a property of type template, you define HTML templates in which you define attributes within {{ and }} characters. When the widget is published, the template is rendered with the data from the datasource.

For this type of property, you can either manage the datasource's collection or just one entity.

- To handle only an entity, you just define the templates in the template property.
- To handle the entire collection, you must specify the *datasourceProperty* property as well as create a property of type datasource for your widget that works in conjunction with it. For more information about *datasourceProperty*, refer to **datasourceProperty property**.

Here is an example of how to create a template property that will handle the selection.

```
collection: widget.property({
    type: 'datasource'
}),
template: widget.property({
    type: 'template',
    templates: [{
        name: 'Template 1',
        template: '<h1>{{name}}</h1>'
    }, {
        name: 'Template 2',
        template: '<h1>{{name}}</h1><p><b>{{revenues}}</b></p>'
    }, {
        name: 'Template 3',
        template: '<h1>{{name}}</h1><p><strong>{{revenues}}</strong><br />From: <a href="mailto:{{email}}">{{email}}
    }],
    datasourceProperty: 'collection'
})
```

templates object

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

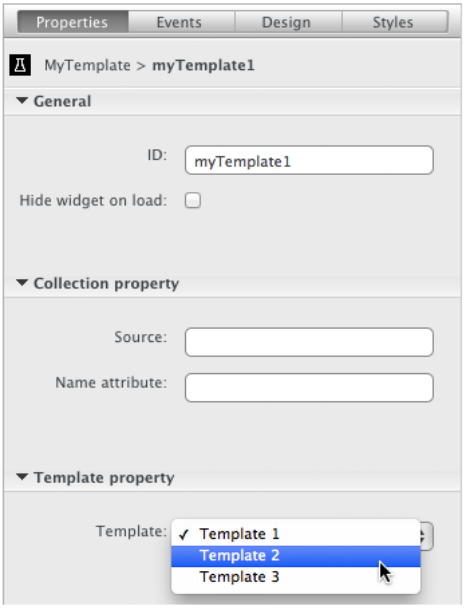
Property	Description
name	Display name for your template
template	String that defines the template with each attribute defined within {{ and }} characters. No spaces are allowed in the attribute name.

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.

Properties

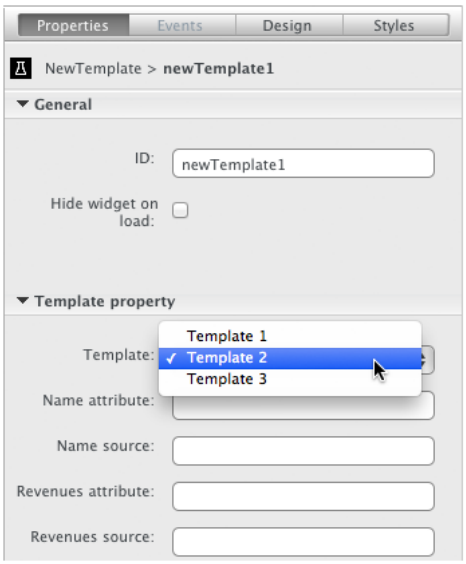
In the **Properties** tab, the templates you defined in the **templates object** appear as shown below.

For each attribute defined in your template, the datasourceProperty section displays a field for each attribute. For example, the "Name attribute" property comes from the {{name}} attribute defined in the *template* object.



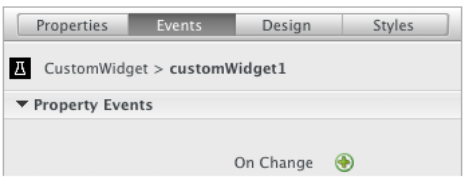
If you did not define a datasource for your template property, two fields appear for each attribute that you defined in your template. One field for the datasource and one field for the static value.

In our example below, we have two fields for each of the two attributes: name and revenues:



Events tab

For this property, you can intercept the **On Change** event in the Prototyper's **Events** tab:



This event is fired when the value for the property has been modified.

event object for onChange event

In the event object are the following properties:

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

Defining events

You can define two types of event for your custom widget that can be intercepted by the user of your custom widget:

- DOM event
- Custom event

For both types of events, you must use first declare it for your custom widget by using the **mapDomEvents( )** function in your custom widget's "widget.js" file.

To customize how the event will be displayed in the **Events** tab for your custom widget, use the **addEvent( )** or **addEvents( )** functions in your custom widget's "designer.js" file.

You can change the order of the events in the **Events** tab by using the **orderEvents( )** function. Use the **removeEvent( )** function to remove an event from the **Events** tab.

