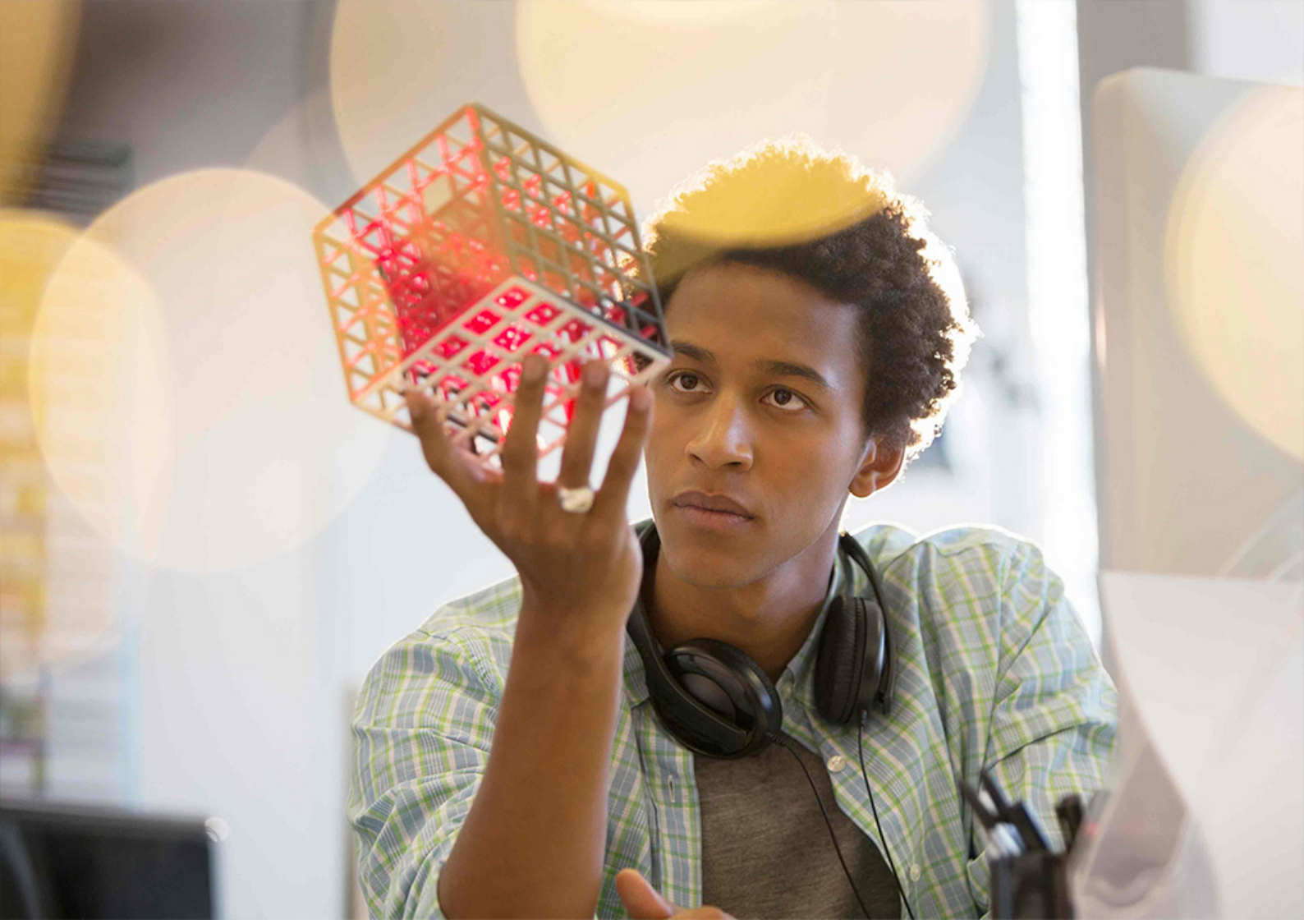

PUBLIC

Document Version: 2021.20 – 2021-11-16

SAP Analytics Cloud Custom Widget Developer Guide

THE BEST RUN SAP

Content

1 Introduction

The SAP Analytics Cloud Custom Widget framework lets you extend the predefined set of widgets provided by analytics designer with your own custom widgets.

This is very useful, for example, if you need a specific user interface element, a particular visualization of data, or a certain functionality in your analytic application that is not provided by the predefined set of widgets.

Custom widgets seamlessly integrate into SAP Analytics Cloud, analytics designer. Like any other predefined widget, a custom widget provides the following capabilities:

- It is listed in the widget menu list from where you can add it to the canvas.
- It can be moved and resized on the canvas.
- It appears in the widget outline.
- It can contribute script methods to the analytics designer script language.
- It can provide areas in analytics designer where you can set property values of the custom widget at design time (Styling Panel and Builder Panel).

Custom widgets are technically a simple set of files. The most basic custom widget consists of simply a JSON file and a JavaScript file. More complex custom widgets add more files to this (JavaScript files, CSS files, images, and so on).

To create custom widgets, you don't need special software. At the beginning, a simple text editor will do. However, when your custom widgets get more complex, a good JavaScript text editor will make your development efforts easier.

2 Web Components

Custom widgets are implemented as Web Components.

The basic idea of Web Components is to provide custom HTML elements (so-called custom elements) in the HTML DOM (Document Object Model) of a web page that do not interfere with the rest of the HTML DOM. In fact, rendering and styling of custom elements is strictly isolated from the remaining HTML DOM. This is achieved by separating a custom element's HTML DOM from the HTML DOM of the web page in a shadow DOM.

Web Components are made of HTML, CSS (optional), and JavaScript.

This document doesn't assume any prior knowledge of Web Components. It explains the relevant ideas and details of Web Components in the text, along with sample code.

For more information on Web Components, see the Mozilla MDN web docs on Web Components.

> **i** Note
>
> The sample code in this document is based on ECMAScript 2015.

3 Restrictions

There are restrictions for custom widgets that you should take into account.

Custom widgets currently have the following restrictions:

- Custom widgets work in Google Chrome and Microsoft Edge (version 79 and higher) only.

4 Hosting Custom Widgets

From a hosting point of view, a custom widget consists of two types of files: the custom widget JSON file (explained in more detail in a later section) and the resource files.

The custom widget JSON file contains the metadata of a custom widget. It defines all the ingredients of a custom widget and references its resource files by their URLs.

The resource files are all the files of the custom widget that make it work properly, for example, JavaScript files, CSS files, HTML files, image files, and so on.

To host a custom widget, you need two locations on which these files are distributed:

- An SAP Analytics Cloud instance, on which you upload the custom widget JSON file of the custom widget (see chapter Adding a Custom Widget to Analytics Designer [page 9]).
- An HTTP Web server with HTTPS enabled, on which you upload the resource files. The Web server acts as a simple repository of the resource files. The resource files are static, they aren't processed or executed on the Web server. Recall that the custom widget JSON contains URL references that point to these resource files.

When a custom widget is rendered in a Web browser, the Custom Widget framework passes the URL of a resource file referenced in the custom widget JSON to the browser. The browser requests that resource file from the Web server. The Custom Widget framework poses no specific demands on the Web server in so far as a resource file should simply be served to any browser that requests the resource file by the appropriate URL. The Custom Widget framework doesn't provide any authentication or authorization capabilities, session cookie support, and so on.

The resource files of a custom widget are requested from the Web server both while creating an analytic application containing this custom widget in analytics designer (design time) and while running the analytic application containing this custom widget in the browser (runtime). Of course, any resource file related to the Builder and Styling Panel of a custom widget is requested only at design time.

The Web server can be either a public or a private (that is, company internal) Web server, as long as it serves the resource files as requested from a browser by the appropriate URL. A private Web server can be reached only from within the company and is useful for internal development of custom widgets. For internal development, you can run, for example, an Apache Web server or a node.js server with the module "http-server" on your local development system.

> **i Note**
>
> - The SAP Analytics Cloud instance never connects to the Web server to request resource files. It is always the browser that requests resource files.
> - When you save an analytic application in analytics designer on the SAP Analytics Cloud instance, only a reference to the custom widget is saved with the analytic application. Specifically, it is a reference to the custom widget JSON of the custom widget on the SAP Analytics Cloud instance. Analytic applications never store resource files of custom widgets with them.

5 Quickstart - Creating a New Custom Widget

To create your own custom widget, copy a simple sample custom widget, modify it, and add it to the analytics designer.