***Note:** By adding a property to your custom widget, the **On Change** event is added automatically in the **Events** tab (in the Prototyper). See the paragraph below to find out more about the onChange event.*

If, however, you declare an event using jQuery, it will not be in the Prototyper's **Events** tab. See the example below.

Refer to **Widget events** for more information about interacting with the Prototyper's **Events** tab.

DOM events

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

For more information about DOM event properties and methods, refer to **DOM event properties and methods**.

onChange event object

In the **On Change** event, an *event* object is returned. Here are a few of the important properties:

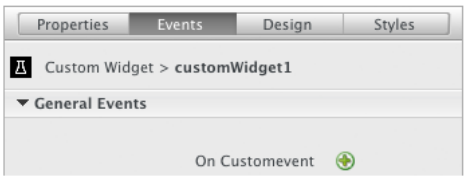
Property	Description
data.oldValue	Widget's previous value
data.value	Widget's current value
kind	Event type. In this case, "change".
parentEvent	Event that is the parent of this event
target	Property that was changed

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:

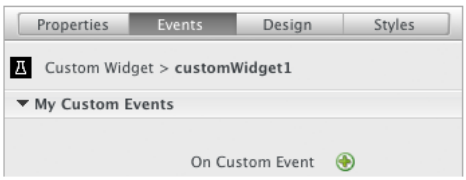
```
CustomWidget.mapDomEvents( { 'click': 'customevent' } );
```

Your "customevent" appears as shown below in the Prototyper:



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'
});
```



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');
};
```

You can customize how the event appears in the Prototyper 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.

Example

You can also declare a DOM event using jQuery that will not have a way to intercept it in the Prototyper:

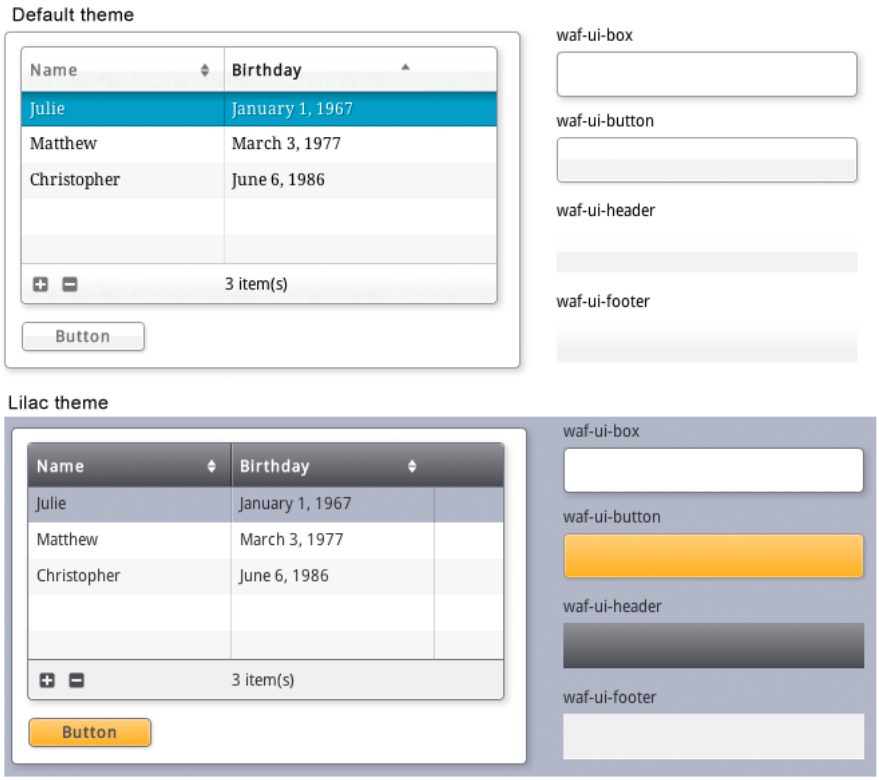
```
CustomWidget.prototype.init = function() {
    $(this.node).on('click', function(event) {
      // do something
    }.bind(this));
};
```

Using Wakanda's generic CSS classes

When you create a custom widget, you can use a few of Wakanda's generic CSS classes:

- waf-ui-box
- waf-ui-button
- waf-ui-header
- waf-ui-footer
- waf-ui-table

These classes are defined per theme and therefore the widget appears differently when you change the page's theme. For example, on the left are Wakanda's widgets and on the right are custom widgets with Wakanda's generic CSS classes added to each one.



waf-ui-box

The **waf-ui-box** class inherits the classes defined for a Wakanda's **Container** widget.

waf-ui-button

The **waf-ui-button** class inherits the classes defined for a Wakanda's **Button** widget.

waf-ui-header

The **waf-ui-header** class inherits the classes used for the header portion of Wakanda's **Grid** widget.

waf-ui-footer

The **waf-ui-footer** class inherits the classes used for the footer portion of Wakanda's **Grid** widget.

waf-ui-table

The **waf-ui-table** class is for all widgets that use the <table> tag.

designer.js

In the "designer.js" file, you define the way your custom widget appears in the Wakanda Studio by defining the style settings to display and how properties, previously defined in the "widget.js" file, appear.