Prerequisites

You need a web server that hosts the resources of your custom widget (JavaScript files, CSS files, images, and so on) (see chapter Hosting Custom Widgets [page 6] for more information). In the samples below, let's assume that your custom widget resources are hosted on:

> **Example**
>
> https://www.sample.com/customwidgets

5.1 Getting a Sample Custom Widget

To create a simple sample custom widget as a starting point, create a folder `coloredbox` and place the following files into it:

- `coloredbox.json` (find the source code in chapter Custom Widget JSON [page 32])
- `coloredbox.js` (find the source code in chapter Web Component JavaScript [page 36])
- `coloredbox_styling.js` (find the source code in chapter Web Component JavaScript of the Styling Panel [page 41])
- `coloredbox_builder.js` (find the source code in chapter Web Component JavaScript of the Builder Panel [page 45])
- `icon.png` (any 16 x 16 pixel icon will do)

5.2 Modifying the Sample Custom Widget

Modify the sample **Colored Box** custom widget to the **Box** custom widget in the following steps:

Procedure

1. Rename the file `coloredbox.json` to `box.json`.
2. In the file `box.json`, edit the following property values:

JSON Property Name	Old JSON Property Value	New JSON Property Value
`id`	`"com.sap.sample.coloredbox"`	`"com.sample.box"`
`name`	`"Colored Box"`	`"Box"`
`newInstancePrefix`	`"ColoredBox"`	`"Box"`
`icon`	`"https://.../icon.png"`	`"https://www.sample.com/customwidgets/box/icon.png"`
`vendor`	`"SAP"`	`"Sample"`
`tag`	`"com-sap-sample-coloredbox"`	`"com-sample-box"`
`url`	`"https://.../coloredbox.js"`	`"https://www.sample.com/customwidgets/box/box.js"`
`tag`	`"com-sap-sample-coloredbox-styling"`	`"com-sample-box-styling"`
`url`	`"https://.../coloredbox_styling.js"`	`"https://www.sample.com/customwidgets/box/box_styling.js"`
`tag`	`"com-sap-sample-coloredbox-builder"`	`"com-sample-box-builder"`
`url`	`"https://.../coloredbox_builder.js"`	`"https://www.sample.com/customwidgets/box/box_builder.js"`
`description`	`"Called when the user clicks the Colored Box."`	`"Called when the user clicks the Box."`

3. Rename the file `coloredbox.js` to `box.js`.
4. In the file `box.js`, edit the following locations:

Old Text	New Text
`class ColoredBox extends`	`class Box extends`
`customElements.define("com-sap-sample-coloredbox", ColoredBox);`	`customElements.define("com-sample-box", Box);`

5. Rename the file `coloredbox_styling.js` to `box_styling.js`.
6. In the file `box_styling.js`, rename the following locations:

Old Text	New Text
`<legend>Colored Box Properties</legend>`	`<legend>Box Properties</legend>`
`class ColoredBoxStylingPanel extends`	`class BoxStylingPanel extends`
`customElements.define("com-sap-sample-coloredbox-styling", ColoredBoxStylingPanel);`	`customElements.define("com-sample-box-styling", BoxStylingPanel);`

7. Rename the file `coloredbox_builder.js to box_builder.js`.
8. In the file `box_builder.js`, rename the following locations:

Old Text	New Text
`<legend>Colored Box Properties</legend>`	`<legend>Box Properties</legend>`
`class ColoredBoxBuilderPanel extends`	`class BoxBuilderPanel extends`
`customElements.define("com-sap-sample-coloredbox-builder", ColoredBoxBuilderPanel);`	`customElements.define("com-sample-box-builder", BoxBuilderPanel);`

9. On your web server, create a folder `box` which can be reached with this URL: `https://www.sample.com/customwidgets/box`.
10. Upload the files `box.js`, `box_styling.js`, `box_builder.js` and `icon.png` to your web server to folder `https://www.sample.com/customwidgets/box`.

5.3 Adding a Custom Widget to Analytics Designer

To add a custom widget to SAP Analytics Cloud, analytics designer, follow these steps:

Prerequisites

Please note that you need the appropriate permission to create and upload custom widgets. For more information, please see chapter “Granting Permissions for Custom Widgets” in SAP Analytics Cloud help on SAP Help Portal at http://help.sap.com.

Procedure

1. On the *Analytic Applications* start page, choose the *Custom Widgets* tab.
2. Click the + *Create* toolbar icon.
3. In the *Upload File* dialog, click the *Select File* button.
4. Select the custom widget file, for example `box.json`.
5. Create a new analytic application. The custom widget is listed in the widget list of the dropdown menu of the *Add* menu (+) in analytics designer.

Next Steps

You can add multiple major versions of a custom widget to analytics designer.

Every custom widget has a version number in the format `majorVersion.minorVersion.patchVersion`, for example, `1.0.0`. You can add custom widgets with different major versions side-by-side to analytics designer, for example, versions `1.0.0` and `2.0.0` of a custom widget.

When you add a custom widget that differs in the minor version with one present in analytics designer, then the added version replaces the present version. For example, if a custom widget of version `1.5.0` is present in analytics designer, then adding either version `1.4.0` or `1.6.0` replaces version `1.5.0`.

5.4 Removing the Custom Widget from Analytics Designer

To remove the custom widget from analytics designer, follow these steps:

Procedure

1. Navigate to *Main Menu* > *Browse* > *Custom Widgets*.
2. Select *Box*.
3. Click the *Delete* toolbar icon (trashcan).
4. Confirm the deletion.

6 Custom Widgets

A custom widget consists of the following files:

File	Required/Optional	Description
Custom widget JSON	Required	Defines the custom widget
Web Component JavaScript	Required	Implements the custom element of the custom widget (Web Component)
Web Component JavaScript of Styling Panel	Optional	Implements the custom element of the Styling Panel of the custom widget (Web Component)
Web Component JavaScript of Builder Panel	Optional	Implements the custom element of the Builder Panel of the custom widget (Web Component)
Icon	Optional	Represents the custom widget's icon (16 x 16 pixels)
More files...	Optional	More files (JavaScript, CSS, images, and so on) depending on the implemented custom widget

The following sections explain these files in detail.

6.1 Custom Widget JSON Reference

The custom widget JSON file specifies the custom widget.

The following sections explain the valid properties of the custom widget JSON. Any other properties present in a custom widget JSON render it invalid

Root Object

The root object of the custom widget JSON specifies the custom widget. Its JSON properties are:

Property	Required/Optional	Type	Description
`id`	Required	string	Unique ID of the custom widget, for example, `"com.sap.sample.coloredbox"`

Property	Required/Optional	Type	Description
`version`	Required	string	Version of the custom widget, for example, `"1.0.0"`
`name`	Required	string	Name of the custom widget, for example, `"Colored Box"`
`description`	Optional	string	Description of the custom widget, which is used, for example, in tooltips
`newInstancePrefix`	Required	string	Prefix of a new custom widget instance in analytics designer, for example, `"ColoredBox"`
`icon`	Optional	string	URL of the custom widget's icon (16 x 16 pixels). **→ Tip** You can also use a Data-URL. A Data-URL lets you include a resource directly into the custom widget JSON that would be otherwise requested via the network.
`vendor`	Optional	string	Vendor string
`eula`	Optional	string	End-user license agreement text
`license`	Optional	string	License text
`webcomponents`	Required	Array of Webcomponent objects	Web components of the custom widget (see section below)
`properties`	Required	Properties object	Properties of the custom widget (see section below)
`methods`	Required	Methods object	Methods of the custom widget (see section below)
`events`	Required	Events object	Events of the custom widget (see section below)

Property	Required/Optional	Type	Description
`imports`	Optional	Array of string	Type libraries used by the custom widget (see chapter Importing Script API Data Types with Type Libraries [page 30])
supportsMobile	Optional	Boolean	Indicates whether the custom widget is rendered on mobile devices. If `false`, then the custom widget is replaced in an analytic application on a mobile device by a placeholder with a message explaining that the custom widget doesn't run on a mobile device. See Custom Widgets on Mobile Devices [page 50] for more information. Possible values are: • `true` • `false`

Webcomponent Object

A Webcomponent object specifies one Web Component of a custom widget. Its JSON properties are:

Property	Required/Optional	Type	Description
`kind`	Required	string	What kind of component this Web Component represents. Possible values are: • `"main"` (the actual custom widget) • `"styling"` (the Styling Panel of the custom widget) • `"builder"` (the Builder Panel of the custom widget)

Property	Required/Optional	Type	Description
`tag`	Required	string	Unique name of the custom element (its HTML tag name) of this Web Component. **i Note** The unique name must contain at least one hyphen (-) to differentiate a custom element from regular HTML elements. **→ Tip** Take the custom widget ID and replace all dots (.) with hyphens (-) to create a unique name for the custom element. For example, turn the custom widget ID `"com.sap.sample.coloredbox"` into the unique name `"com-sap-sample-coloredbox"` **→ Tip** Append the major version number to the tag name, for example, `"com-sap-sample-coloredbox-1"`. This avoids confusion between multiple versions of a custom widget that have been added side-by-side to analytics designer (see section Adding a Custom Widget to Analytics Designer [page 9]).
`url`	Required	string	URL to the Web Component JavaScript file

Property	Required/Optional	Type	Description
`type`	Optional	string	This property can help with the modularization of your JavaScript code. For more information, see the description of the `type` attribute with the value `module` in the Mozilla MDN webdocs on the script element.
`integrity`	Required	string	String containing the hash value of the Web Component JavaScript file. The hash value has the format `"hashAlgorithm-hashValue"`, for example, `"sha256-ih2Us20RAQ9CaWjbO/LfZPetSX7gTrVlTtbdF6mk4io="` Web browsers support several hash algorithms. For more information, see the Mozilla MDN web docs on Subresource Integrity. **→ Tip** To quickly obtain a hash value add your custom widget to analytics designer (setting the JSON property `ignoreIntegrity` to `false`). This operation fails, as expected, but the browser console lists the expected hash value.
`ignoreIntegrity`	Optional	Boolean	Indicates whether to ignore the property `integrity`, for example, during development of a custom widget. Possible values are: • `true` • `false`

Properties Object

A Properties object specifies the properties of a custom widget. Each JSON property name of a Properties object is the name of a custom widget property. Its value is an object with the following JSON properties:

Properties	Required/Optional	Type	Description
`type`	Required	string	Type of the custom widget property. Possible values are, for example: • `"string"` • `"number"` • `"boolean"` • `"integer"` • `"string[]"` • `"number[]"` • `"boolean[]"` • `"integer[]"` • You can also use certain script API data types. For more information see section Using Script API Data Types in JavaScript Functions [page 29]. • You can also use simple object types. For more information see section *Simple Object Types*.
`description`	Optional	string	Description of the custom widget property

Properties	Required/Optional	Type	Description
`default`	Optional	Depends on `type`	Default value of the custom widget property. Possible values are one of the following JavaScript types: • String, for example, `"green"` • Number, for example, `1` • Boolean, for example, `true` • An array of string values, for example, `["red", "yellow", "green"]` • An array of number values, for example, `[1, 2, 3]` • An array of Boolean values, for example, `[true, false, true]`

> **i Note**
>
> When you modify a property that is an array in a script method (see section *Methods Object*), always reassign the modified array to the property after the modification operation.
>
> For example, your custom widget has a property `listOfItems` of type `string[]`.To add a new item to this array in a script method, follow this pattern: Create a helper reference to the array, add the item, then reassign the helper reference to the array property, like this:
>
> `"body": "var list = this.listOfItems; list.push(newItem); this.listOfItems = list;"`

Methods Object

A Methods object specifies the script method of a custom widget. Each JSON property name of a Methods object is the name of a custom widget script method. Its value is an object with the following JSON properties:

Properties	Required/Optional	Type	Description
`description`	Required	string	Description of the script method

Properties	Required/Optional	Type	Description
`parameters`	Optional	Array of Parameter objects	Parameters of the script method (see section below). Omit this property if the script method has no parameters.
`body`	Optional	string	Implementation of the script method expressed in the analytics designer script language. **i Note** If you omit this property then the Custom Widget framework looks up and calls a JavaScript function of this name (a so-called native JavaScript function) in the Web Component JavaScript file whose property `kind` has the value `"main"`. If you change a custom widget property in a native JavaScript function, dispatch a `"propertiesChanged"` custom event to notify the Custom Widget framework of the change. For a code example, see the section Implementing Property Setter and Getter Methods Consistently [page 54]. Otherwise this may lead to outdated property values when you retrieve them in a script method.
`returnType`	Optional	string	Return type of the script method. Omit this property if the script method has no return value.

Parameter Object

A Parameter object specifies one parameter of a script method of a custom widget. Its JSON properties are:

Properties	Required/Optional	Type	Description
`name`	Required	string	Name of the parameter
`type`	Required	string	Type of the parameter. Possible values are, for example: • `"string"` • `"number"` • `"boolean"` • `"integer"` • `"string[]"` • `"number[]"` • `"boolean[]"` • `"integer[]"` • You can also use certain script API data types. For more information see chapter Using Script API Data Types in JavaScript Functions [page 29] . • You can also use simple object types. For more information see section *Simple Object Types*.
`description`	Optional	string	Description of the parameter

Events Object

An Events object specifies the events of a custom widget. Each JSON property name of an Events object is the name of a custom widget event. Its value is an object with the following JSON properties:

Properties	Required/Optional	Type	Description
`descripton`	Optional	string	Description of the event

Simple Object Types

You can also specify a simple object type. In addition to simple types (`string`, `number`, `integer`, `boolean`), script API data types, and arrays of these types, you can also specify a simple object type using the form `object<type>`.

This type represents a JavaScript object with keys of type `string` and values of type <type>. For example, the type `object<string>` represents a JavaScript object with keys of type `string` and values of type `string`.

The following example shows how to use an object of type `object<Button>`, which represents a JavaScript object with keys of type `string` and values of type `Button`:

Custom Widget JSON

Sample Code

```
{
    ...
    "properties": {
        "buttons": {
            "type": "object<Button>",
            "description": "A collection of Button objects"
        }
    },
    "methods": {
        "putButton": {
            "parameters": [
                {"name": "name", "type": "string"},
                {"button": "button", "type": "Button"}
            ],
            "body": "this.buttons[name] = button;",
            "description": "Adds a Button object, using a name, to the
collection."
        },
        "getButton": {
            "parameters": [
                {"name": "name", "type": "string"}
            ],
            "returnType": "Button",
            "body": "return this.buttons[name];",
            "description": "Returns a Button object, using a name, from the
collection."
        }
    }
}
```

Sample Analytics Designer Script

Sample Code

```
var button1 = BUTTON_1;
var button2 = BUTTON_2;
customWidget.put("b1", button1);
customWidget.put("b2", button2);
var button = customWidget.get("b2");
```

6.2 Web Component JavaScript

A custom widget is composed of one or more Web Components.

Each Web Component is implemented in a Web Component JavaScript file, which defines and registers a custom element and implements the custom element's JavaScript API.

For an example, see the section below that explains the Web Component JavaScript of the sample custom widget Colored Box.

Web Component Lifecycle

During the lifetime of a custom widget, the Custom Widget framework calls specific JavaScript functions of the Web Component in a defined order. You can implement these JavaScript functions to control the behavior of the custom widget. See the next section for a detailed explanation of each of these functions.

When the custom widget is rendered for the first time, the Custom Widget framework calls the following sequence of JavaScript function on the custom widget:

1. `constructor()`
2. `onCustomWidgetBeforeUpdate()`
3. Property setter functions, if present, to update the custom widget properties
4. `onCustomWidgetAfterUpdate()`
5. `connectedCallback()`

When the custom widget is updated, the Custom Widget framework executes the following sequence of JavaScript function calls on the custom widget:

1. `onCustomWidgetBeforeUpdate()`
2. Property setter functions, if present, to update the custom widget properties
3. `onCustomWidgetAfterUpdate()`

When the custom widget is removed from the canvas or the analytic application is closed, the Custom Widget framework executes the following sequence of JavaScript function calls on the custom widget:

1. `onCustomWidgetDestroy()`
2. `disconnectedCallback()`

> **i Note**
>
> Function `onCustomWidgetDestroy` is not called on custom widgets when their visibility is set to false. It is also not called when custom widgets are an invisible part of a container, for example, when in the non-visible Tab of a TabStrip.

When the custom widget is resized on the canvas, the Custom Widget framework executes the following JavaScript function call on the custom widget:

1. `onCustomWidgetResize()`

> **i Note**
>
> Dragging the Custom Widget on the Canvas:
>
> At design time, when you are about to drag a custom widget on the canvas of analytics designer, the custom widget is cloned to provide you an object to drag. Thus, the `constructor` of the custom widget is called, as well as the `connectedCallback` and `disconnectedCallback` callbacks.

Web Component JavaScript API

The following table lists the JavaScript API of a Web Component of a custom widget.

Function	Syntax	Description
`constructor`	`constructor()`	You can implement this function to execute JavaScript code when the Web Component is initialized, for example, when the custom widget is added to the canvas or the analytic application is opened. This function is the counterpart of `onCustomComponentDestroy`.

<table>
<tr><th>Function</th><th>Syntax</th><th>Description</th></tr>
<tr><td><code>onCustomWidgetBeforeUpdate</code></td><td><code>onCustomWidgetBeforeUpdate(oChangedProperties)</code></td><td>You can implement this function to execute JavaScript code before the properties of the custom widget are updated.

The argument <code>oChangedProperties</code> is a JavaScript object containing the changed properties as key-value pairs. The key is the name of the property, the value is the changed value of the property.

When this function is called for the very first time by the Custom Widget framework, the full list of properties is passed in <code>oChangedProperties</code>. The property value of a property is the default value if defined in the custom widget JSON or <code>undefined</code> otherwise.

→ Tip

For Web Components that implement the actual custom widget (with a value of <code>kind="main"</code> in the custom widget JSON), some more build-in properties are passed in <code>oChangedProperties</code> in the very first call, in addition to the properties defined in the custom widget JSON:

<table>
<tr><th>Property</th><th>Type</th><th>Description</th></tr>
<tr><td><code>widgetName</code></td><td>string</td><td>Name of the widget</td></tr>
<tr><td><code>designMode</code></td><td>Boolean</td><td>Indicates whether the custom widget currently runs in the analytics designer.</td></tr>
</table>
</td></tr>
</table>

Function	Syntax	Description

Property	Type	Description
		Possible values are:ol • `true` • `false`
Event name, for example, `onClick`	Boolean	Indicates whether this event of the custom widget has currently an event script assigned. Possible values are: • `true` • `false`

Function	Syntax	Description
`onCustomWidgetAfterUpdate`	`onCustomWidgetAfterUpdate(oChangedProperties)`	You can implement this function to execute JavaScript code after the properties of the custom widget have been updated. The argument `oChangedProperties` is a JavaScript object containing the changed properties as key-value pairs. The key is the name of the property, the value is the changed value of the property. When this function is called for the very first time by the Custom Widget framework, the full list of properties is passed in `oChangedProperties`. The property value of a property is the default value if defined in the custom widget JSON or undefined otherwise.