You can customize the following tabs for your custom widget:

- Properties,
- Events,
- Design, and
- Styles.

***Important Note**: You must close and reopen any open Page Prototypes in the Prototyper after modifying the "designer.js" file. You might also need to recreate the instance of your custom widget on the Page because, depending on the modifications you make, the widget might not be updated accordingly.*

Defining the widget in the designer.js file

To define the widget's properties, styles, and overall design in the Prototyper, you can do so in the widget's "designer.js" file with the help of the functions in the **Studio** class.

By default, the following lines of code are included in the "designer.js" file:

```
(function(CustomWidget) {
        //Customize properties (previously defined in the widget.js file)
        //Add events to the Events tab (previously defined in the widget.js file)
        //Define settings for the Styles and Design tabs
});
```

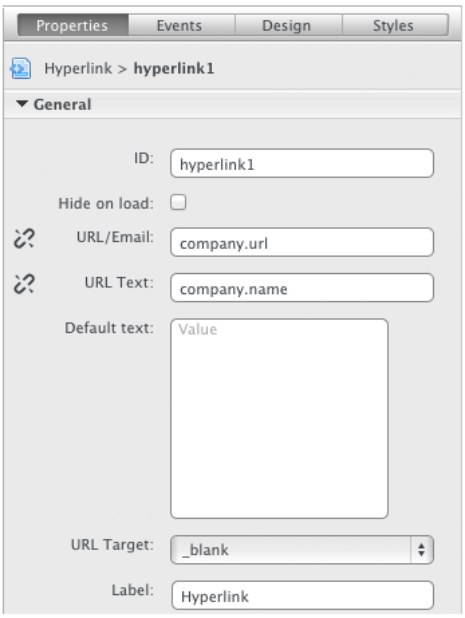
Widget properties

Every widget has two properties defined by default:

- ID
- Hide on load

To add properties to your custom widget, you must do so in the "widgets.js" file. Afterwards, you can customize them using the **customizeProperty( )** function in the "designer.js" file.

Below is an example of our custom widget's **Properties** tab:



Customizing properties

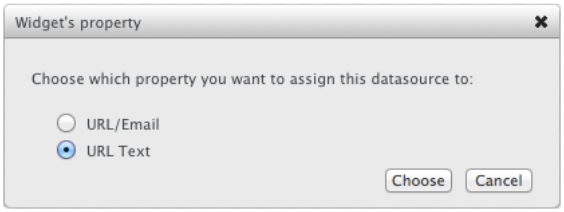
After you add a property in the "widget.js" file by using **addProperty( )**, you can then customize it in the "designer.js" file by using the **customizeProperty( )** function. For more information, refer to **Defining a widget's properties**.

```
/*Customize existing properties*/
Hyperlink.customizeProperty('url', {
    title: 'URL/Email',
    description: 'A datasource containing a URL or email.',
    display: true,
    sourceDisplay: true
});
Hyperlink.customizeProperty('urlText', {
    title: 'URL Text',
    description: 'A datasource containing the text to display for the hyperlink.',
    display: true,
    sourceDisplay: true
});
Hyperlink.customizeProperty('defaultText', {
    title: 'Default text',
    multiline: 'true',
    description: 'The default text to display if the URL Text is blank.',
    display: true,
    sourceDisplay: false
});
```

```
Hyperlink.customizeProperty('target', {
    title: 'URL Target',
    description: 'Target for the URL.'
});
```

Multiple bindable properties

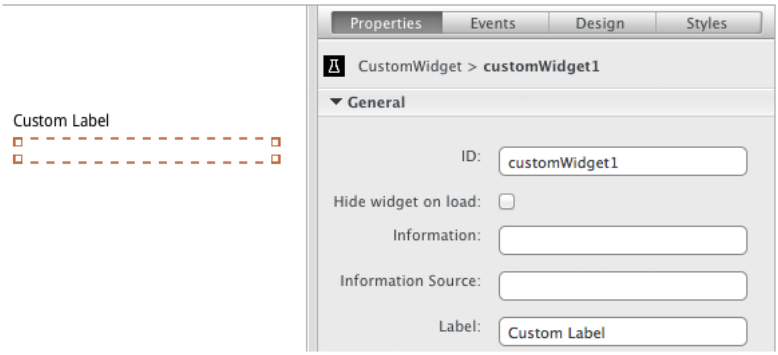
When you have more than one bindable property defined for your custom widget, the following dialog appears so that you can select the datasource property to which the datasource will be bound when you drop a datasource on top of the custom widget:



Label property

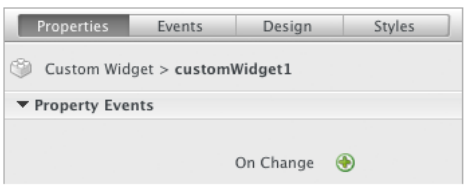
If you want to define a label property for your widget, which is managed by Wakanda, you can use the **addLabel( )** function. This property adds a Label widget to your widget if a value is defined for it.

In the Prototyper, the **Label** property appears on the **Properties** tab. When a value is entered in the Label property, a Label widget is created and attached to the custom widget.



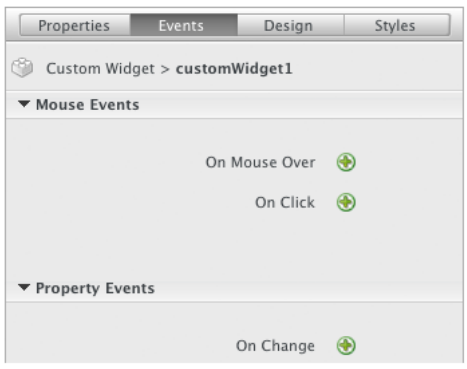
Widget events

By default, your custom widget has no events until you create at least one property. In this case, the "On Change" event is added:



To add an event, the **mapDomEvents( )** function in the "widget.js" file. Afterwards, you can add the same event to the **Events** tab by using the **addEvent( )** or **addEvents( )** functions.

Below is an example of our widget's **Events** tab:



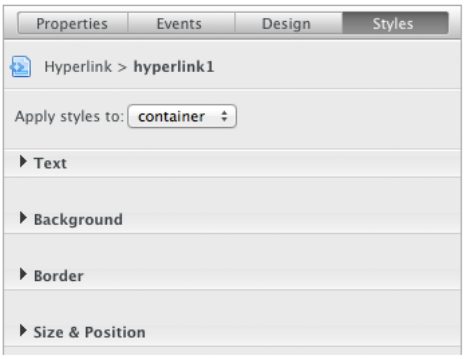
Here is the code associated to this example:

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

In the **widget.js** file, you declare the events. For more information, refer to .

Widget styles

By default, no styles are added to your widget. To define the sections available in the "Design" and "Styles" tabs, use the **setPanelStyle( )** function.
The styles appear as shown below (in our case, a value was entered in the Label property):



Here is the code we used to define the styles for our custom widget:

```
CustomWidget.setPanelStyle({
        'fClass': true, //This property is for the design panel
        'text': true,
        'background': true,
        'border': true,
        'sizePosition': true,
        'label': true,
        'disabled': ['border-radius']
    });
```

Widget's default size

The **setWidth( )** and **setHeight( )** functions allow you to define the widget's default width and height when it is added to a Page. By default, the size is 200px by 200px.
For example, you can write the following to set the width to 200 pixels and the height to 20 pixels:

```
CustomWidget.setWidth('200');
CustomWidget.setHeight('20');
```

Widget's display name

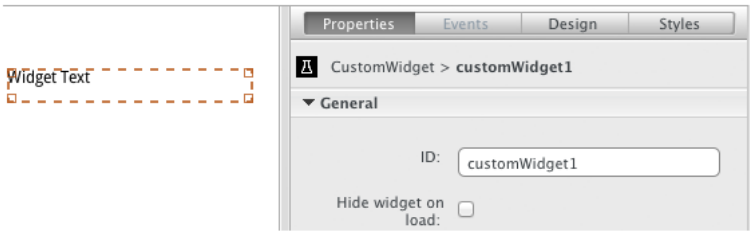
To define your widget's display name, you modify the "studioName" property in the **package.json** file. This name can be different from the widget's name that you defined initially. For example, if your widget's name (its class) is "CustomWidget", you can rename the widget to be "Custom Widget". This name will appear in the **Widgets** list, **Overview**, and **Properties** tab.