Function	Syntax	Description
`connectedCallback`	`connectedCallback()`	You can implement this function to execute JavaScript code when this Web Component of the custom widget is connected to the HTML DOM of the web page. This happens, for example, in the following situations: • at design time and runtime, when the custom widget is rendered for the first time • at design time, when you drag a custom widget on the canvas. The custom widget is cloned to provide an object to drag. • at design time, when you move a custom widget from one container to another, for example, from a tab strip to a panel or from one tab of a tab strip to another tab • at runtime, when you move a custom widget from one container to another by using the analytics designer script method `Widget.moveWidget()` **→ Tip** It's good practice to implement this function such that it may be called by the framework at any time, restoring the state of the custom widget. For simple custom widgets without much state data, it may be sufficient to recreate the custom widget just from scratch. See also the section on `disconnectedCallback()` below.
`onCustomWidgetDestroy`	`onCustomWidgetDestroy()`	You can implement this function to execute JavaScript code when the custom widget is destroyed, for example, when the custom widget is removed from the canvas or the analytic application containing the custom widget is closed. This function is the counterpart of `constructor()`.

Function	Syntax	Description
`disconnectedCallback`	`disconnectedCallback()`	You can implement this function to execute JavaScript code when this Web Component of the custom widget is disconnected from the HTML DOM of the web page. This happens, for example, in the following situations: • at design time and runtime, when you close the application • at design time, when you remove a custom widget from the canvas • at design time, when you drag a custom widget on the canvas. The custom widget is cloned to provide an object to drag. • at design time, when you move a custom widget from one container to another, for example, from a tab strip to a panel or from one tab of a tab strip to another tab • at runtime, when you move a custom widget from one container to another by using the analytics designer script method `Widget.moveWidget()` **→ Tip** It's good practice to implement this function such that it may be called by the framework at any time, saving the state of the custom widget – unless the custom widget is removed permanently. For simple custom widgets without much state data, saving their state may not be necessary. See also the section on `connectedCallback()` above.

Function	Syntax	Description
`onCustomWidgetResize`	`onCustomWidgetResize(width, height)`	You can implement this function to execute JavaScript code when the custom widget is resized. The argument `width` is the new width of the custom widget in pixels. The argument `height` is the new height of the custom widget in pixels. **i Note** This function is called only with Web Components that implement a custom widget proper (with a value of `kind="main"` in the custom widget JSON) but not with Styling Panels or Builder Panels (with values of `kind="styling"` or `kind="builder"`). **→ Tip** If your custom widget doesn't depend on resizing, don't implement this function. This frees the framework from calling this (potentially empty) function repeatedly during a resize operation. **i Note** To improve the performance of a resize operation, this function is called in certain time intervals. The length of those time intervals may be subject to change.

Property Setter Functions

You can implement property setter functions to set the properties of your custom widget.

A property setter function starts with the `set` keyword (object accessor notation) followed by the name of the property. It takes the new property value as an argument. The following example shows a setter function for a property `color`:

Sample Code

```
set color(newColor) {
    this.style["background-color"] = newColor;
}
```

Property setters are optional. The property setters are called by the Custom Widget framework when implemented and skipped when they are not.

Note

A getter function, implemented in a similar way using the `get` keyword, is ignored by the Custom Widget framework.

Tip

Depending on the functional design of your custom widget, a common programming pattern is to omit the implementation of specific property setter functions and instead implement the update of the associated properties in the `onCustomWidgetAfterUpdate` function.

6.2.1 Using Script API Data Types in JavaScript Functions

You can use certain script API data types, for example, `Button`, in native JavaScript functions in a Web Component implementation of your custom widget. For example, you can use them as function arguments and return values. You can also call functions provided by those script API data types.

A native JavaScript function is a JavaScript function in a Web Component of your custom widget as opposed to a script method that your custom widget exposes in the analytics designer script editor.

The following rules apply when using script API data types in native JavaScript functions:

- You must call only documented functions provided by a script API data type.
- You must prepare your code to correctly accept the returned value of a function call on a script API data type, as it is returned asynchronously. To retrieve such a function result correctly, apply one of the following programming patterns in your native JavaScript function:
  1. Use the `await` and `async` keywords. The following example shows how to call the function `getText` of a `Button` that has been passed as a function argument to function `myFunction`:

     Example

     Sample Code

     ```
     async function myFunction(Button button) {
       var text = await button.getText();
       // do something with the passed text
     }
     ```

  2. Use the `then` function. The following example shows how to call the function `getText` of a `Button` that has been passed as a function argument to function `myFunction`:

Example

Sample Code

```
function myFunction(Button button) {
  button.getText().then(myOtherFunction(text) {
    // do something with the passed text
  });
}
```

3. If your browser doesn't support the `await` and `async` keywords or the `then` function, use a transpiler that provides the functionality of one of the above patterns.

6.2.2 Importing Script API Data Types with Type Libraries

Your custom widget can use script API data types in its implementation, for example, with method arguments, properties, and return values.

The script API types are grouped into type libraries.

> **Note**
>
> Currently, type libraries are optional in the Custom Widget framework.

To indicate that your Web Component uses a specific script API data type, request the appropriate type library in the Custom Widget JSON. For example, if your Web component uses a `Button` and a `Table` script API data type, add this line to the Custom Widget JSON:

Sample Code

```
"imports": ["table", "input-controls"]
```

> **Note**
>
> The type library *standard* is loaded automatically.

The list of type libraries is available in the Analytics Designer API Reference on SAP Help Portal at http://help.sap.com.

6.3 Styling Panel

Implementing a Styling Panel is optional.

The Styling Panel of a custom widget is an area in analytics designer where you can set property values of the custom widget at design time. This area is located in the *Styling* tab of the *Designer* panel of analytics designer and has the title *Custom Widget Additional Properties*.

The Styling Panel shares the screen space of the *Styling* tab with other UI elements.

The Styling Panel is also implemented as a Web Component. Its implementation is contained in a separate Web Component JavaScript file.

> **Note**
>
> At design time, the analytics designer creates for each custom widget a separate Styling Panel next to the canvas. As the Styling Panel is implemented as a Web Component, the same lifecycle functions and callbacks are called like those of the Web Component proper (with a value of `kind="main"` in the custom widget JSON) – except for `onCustomWidgetResize`, which is never called.

> **Note**
>
> At design time, when you delete a custom widget from the canvas, then the Styling Panel is destroyed and its lifecycle functions and callbacks `onCustomWidgetDestroy` and `disconnectedCallback` are called. The exact sequence is not defined. If you need to clean up the state of your Styling Panel, implement this in `onCustomWidgetDestroy`.

6.4 Builder Panel

Implementing a Builder Panel is optional.

The Builder Panel of a custom widget is an area in analytics designer where you can set property values of the custom widget at design time. This area is located in the *Builder* tab of the *Designer* panel of analytics designer and has the title *Builder*.

The Builder Panel fills the entire screen space of the *Builder* tab.

The Builder Panel is also implemented as a Web Component. Its implementation is contained in a separate Web Component JavaScript file.

> **Note**
>
> At design time, the analytics designer creates for each custom widget a separate Builder Panel next to the canvas. As the Builder Panel is implemented as a Web Component, the same lifecycle functions and callbacks are called like those of the Web Component proper (with a value of `kind="main"` in the custom widget JSON) – except for `onCustomWidgetResize`, which is never called.

> **Note**
>
> At design time, when you delete a custom widget from the canvas, then the Builder Panel is destroyed and its lifecycle functions and callbacks `onCustomWidgetDestroy` and `disconnectedCallback` are called. The exact sequence is not defined. If you need to clean up the state of your Builder Panel, implement this in `onCustomWidgetDestroy`.

7 A Sample Custom Widget: Colored Box

The following sections lead you through the code of a very simple custom widget, the Colored Box.

Basically, it is nothing more than the name implies: a red box with a black border. Yet, it touches all relevant capabilities of custom widgets: It can be inserted on the canvas, it can be moved and resized, it adds script methods to the script editor, and it adds a Styling Panel and a Builder Panel with which you can modify properties of the Colored Box in the analytics designer at design time.

The Colored Box custom widget consists of three Web Components: the actual Colored Box, the Styling Panel, and the Builder Panel of the Colored Box.

The sample consists of the following files:

File	Description
`coloredbox.json`	Custom widget JSON of the Colored Box. Find the source code in chapter Custom Widget JSON [page 32].
`coloredbox.js`	Web Component JavaScript file of the Colored Box. Find the source code in chapter Web Component JavaScript [page 36].
`coloredbox_styling.js`	Web Component JavaScript file of the Styling Panel of the Colored Box. Find the source code in chapter Web Component JavaScript of the Styling Panel [page 41].
`coloredbox_builder.js`	Web Component JavaScript file of the Builder Panel of the Colored Box. Find the source code in chapter Web Component JavaScript of the Builder Panel [page 45].
`icon.png`	Icon of the Colored Box. Any 16 x 16 pixel icon will do.