Widget's init

In this file, you can override the custom widget's initialization by writing code in this function:

```
CustomWidget.prototype.init = function() {
        this.node.innerHTML = "Widget Text";         /* Include text inside the widget */
    }
```

In our example, the "Widget Text" will display inside the custom widget:



widget.js

In the "widget.js" file, you define how your custom widget looks and behaves at runtime.
In this file, you can:

- Define properties
- Modify the widget's HTML tag
- Create events
- Create public and private functions

For more information about creating events for your custom widget, refer to **Defining events**.

widget.js template

By default, the following lines of code are included in this file. In our example, "CustomWidget" is the widget's name:

```
WAF.define('CustomWidget', ['waf-core/widget'], function(widget) {
    var CustomWidget = widget.create('CustomWidget', 'parentWidget', {

        /* Modify tag */
        tagName: 'li',

        init: function() {
            /* Define custom events */
        },
        /* Create functions for the widget */
        publicWidgetFunction: function() {},
        _privateWidgetFunction: function() {},

        /* Create properties */
        propertyOne: widget.property(),
        propertyTwo: widget.property({
            defaultValue: 'first'
        })
    });
});
```

You can also create the widget as shown below:

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

    CustomWidget.prototype.init = function() {};

    /* Create functions for the widget */
    CustomWidget.prototype.publicWidgetFunction = function() {};
    CustomWidget.prototype._privateWidgetFunction = function() {};

    /* Modify tag */
    CustomWidget.tagName = 'li';

    /* Create properties */
    CustomWidget.addProperty('propertyOne');
    CustomWidget.addProperty('propertyTwo', {
        defaultValue: '#first'
    });
});
```

WAF.define()

The **WAF.define()** function accepts three parameters:

Parameter	Type	Description
widgetName	String	Widget's name
requires	Array	Behaviors to require for the custom widget, e.g., ['waf-core/widget', 'waf-behavior/layout/container']
widgetFunction	Function	Function defining the custom widget, which includes the **widget.create()** function

widget.create()

The **widget.create()** function accepts three parameters:

Parameter	Type	Description
widgetName	String	Widget's name
widgetParentName	String	Widget's parent name (optional)
definition	Object	Define the widget's tag, init function, public and private functions, and properties

For more information on creating a function for your custom widget, refer to **Creating a function for your custom widget**.

widget.property()

The **widget.property()** function allows you to define a property for the widget.

Parameter	Type	Description
widgetName	String	Widget name
options	Object	Options for the property (for more information, see **options property**)

Modifying the widget's HTML tag

You can modify the custom widget's HTML tag, which is "div" by default, by using the **tagName** property.
For example, you can write the following to change the "div" tag to an "input" tag:

```
WAF.define('CustomWidget', ['waf-core/widget'], function(widget) {
    var CustomWidget = widget.create('CustomWidget', {
        init: function() { },
        test: widget.property({
            onChange: function(newValue) {
                this.node.innerHTML = this.test();
            }
        })
    });
    CustomWidget.tagName = 'input';
    return CustomWidget;
});
```

Or

```
WAF.define('CustomWidget', ['waf-core/widget'], function(widget) {
    var CustomWidget = widget.create('CustomWidget', {
        init: function() { },
        tagName = 'input',
        test: widget.property({
            onChange: function(newValue) {
                this.node.innerHTML = this.test();
            }
        })
    });
    return CustomWidget;
});
```

Creating a function for your custom widget

In order to create a function for your custom widget, you can do so by declaring it in the **widget.create()**'s *options* object:

```
publicWidgetFunction: function() {}
```

You can also declare a function outside of the object:

```
CustomWidget.prototype.myFunction = function()
{
       // do what you'd like here
}
```

This function can be accessed at runtime, by writing:

```
$$('customWidget1').myFunction(); //call the function on the instance of your widget
```

You can also create instance or class methods for your widget by using the functions in the **Methods Helper** category.
***Note**: Use an underscore as a prefix to your function's name to specify that it is private.*

widget.css

In the "widget.css" file, you can insert any CSS code you'd like to define for your widget by default.

Default CSS styles no longer added by default

Wakanda added the "waf-customwidget" class (if the widget's name was "CustomWidget") by default for any new widget and added directly to the custom widget's "widget.css" file. When the widget was added to the Page, the "waf-widget" and "waf-customwidget" classes were added to the custom widget.

As of Wakanda 11, neither one of these classes is added. For compatibility reasons, you can add them by using the **addClass( )** function.

Either directly on the widget's class:

```
CustomWidget.addClass('waf-widget');
CustomWidget.addClass('waf-customwidget');
```

Or in the init phase of the custom widget's "widget.js" file:

```
init: function() {
    this.addClass('waf-widget');
    this.addClass('waf-customwidget');
}
```

***Note**: It's not necessary to add the waf-widget class if you have not taken advantage of its styles. You can add only your custom widget's class if you prefer.*

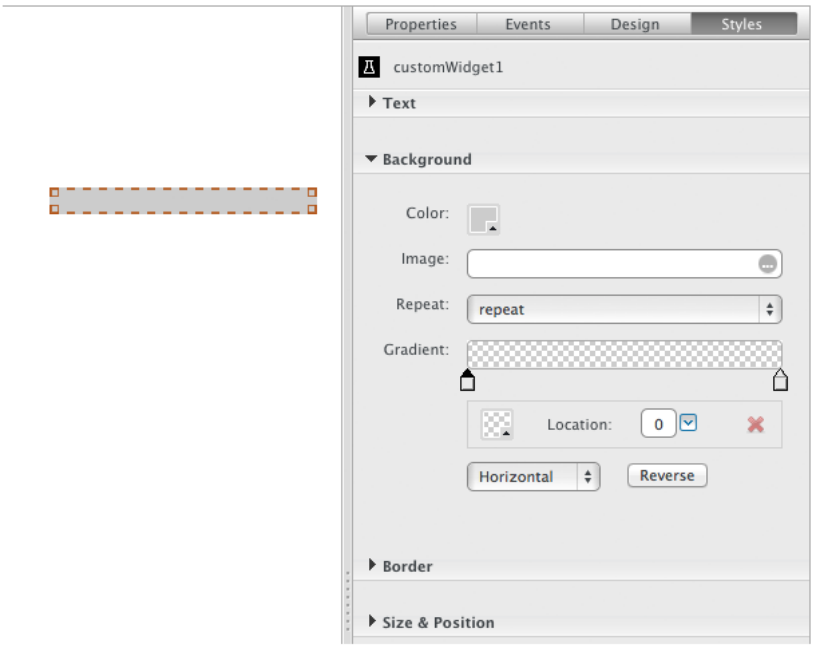
Defining default style settings

To define the default style settings for your custom widget, you do so in your custom widget's "widget.css" file:

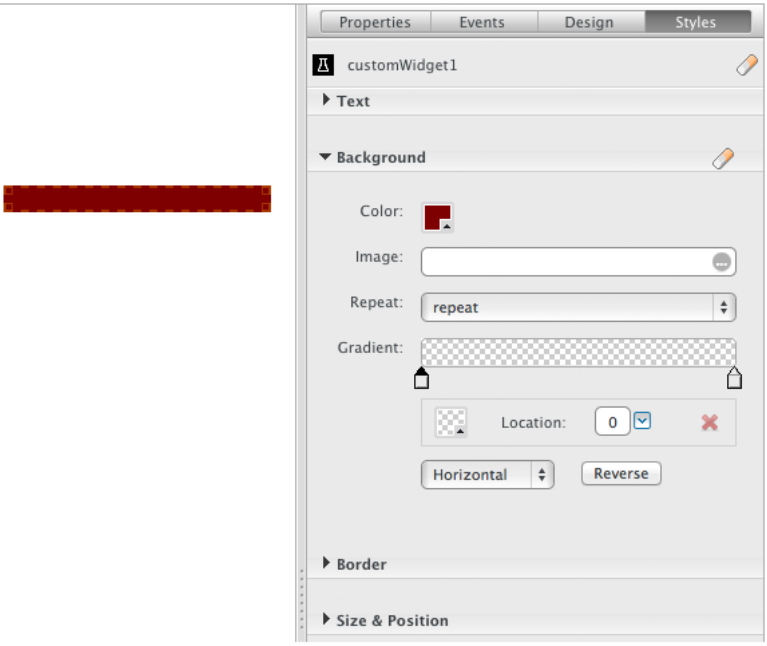
```
.mywidgetstyle {
    background-color: #ccc;  /*  add any CSS information here  */
}
```

You must also add the CSS class to your widget using the **addClass( )** function.

In the above example, your custom widget's background color is grey (#ccc). When you drag the widget to the Page, your custom widget will have a grey background. In the **Styles** tab, that setting will also be set as shown below:



If your user modifies the color in the **Styles** tab, it will be taken into account. The eraser icon appears next to the setting to show that it can be reverted to the original setting, which you defined in the "widget.css" file.



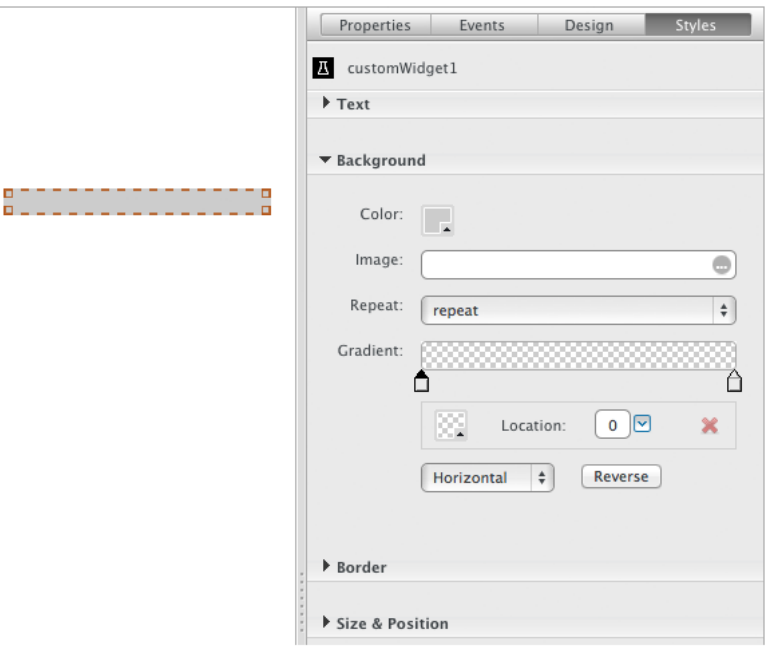
The modifications made to the custom widget on the Page will be added to the Page's CSS file.

Defining default style settings

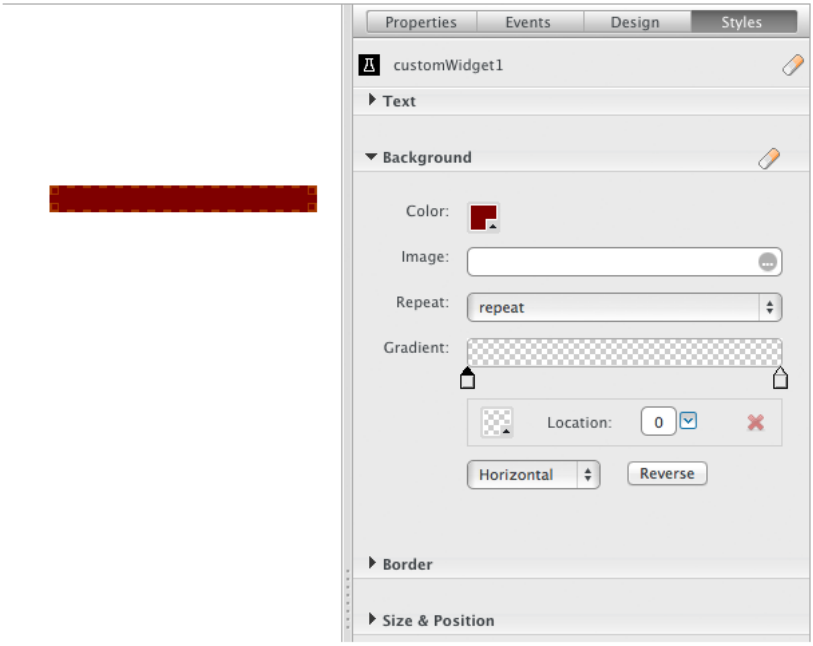
To define the default style settings for your custom widget, you could do so in the predefined class in your custom widget's "widget.css" file:

```
.waf-widget.waf-customwidget {
    background-color: #ccc;  /*  add any CSS information here  */
}
```

In the above example, your custom widget's background color is grey (#ccc). When you drag the widget to the Page, your custom widget will have a grey background. In the Styles tab, that setting will also be set as shown below:



If your user modifies the color in the Styles tab, it will be taken into account. The eraser icon appears next to the setting to show that it can be reverted to the original setting, which you defined in the "widget.css" file.



The modifications made to the custom widget on the Page will be added to the Page's CSS file.

Default style settings for the Label widget

If you have a Label widget that you added to your widget by using the **addLabel( )** function, you can customize the Label widget's default style settings in your custom widget's "widget.css" file.

You must use the following CSS syntax (in our case, the widget's name is "CustomWidget":

```
.waf-widget.waf-label.waf-label-customwidget {
    font-size: 22px;
    color: #c30;        /*  add any CSS information here  */
}
```

***NOTE**: For the moment, the Label widget is currently using the v1 architecture and so the classes are added to it by default even if you create a custom widget that uses the v2 architecture.*

Defining default style settings for states

If you define any states (or selectors) for your widget by using the **addState( )** or **addStates( )** functions, you can customize the CSS information directly in your custom widget's "widget.css" file.

```
.mywidget:active { background: #c30; }
.mywidget.mystate1 { background: #ccc; }
.mywidget.waf-state-mystate2 { background: #ccc; }
```

In the above case, we added these states to our widget's "mywidget" class in our widget's "designer.js" file:

```
CustomWidget.addStates(':active','.mystate1','mystate2');
```

***Note**: If you do not add a ":" or a ".", the prefix "waf-state-" will be added to the class name.*

In your widget's "widget.js" file, you define the "mywidget" class as shown below:

```
CustomWidget.addClass('mywidget');
```

package.json

This file describes general information about your custom widget as well as the files necessary to build the custom widget (in the **loadDependencies array**).

package.json file

The JSON object for a custom widget by default is as follows:

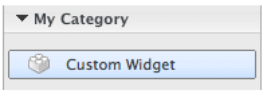
```
{
    "name": "CustomWidget",
    "type": "widget",
    "studioName": "CustomWidget",
    "category": "Custom Widgets",
    "iconPath": "/icons/widget.png",
    "author": "Widget Developer",
    "contributors": [
        "Developer 1",
        "Developer 2"
    ],
    "version": "1.0.0",
    "copyright": "(c) 2015 Widget Developer",
    "repository": {
        "type": "git",
        "url": "https://github.com/developer/Widget.git"
    },
    "keywords": ["wakanda", "widget"],
    "engines": {"wakanda": ">= 11"},
    "license": "MIT",
    "externalWidgets": [ ],
    "loadDependencies": [
        {"file": "widget.js"},
        {"file": "css/widget.css"},
        {"file": "designer.js", "studioOnly": true}
    ]
}
```

Customizing the widget

In this package.json file, you can define the following attributes:

- Display name,
- Category, and
- Icon

Here are the results after we modified the three properties (studioName, category, and iconPath) as shown in the JSON object above:



Defining your widget's platform

In the package.json file, you can define the platform(s) for your custom widget, which can be desktop, smartphone, and/or tablet. If this property is missing, the custom widget is available for all platforms in the Prototyper.

To do so, use the *studioPlatform* property and define in an array the platform(s) as shown below:

```
"studioPlatform": ["smartphone", "desktop", "tablet"]
```

The following platforms are possible:

Platform	Description
smartphone	For smartphone pages
desktop	For desktop pages
tablet	For tablet pages

If you want your widget to only be used on the desktop, you can write the following:

```
"studioPlatform": ["desktop"]
```

***Note**: The value defined in the **studioPlatform** property is the same as the **suffix** property in the targets.json file. For more information, refer to **Routing Pages**.*

External libraries and other dependencies

In the *loadDependencies* array, you can:

- add other libraries to your custom widget,
- include another widget to your custom widget, and
- share a library between widgets

During the installation process, however, you must define which widgets are necessary for the proper functioning of your custom widget by specifying its depository in the *externalWidgets* property.

Adding other libraries to your custom widget

If you want to add other libraries to your widget, you can include them in this file.

For example, if you have a "WidgetLibrary" folder containing two files ("code.js" and "styles.css") that your widget needs, you can write the following:

```
"loadDependencies": [
```

```
        {"file": "widget.js"},
        {"file": "designer.js", "studioOnly": true},
        {"file": "/WidgetLibrary/code.js"}, //custom library's code.js file
        {"file": "/WidgetLibrary/styles.css"} //custom library's css file
    ]
```

Adding another custom widget

If you want to add another custom widget that has already been installed in the "Widgets" folder, you can do so in your widget's package.json file. For example, if you have a "TestWidget" widget that you want to add to your "Custom Widget", you can write the following:

```
"loadDependencies": [
        {"file": "widget.js"},
        {"file": "css/widget.css"},
        {"file": "designer.js", "studioOnly": true},
        {"package": "TestWidget"}
    ]
```

Sharing a library between widgets

If you want to share a library between multiple widgets, you can do the following:

1. Create a folder in your Custom Widgets folder. In our example, we call it "libs".
2. Add your external script JS files to this folder. We have included only one for our example, "externalScript.js".
3. In the library folder, create a package.json folder that defines the files in the folder that you want to use:

```
{
    "name": "libs",
    "loadDependencies": [
        {"file": "/libs/externalScript.js", "runtimeOnly":true}
    ]
}
```

In your custom widget's package.json file. you indicate this package as shown below in the last object in the "loadDependencies" array:

```
"loadDependencies": [
    {"file": "widget.js",
    {"file": "/css/widget.css"},
    {"file": "designer.js", "studioOnly": true},
    {"file": "/libs/package.json", "runtimeOnly": true}
]
```