7.1 Custom Widget JSON

The following code shows the custom widget JSON of the Colored Box (file `coloredbox.json`):

Sample Code

```
{
    "id": "com.sap.sample.coloredbox",
    "version": "1.0.0",
    "name": "Colored Box",
    "description": "A colored box",
    "newInstancePrefix": "ColoredBox",
    "icon": "https://www.sample.com/customwidgets/coloredbox/icon.png",
    "vendor": "SAP",
    "eula": "",
    "license": "",
    "webcomponents": [
        {
            "kind": "main",
```

```
            "tag": "com-sap-sample-coloredbox",
            "url": "https://www.sample.com/customwidgets/coloredbox/
coloredbox.js",
            "integrity": "",
            "ignoreIntegrity": true
        },
        {
            "kind": "styling",
            "tag": "com-sap-sample-coloredbox-styling",
            "url": "https://www.sample.com/customwidgets/coloredbox/
coloredbox_styling.js",
            "integrity": "",
            "ignoreIntegrity": true
        },
        {
            "kind": "builder",
            "tag": "com-sap-sample-coloredbox-builder",
            "url": "https://www.sample.com/customwidgets/coloredbox/
coloredbox_builder.js",
            "integrity": "",
            "ignoreIntegrity": true
        }
    ],
    "properties": {
        "color": {
            "type": "string",
            "description": "Background color",
            "default": "red"
        },
        "opacity": {
            "type": "number",
            "description": "Opacity",
            "default": 1
        },
        "width": {
            "type": "integer",
            "default": 100
        },
        "height": {
            "type": "integer",
            "default": 100
        }
    },
    "methods": {
        "setColor": {
            "description": "Sets the background color.",
            "parameters": [
                {
                    "name": "newColor",
                    "type": "string",
                    "description": "The new background color"
                }
            ],
            "body": "this.color = newColor;"
        },
        "getColor": {
            "returnType": "string",
            "description": "Returns the background color.",
            "body": "return this.color;"
        }
    },
    "events": {
        "onClick": {
            "description": "Called when the user clicks the Colored Box."
        }
    }
}
```

The following text explains each section of the custom widget JSON. Let's start with the top part of the custom widget JSON:

Sample Code

```
"id": "com.sap.sample.coloredbox",
"version": "1.0.0",
"name": "Colored Box",
"description": "A colored box",
"newInstancePrefix": "ColoredBox",
"icon": "https://www.sample.com/customwidgets/coloredbox/icon.png",
"vendor": "SAP",
"eula": "",
"license": "",
```

The Colored Box custom widget has the unique id `"com.sap.sample.coloredbox"`, version `"1.0.0."`, and the name `"Colored Box"`, which is displayed in the analytics designer. It has the description `"A colored box"`, which is used, for example, in tooltips. The prefix for new instances of Colored Box custom widgets in the analytics designer is `"ColoredBox"`. The icon for the Colored Box is provided by a URL. The vendor is `"SAP"`. The end-user license agreement and the license string are empty.

The Colored Box custom widget is composed of the following three Web Components:

Sample Code

```
  "webcomponents": [
        {
            "kind": "main",
            "tag": "com-sap-sample-coloredbox",
            "url": "https://www.sample.com/customwidgets/coloredbox/
coloredbox.js",
            "integrity": "",
            "ignoreIntegrity": true
        },
        {
            "kind": "styling",
            "tag": "com-sap-sample-coloredbox-styling",
            "url": "https://www.sample.com/customwidgets/coloredbox/
coloredbox_styling.js",
            "integrity": "",
            "ignoreIntegrity": true
        },
        {
            "kind": "builder",
            "tag": "com-sap-sample-coloredbox-builder",
            "url": "https://www.sample.com/customwidgets/coloredbox/
coloredbox_builder.js",
            "integrity": "",
            "ignoreIntegrity": true
        }
    ],
```

The first Web Component is the actual Colored Box as indicated by the kind of `"main"`. The name of its custom element is `"com-sap-sample-coloredbox"`. A URL references this Web Component's JavaScript file. A SHA256-hash can be added to check the integrity of the referenced file. However, for development purposes, this check is disabled by setting the `ignoreIntegrity` property to `true`.

The second Web Component is the Styling Panel of the Colored Box as indicated by the kind of `"styling"`. The name of its custom element is `"com-sap-sample-coloredbox-styling"`. A URL references this Web Component's JavaScript file. A SHA256-hash can be added to check the integrity of the referenced file. However, for development purposes, this check is also disabled by setting the `ignoreIntegrity` property to `true`.

The third Web Component is the Builder Panel of the Colored Box as indicated by the kind of `"builder"`. The name of its custom element is `"com-sap-sample-coloredbox-builder"`. A URL references this Web Component's JavaScript file. A SHA256-hash can be added to check the integrity of the referenced file. However, for development purposes, this check is also disabled by setting the `ignoreIntegrity` property to `true`.

Next are the properties of the Colored Box custom widget: `color`, `opacity`, `width`, and `height`.

Sample Code

```
  "properties": {
        "color": {
            "type": "string",
            "description": "Background color",
            "default": "red"
        },
        "opacity": {
            "type": "number",
            "description": "Opacity",
            "default": 1
        },
        "width": {
            "type": "integer",
            "default": 100
        },
        "height": {
            "type": "integer",
            "default": 100
        }
    },
```

The property `color` represents the background color of the Colored Box. Its type is a string with a default value of `"red"`.

The property `opacity` represents how much the area behind the Colored Box is obscured by the Colored Box. Its type is a number with a default value of `1` (completely block the area behind the Colored Box).

The properties `width` and `height` represent the initial width and height of the custom widget. They are both of type `integer` with a default value of 100 pixels. The properties `width` and `height` are special properties to the Custom Widget framework: Implemented setter functions for them will not be called.

Then, the script methods of the Colored Box are defined:

Sample Code

```
  "methods": {
    "setColor": {
      "description": "Sets the background color.",
      "parameters": [
        {
          "name": "newColor",
          "type": "string",
          "description": "The new background color"
        }
```

```
        ],
        "body": "this.color = newColor;"
      },
      "getColor": {
        "returnType": "string",
        "description": "Returns the background color.",
        "body": "return this.color;"
      }
    },
```

Function `setColor` takes one parameter, the new color. The `body` property contains the script code, which sets the passed parameter `newColor` , a `string`, to the Colored Box's `color` property.

This function definition lets you write script code in the analytics designer script editor like, for example, `ColoredBox_1.setColor("red");`.

Function `getColor` takes no parameters and returns the color. The `body` property contains the script code, which returns the value of the Colored Box's `color` property as a value of type `string`.

This function definition lets you write script code in the analytics designer script editor like, for example, `var theColor = ColoredBox_1.getColor();`.

To keep the sample short, similar script methods for the `opacity` property are not implemented.

Finally, an `onClick` event is defined:

Sample Code

```
  "events": {
    "onClick": {
      "description": "Called when the user clicks the Colored Box."
    }
  }
```

This event has no parameters, as events can't have parameters yet.

7.2 Web Component JavaScript

The following listing shows the Web Component JavaScript of the Colored Box (file `coloredbox.js`).

It is explained step by step in the following sections.

Sample Code

```
(function() {
    let template = document.createElement("template");
    template.innerHTML = `
        <style>
        :host {
            border-radius: 25px;
            border-width: 4px;
            border-color: black;
            border-style: solid;
            display: block;
        }
```

```
        </style>
    `;
    class ColoredBox extends HTMLElement {
        constructor() {
            super();
            let shadowRoot = this.attachShadow({mode: "open"});
            shadowRoot.appendChild(template.content.cloneNode(true));
            this.addEventListener("click", event => {
                var event = new Event("onClick");
                this.dispatchEvent(event);
            });
            this._props = {};
        }
        onCustomWidgetBeforeUpdate(changedProperties) {
            this._props = { ...this._props, ...changedProperties };
        }
        onCustomWidgetAfterUpdate(changedProperties) {
            if ("color" in changedProperties) {
                this.style["background-color"] = changedProperties["color"];
            }
            if ("opacity" in changedProperties) {
                this.style["opacity"] = changedProperties["opacity"];
            }
        }
    }
    customElements.define("com-sap-sample-coloredbox", ColoredBox);
})();
```

Defining the Custom Element and Its JavaScript API

The Web Component JavaScript defines a new custom element `com-sap-sample-coloredbox` and associates it with a JavaScript API represented by the `ColoredBox` class in the following code:

Sample Code

```
(function() {
  ...
  class ColoredBox extends HTMLElement {
    ...
  }
  customElements.define("com-sap-sample-coloredbox", ColoredBox);
})();
```

The entire JavaScript code of the file is nested into a self-executing JavaScript function. This isolates the scope of its variables from the global scope. The JavaScript API of the new custom element is implemented in the `ColoredBox` class, which extends the JavaScript API of the `HTMLElement` class. Calling `customElements.define()` defines a new custom element named `com-sap-sample-coloredbox` and associates it with the JavaScript API represented by the `ColoredBox` class. Note that `com-sap-sample-coloredbox` is the same name as the property value of property name `tag` of the first item in the `webcomponents` array inside the custom widget JSON.

Creating a Template Object

The following code creates a template HTML element:

Sample Code

```
let template = document.createElement("template");
template.innerHTML = `
  <style>
  :host {
    border-radius: 25px;
    border-width: 4px;
    border-color: black;
    border-style: solid;
    display: block;
  }
  </style>
`;
```

Template elements are an important part of Web Components. This template HTML element is a template for the “shadow DOM” HTML element that represents the HTML DOM of the Colored Box, which exists separately from the HTML DOM of the web page. It is inserted only during rendering into the HTML DOM of the web page. Thus, you can, for example, apply styles to a shadow DOM element without interfering with the HTML DOM of the web page.

This template element is very simple. It declares a CSS style of several CSS properties together with the CSS selector `:host`. This CSS style is assigned to the template element. The selector `:host`, part of the Web Components standard, selects the root of the shadow DOM element, that is, the root of our template element. The CSS property `display` controls the visualization of the custom HTML element, possible values are, for example, `block`, `flex`, or `inline-block`.

Defining the JavaScript API of the Custom Element

The JavaScript API of the custom element is defined by the `ColoredBox` class, which extends the `HTMLElement` class:

Sample Code

```
class ColoredBox extends HTMLElement {
  ...
}
```

Implementing the Constructor

The first function in the JavaScript API is the constructor:

Sample Code

```
    constructor() {
            super();
            let shadowRoot = this.attachShadow({mode: "open"});
            shadowRoot.appendChild(template.content.cloneNode(true));
            this.addEventListener("click", event => {
                var event = new Event("onClick");
                this.dispatchEvent(event);
            });
            this._props = {};
        }
```

First, the `super()` function is called. Then, the shadow DOM root element is created. Next, a copy of the template element is added as a child element to the shadow DOM root element. Finally, an event listener is attached to the custom element, listening for `click` events. If one such event occurs, its handler function creates a new `onClick` event and dispatches ("fires") it. Note that the name of the event, `onClick`, must be the same name as it is defined in the `events` section of the custom widget JSON. Note also, that `this` in the code refers to the custom element.

Lastly, to make managing the properties of the Web Component easier, an empty `_props` object is initialized.

Handling Custom Widget Updates

The Colored Box handles updates of its two properties `color` and `opacity`, defined in the `properties` array of the custom widget JSON, by implementing the `onCustomWidgetBeforeUpdate` and `onCustomWidgetAfterUpdate` functions.

In the `onCustomWidgetBeforeUpdate` function, the properties in the passed `changedProperties` object are merged with the properties of the `_props` object. Thus, `_props` contains the state of all Colored Box properties before the Colored Box is updated.

Sample Code

```
onCustomWidgetBeforeUpdate(changedProperties) {
   this._props = { ...this._props, ...changedProperties };
}
```

In the `onCustomWidgetAfterUpdate` function, the properties in the passed `changedProperties` object is used to directly set the color and opacity CSS styles of the Colored Box.

Sample Code

```
onCustomWidgetAfterUpdate(changedProperties) {
   if ("color" in changedProperties) {
      this.style["background-color"] = changedProperties["color"];
   }
   if ("opacity" in changedProperties) {
      this.style["opacity"] = changedProperties["opacity"];
   }
}
```

→ Tip

To print a property value to the browser console, for example, during debugging, use code in the following form:

‘≒› Sample Code

```
console.log(`${this._props[<propertyName>]}`);
```

The following example prints the widget's name to the browser console during `onCustomWidgetBeforeUpdate`:

‘≒› Sample Code

```
onCustomWidgetBeforeUpdate(changedProperties) {
        // ...
        console.log(`${this._props["widgetName"]}`);
    }
```

Implementing Property Setters

Another way to implement the update of the Colored Box properties `color` and `opacity` is to implement property setter functions.

i Note

This approach is useful only for simple custom widgets with very few properties that are independent from each other. The more properties are added to the custom widget the more likely it is that (potentially complex) dependencies between properties arise. Managing these dependencies by implementing a `onCustomWidgetAfterUpdate` function is the preferred alternative.

The following code shows the implementation of the property setter functions:

‘≒› Sample Code

```
set color(newColor) {
   this.style["background-color"] = newColor;
}
set opacity(newColor) {
   this.style["opacity"] = newColor;
}
```

Property setter functions have the name of the property as it is defined in the custom widget JSON. A setter function is prefixed with the `set` keyword (object accessor notation).

The actual implementation is simple: For example, in the setter function of the `color` property, the passed value of the new color is assigned to the CSS style property `background-color` of the custom element.

Similar rules apply to the `opacity` property.

7.3 Web Component JavaScript of the Styling Panel

The following listing shows the JavaScript of the Styling Panel of the Colored Box (file `coloredbox_styling.js`).

This Styling Panel lets you change the color of the Colored Box in analytics designer.

It is explained step by step in the following sections. This JavaScript code is very similar to the Web Component JavaScript of the Colored Box.

Sample Code

```
(function()  {
    let template = document.createElement("template");
    template.innerHTML = `
        <form id="form">
            <fieldset>
                <legend>Colored Box Properties</legend>
                <table>
                    <tr>
                        <td>Color</td>
                        <td><input id="styling_color" type="text" size="40"
maxlength="40"></td>
                    </tr>
                </table>
                <input type="submit" style="display:none;">
            </fieldset>
        </form>
    `;
    class ColoredBoxStylingPanel extends HTMLElement {
        constructor() {
            super();
            this._shadowRoot = this.attachShadow({mode: "open"});
            this._shadowRoot.appendChild(template.content.cloneNode(true));

this._shadowRoot.getElementById("form").addEventListener("submit",
this._submit.bind(this));
        }
        _submit(e) {
            e.preventDefault();
            this.dispatchEvent(new CustomEvent("propertiesChanged", {
                    detail: {
                        properties: {
                            color: this.color
                        }
                    }
            }));
        }
        set color(newColor) {
            this._shadowRoot.getElementById("styling_color").value = newColor;
        }
        get color() {
            return this._shadowRoot.getElementById("styling_color").value;
        }
    }
customElements.define("com-sap-sample-coloredbox-styling",
ColoredBoxStylingPanel);
```

Defining the Custom Element and Its JavaScript API

The Web Component JavaScript defines a new custom element `com-sap-sample-coloredbox-styling` and associates it with a JavaScript API represented by the `ColoredBoxStylingPanel` class in the following code:

Sample Code

```
(function() {
  ...
  class ColoredBoxStylingPanel extends HTMLElement {
    ...
  }
  customElements.define("com-sap-sample-coloredbox-styling",
ColoredBoxStylingPanel);
})();
```

The entire JavaScript code of the file is nested into a self-executing JavaScript function. This isolates the scope of its variables from the global scope. The JavaScript API of the new custom element is implemented in the `ColoredBoxStylingPanel` class, which extends the JavaScript API of the `HTMLElement` class. Calling `customElements.define()` defines a new custom element named `com-sap-sample-coloredbox-styling` and associates it with the JavaScript API represented by the `ColoredBoxStylingPanel` class.

Note

Note that `com-sap-sample-coloredbox-styling` is the same name as the property value of property name tag of the second item in the `webcomponents` array inside the custom widget JSON.

Creating a Template Object

The following code creates a template HTML element:

Sample Code

```
  let template = document.createElement("template");
  template.innerHTML = `
    <form id="form">
      <fieldset>
        <legend>Colored Box Properties</legend>
        <table>
          <tr>
            <td>Color</td>
            <td><input id="styling_color" type="text" size="40"
maxlength="40"></td>
          </tr>
        </table>
        <input type="submit" style="display:none;">
      </fieldset>
    </form>
  `;
```

This template HTML element is a template for the shadow DOM HTML element that represents the HTML DOM of the Styling Panel of the Colored Box.

This template element defines a simple form with id `"form"`. It contains a table with an input field with id `"styling_color"` and a length and capacity of 40 characters.

> **→ Tip**
>
> Defining a hidden input field of type `submit` is a trick to make to form work when the form contains more than one input element. Strictly speaking, this it is not necessary yet, but as soon as you add more input elements.

Defining the JavaScript API of the Custom Element

The JavaScript API of the custom element is defined by the `ColoredBoxStylingPanel` class, which extends the `HTMLElement` class:

Sample Code

```
class ColoredBoxStylingPanel extends HTMLElement {
  ...
}
```

Implementing the Constructor

The first function in the JavaScript API is the constructor:

Sample Code

```
    constructor() {
      super();
      this._shadowRoot = this.attachShadow({mode: "open"});
      this._shadowRoot.appendChild(template.content.cloneNode(true));
      this._shadowRoot.getElementById("form").addEventListener("submit",
this._submit.bind(this));
    }
```

First, the `super()` function is called. Then, the shadow DOM root element is created. A reference to it is stored in the local variable `_shadowRoot`, because the shadow DOM root element is referenced later in the `color` setter and getter functions of this class. Next, a copy of the template element is added as a child element to the shadow DOM root element. Finally, an event listener is attached to form, listening for `submit` events. If one such event occurs, the event handler function `_submit()` is called. Calling `bind()` and passing `this` to `_submit()` ensures that in `_submit()` the keyword `this` references the custom element.

The `_submit()` function is implemented as follows:

Sample Code

```
      _submit(e) {
        e.preventDefault();
        this.dispatchEvent(new CustomEvent("propertiesChanged", {
```

```
            detail: {
              properties: {
                color: this.color
              }
            }
          }));
        }
```

The `_submit()` function calls function `preventDefault()` on the passed event object, which prevents submitting the form to the server.

Then, a custom event is created. Its name `"propertiesChanged"` indicates a change of properties to the Custom Widget framework (incidentally, you can also create and dispatch such events in the Web Component JavaScript of your main Web Component). This custom event contains a JSON payload, which defines which properties actually have changed and to what values. In this case it is the `color` property of the custom widget. Its new value is retrieved by executing `this.color`, which implicitly calls the `color` getter function of class `ColoredBoxStylingPanel`.

Implementing Property Getters and Setters

The following code shows the implementation of `color` setter and getter functions.

> **i** Note
>
> Do not confuse them with the `color` setter function of the `ColoredBox` class.

> **‹≡›** Sample Code

```
    set color(newColor) {
      this._shadowRoot.getElementById("styling_color").value = newColor;
    }
    get color() {
      return this._shadowRoot.getElementById("styling_color").value;
    }
```

The `color` setter function places a text representation of the new color into the input field of the Colored Box's Styling Panel. Incidentally, in this JavaScript code this function is not called directly, but by the Custom Widget framework when the Styling Panel Sheet of the Colored Box is created, assigning an initial value to the input field.

The `color` getter function returns the text of the input field of the Colored Box's Styling Panel. This function is called indirectly in the `_submit()` event handler function during the construction of the JSON event payload when the current color is retrieved by `this.color`.

7.4 Web Component JavaScript of the Builder Panel

The following listing shows the JavaScript of the Builder Panel of the Colored Box (file `coloredbox_builder.js`).

This Builder Panel lets you change the opacity of the Colored Box in analytics designer.

The JavaScript code is very similar to the Web Component JavaScript of the Styling Panel of the Colored Box.

Sample Code

```
(function()  {
    let template = document.createElement("template");
    template.innerHTML = `
        <form id="form">
            <fieldset>
                <legend>Colored Box Properties</legend>
                <table>
                    <tr>
                        <td>Opacity</td>
                        <td><input id="builder_opacity" type="text" size="5"
maxlength="5"></td>
                    </tr>
                </table>
                <input type="submit" style="display:none;">
            </fieldset>
        </form>
        <style>
        :host {
            display: block;
            padding: 1em 1em 1em 1em;
        }
        </style>
    `;
    class ColoredBoxBuilderPanel extends HTMLElement {
        constructor() {
            super();
            this._shadowRoot = this.attachShadow({mode: "open"});
            this._shadowRoot.appendChild(template.content.cloneNode(true));

this._shadowRoot.getElementById("form").addEventListener("submit",
this._submit.bind(this));
        }
        _submit(e) {
            e.preventDefault();
            this.dispatchEvent(new CustomEvent("propertiesChanged", {
                    detail: {
                        properties: {
                            opacity: this.opacity
                        }
                    }
            }));
        }
        set opacity(newOpacity) {
            this._shadowRoot.getElementById("builder_opacity").value =
newOpacity;
        }
        get opacity() {
            return this._shadowRoot.getElementById("builder_opacity").value;
        }
    }
    customElements.define("com-sap-sample-coloredbox-builder",
ColoredBoxBuilderPanel);
})();
```

Defining the Custom Element and Its JavaScript API

The Web Component JavaScript defines a new custom element `com-sap-sample-coloredbox-builder` and associates it with a JavaScript API represented by the `ColoredBoxBuilderPanel` class in the following code:

Sample Code

```
(function() {
    ...
    class ColoredBoxBuilderPanel extends HTMLElement {
      ...
    }
    customElements.define("com-sap-sample-coloredbox-builder",
ColoredBoxBuilderPanel);
})();
```

The entire JavaScript code of the file is nested into a self-executing JavaScript function. This isolates the scope of its variables from the global scope. The JavaScript API of the new custom element is implemented in the `ColoredBoxBuilderPanel` class, which extends the JavaScript API of the `HTMLElement` class. Calling `customElements.define()` defines a new custom element named `com-sap-sample-coloredbox-builder` and associates it with the JavaScript API represented by the `ColoredBoxBuilderPanel` class.

Note

Note that `com-sap-sample-coloredbox-builder` is the same name as the property value of property name `tag` of the third item in the `webcomponents` array inside the custom widget JSON.

Creating a Template Object

The following code creates a template HTML element:

Sample Code

```
let template = document.createElement("template");
    template.innerHTML = `
        <form id="form">
            <fieldset>
                <legend>Colored Box Properties</legend>
                <table>
                    <tr>
                        <td>Opacity</td>
                        <td><input id="builder_opacity" type="text" size="5"
maxlength="5"></td>
                    </tr>
                </table>
                <input type="submit" style="display:none;">
            </fieldset>
        </form>
```

```
        <style>
        :host {
            display: block;
            padding: 1em 1em 1em 1em;
        }
        </style>
    `;
```

This template HTML element is a template for the shadow DOM HTML element that represents the HTML DOM of the Styling Panel of the Colored Box.

This template element defines a simple form with id `"form"`. It contains a table with an input field with id `"builder_opacity"` and a length and capacity of 5 characters.

> **→ Tip**
>
> Defining the hidden input field of type `submit` is a trick to make to form work when the form contains more than one input element. Strictly speaking, this it is not necessary yet, but as soon as you add more input elements.

The CSS selector `:host` adds CSS styles to the template element. The selector `:host`, part of the Web Components standard, selects the root of the shadow DOM element, that is, the root of our template element. The CSS property `display` controls the visualization of the custom HTML element, in this case as a `block`. The CSS property padding ensures enough `padding` around the form.

Defining the JavaScript API of the Custom Element

The JavaScript API of the custom element is defined by the `ColoredBoxBuilderPanel` class, which extends the `HTMLElement` class:

Sample Code

```
class ColoredBoxBuilderPanel extends HTMLElement {
   ...
}
```

Implementing the Constructor

The first function in the JavaScript API is the constructor:

Sample Code

```
constructor() {
            super();
            this._shadowRoot = this.attachShadow({mode: "open"});
            this._shadowRoot.appendChild(template.content.cloneNode(true));

this._shadowRoot.getElementById("form").addEventListener("submit",
this._submit.bind(this));
```

First, the `super()` function is called. Then, the shadow DOM root element is created. A reference to it is stored in the local variable `_shadowRoot`, because the shadow DOM root element is referenced later in the `color` setter and getter functions of this class. Next, a copy of the template element is added as a child element to the shadow DOM root element. Finally, an event listener is attached to form, listening for `submit` events. If one such event occurs, the event handler function `_submit()` is called. Calling `bind()` and passing `this` to `_submit()` ensures that in `_submit()` the keyword `this` references the custom element.

The `_submit()` function is implemented as follows:

Sample Code

```
_submit(e) {
   e.preventDefault();
   this.dispatchEvent(new CustomEvent("propertiesChanged", {
      detail: {
         properties: {
            opacity: this.opacity
         }
      }
   }));
}
```

The `_submit()` function calls function `preventDefault()` on the passed event object, which prevents submitting the form to the server.

Then, a custom event is created. Its name `"propertiesChanged"` indicates a change of properties to the Custom Widget framework. This custom event contains a JSON payload, which defines which properties actually have changed and to what values. In this case it is the `opacity` property of the custom widget. Its new value is retrieved by executing `this.opacity`, which implicitly calls the `opacity` getter function of class `ColoredBoxBuilderPanel`.

Implementing Property Getters and Setters

The following code shows the implementation of the `opacity` setter and getter functions.

Sample Code

```
set opacity(newOpacity) {
   this._shadowRoot.getElementById("builder_opacity").value = newOpacity;
}
get opacity() {
   return this._shadowRoot.getElementById("builder_opacity").value;
}
```

The `opacity` setter function places a text representation of the new opacity into the input field of the Colored Box's Builder Panel. Incidentally, in this JavaScript code this function is not called directly, but by the Custom Widget framework when the Builder Panel of the Colored Box is created, assigning an initial value to the input field.

The `opacity` getter function returns the text of the input field of the Colored Box's Builder Panel. This function is called indirectly in the `_submit()` event handler function during the construction of the JSON event payload when the current opacity is retrieved by `this.opacity`.

8 Custom Widgets on Mobile Devices

You can enable and run a custom widget on a mobile device.

Enabling Your Custom Widget for Mobile Devices

To enable a custom widget to run on a mobile device, set its `supportsMobile` property to `true` (see section *Root Object* in chapter Custom Widget JSON Reference [page 11] for more information).

> ⚠ Caution
>
> Before enabling this flag, please make sure that your custom widget's functionality is working properly on mobile devices.

Detecting if Your Custom Widget is Running on a Mobile Device

If the `supportsMobile` property is set to true, then the property `mobileMode` is passed in the `otherProperties` array to functions `onCustomWidgetBeforeUpdate` and `onCustomWidgetAfterUpdate`. This lets you find out at runtime if the custom widget is currently running on a mobile device. See also section *Provide alternatives to user interactions that use modifier keys* below for a code example.

Tips for Developing Custom Widgets for Mobile Devices

When you develop a custom widget for a mobile device instead of a non-mobile device (for example, a desktop or laptop computer), consider the following suggestions:

- Make areas larger that provide user interaction.
  Ensure that areas of your custom widget that provide user interaction are sufficiently large. For example, you may want to enlarge clickable areas in your custom widget when it is running on a mobile device, so that it can be comfortably tapped with a finger.
- Provide alternatives to user interactions that use modifier keys.
  User interactions on a desktop computer often involve using the modifier keys of a keyboard, such as, for example, the `Shift` or `Control` key. Such modifier keys are not present on a mobile device. If your custom widget supports using such modifier keys when running on a desktop computer, think about providing an alternative when your custom widget is running on a mobile device. Such an alternative should comply with the user interaction guidelines and standards of your mobile device.
  The following example shows custom widget code that performs the operation `doAction` after a user interaction. The specific user interaction depends on whether the custom widget is running on a mobile or

on a non-mobile device. They are implemented as follows: With a custom widget running on a non-mobile device, you need to click the custom widget and press the `Shift` key at the same time to trigger the operation. With a custom widget running on a mobile device, you need to tap the custom widget with two fingers to trigger the operation.
The example code below registers one of two event listeners when the custom widget is in view mode. When the custom widget is running on a non-mobile device, then an event listener is registered that performs operation `doAction` when the custom widget is clicked and at the same time the `Shift` key is pressed. When the custom widget is running on a mobile device, then an event listener is registered that performs operation `doAction` when the custom widget is tapped with two fingers.
File: coloredbox_main.js

Sample Code

```
class ColoredBox extends HTMLElement {
    constructor() {
        ...
        this._myEventListenersRegistered = false;
    }
    onCustomWidgetBeforeUpdate(changedProps) {
        this._props = { ...this._props, ...changedProps };
        // registers the event listeners when not in design mode,
        // when the event listeners haven't been registered yet,
        // and when in mobile mode
        if (this._props["designMode"] === false && !
this._myEventListenersRegistered) {
            this._props["mobileMode"] && this._props["mobileMode"] ===
true ?
            this.registerTwoFingerTapListener() :
            this.registerShiftClickListener();
            this._myEventListenersRegistered = true;
        }
    }
    registerTwoFingerTapListener() {
        this.addEventListener("touchstart", event => {
            event.preventDefault();
            if (event.targetTouches.length === 2) {
                // at view time, touching with two fingers performs the
action below
                this.doAction();
            }
        });
    }
    registerShiftClickListener() {
        this.addEventListener("click", event => {
            const shiftKey = event.shiftKey;
            if (shiftKey) {
                // at view time, shift-clicking performs the action below
                this.doAction();
            } else {
                this.dispatchEvent(new Event("onClick"));
            }
        });
    }
    doAction() {
        // do something
    }
```

9 Best Practices

9.1 Dispatching a Property Change

When one of your properties has changed, store the changed property value, for example, in a variable of your Web Component JavaScript, then notify the framework of this change by dispatching a `propertiesChanged` custom event.

The following example shows dispatching a property change of property `color`:

Sample Code

```
this.dispatchEvent(new CustomEvent("propertiesChanged", {
   detail: {
      properties: {
        color: this.color
      }
   }
}));
```

Do dispatch a `propertiesChanged` custom event in the following situations:

- as a result of a user interaction
- in native JavaScript functions

Do not dispatch a `propertiesChanged` custom event in the following situations and functions, as this may cause, for example, an endless loop in JavaScript execution:

- constructor
- `onCustomWidgetBeforeUpdate`
- `onCustomWidgetAfterUpdate`
- `onCustomWidgetDestroy`
- `connectedCallback`
- `disconnectedCallback`

Note that after dispatching the `propertiesChanged` custom event, the changed property value is not part of the properties that are passed later to the `onCustomWidgetBeforeUpdate(oChangedProperties)` and `onCustomWidgetAfterUpdate(oChangedProperties)` functions of this Web Component.

This is simply because, as this Web Component initiated dispatching the property change in the first place, it doesn't need to be notified of it again. However, the changed property is automatically passed to related Web Components, because they need to be notified of the property change. Related Web Components of the custom widget's Web Component proper are the Web Component of the Styling Panel and the Web Component of the Builder Panel. The same rules, with appropriately reversed roles, apply when dispatching property changes in the Web Component of the Styling Panel and the Web Component of the Builder Panel.

9.2 Calling Methods of Passed Objects of Script API Types

Call methods of passed objects of script API data types, for example `Button`, only in script methods.

As you recall, there are two kinds of script methods, depending on where the script method is implemented:

1. The method definition in the custom widget JSON has a `body` property that contains the script method implementation.
2. The method definition in the custom widget JSON does not have a `body` property. The script method implementation is a native JavaScript function in the Web Component JavaScript.

The following example shows parts of a custom widget JSON and the corresponding Web Component JavaScript.

This shows the correct use: The script method `setButtonText1`, with a `body` property, calls the `setText` method of the passed `Button` object (see (1)), as does the script method `setButtonText2`, which has no `body` property but is implemented as a native JavaScript function in the Web Component JavaScript (see (2)).

Don't call methods of passed objects of script API data types outside these two situations, for example:

- in a constructor
- during rendering
- in other functions (see (3) and (4)

Custom widget JSON

Sample Code

```
{
    ...
    "methods": {
        "setButtonText1": {
            "parameters": [
                {"name": "button", "type": "Button"},
                {"name": "text", "type": "string"}
            ],
            ...
            "body": "button.setText(text);"  // (1) OK
        },
        "setButtonText2": {
            "parameters": [
                {"name": "button", "type": "Button"},
                {"name": "text", "type": "string"}
            ],
            ...
        },
        ...
    },
    ...
}
```

Web Component JavaScript

Sample Code

```
class CustomWidget /* extends ...*/ {
    _button = undefined;
    setButtonText2(button, text) {
```

```
        button.setText(text);           // (2) OK
        _button = button;
        setTimeout(() => {
            this._button.setText(text); // (3) WILL NOT WORK!
        });
    }
    handleClick() {
        this._button.setText("Hello!"); // (4) WILL NOT WORK!
    }
}
```

9.3 Implementing Property Setter and Getter Methods Consistently

Shows you how to implement a pair of property setters and getters.

To implement a pair of property setters and getters, for example `setText` and `getText`, which can be used like in the following example:

Sample Code

```
customWidget.setText("Hello");
var text = customWidget.getText();
```

you can choose between two implementation styles:

In the first style, you define the necessary elements (the property and the setter and the getter method) in the custom widget JSON. Both methods are implemented in their `body` properties:

Custom Widget JSON

Sample Code

```
{
    ...
    "properties": {
        "text": {"type": "string"}
    },
    "methods": {
        "setText": {
            "parameters": [{"name": "newText", "type": "string"}],
            "body": "this.text = newText;"
        },
        "getText": {
            "returnType": "string",
            "body": "return this.text;"
        }
    }
}
```

Written this way, the framework keeps track of the state of `text`. When the setter method is called, the framework implicitly triggers a state change.

In the second style, you define the necessary elements (the property and the setter and getter method) in the custom widget JSON and implement them in the Web Component JavaScript. Both methods are defined

without a `body` property in the custom widget JSON. They are implemented as native JavaScript functions in the Web Component JavaScript:

Custom Widget JSON

Sample Code

```
{
    ...
    "properties": {
        "text": {"type": "string"}
    },
    "methods": {
        "setText": {
            "parameters": [{"name": "newText", "type": "string"}],
        },
        "getText": {
            "returnType": "string"
        }
    }
}
```

Web Component JavaScipt

Sample Code

```
class CustomWidget extends /* ... */ {
    var _text;
    setText(newText) {
        this._text = newText;
        // fire "properties changed"
        this.dispatchEvent(new CustomEvent("propertiesChanged", {
            detail: {
                properties: {
                    text: this._text
            }
        }));
    }
    getText() {
        return this._text;
    }
}
```

Written this way, the framework keeps track of the state of `text`. When the setter method is called, the Web Component JavaScript explicitly triggers a state change.

⚠ Caution

Do not mix those implementation styles when implementing property setters and getters!

For example, do not implement the setter of a property in the first style and the getter of the same property in the second style. Strange effects may occur. For example, when running the following sample script with such an implementation

Sample Code

```
customWidget.setText("Hello");
var text = customWidget.getText();
```

it is not guaranteed that at the end of this script `text` has the value `"Hello"`.

> **i Note**
>
> In the Custom Widget framework there is a separation of concerns about custom widgets: The framework handles the property state and the Web Component handles the rendering. For performance reasons, all property changes are collected by the framework first (this includes the property changes during the execution of a script) and only after that, rendering is executed. When mixing the above implementation styles of setter and getter methods, the separation between property state handling and rendering becomes fuzzy. This can lead to unexpected, for example, outdated, property values.

9.4 Returning Arguments by Value and by Reference

Simple types (`string`, `integer`, `number`, `boolean`) and script API data types (for example, `Button`) that are passed to script methods are always returned by value, that is, a copy of the argument is returned.

Arrays and simple typed objects (for example, `object<string>`) that are passed to script methods are returned either by reference or by value. It depends on where the script method's implementation is defined:

- If the script method's implementation is in the `body` property of the custom widget JSON, then arguments of arrays and simple typed objects are returned by reference (the identical argument is returned).
- If the script method's implementation is in a native JavaScript implementation of the Web Component JavaScript, then arguments of arrays and simple typed objects are returned by value (a copy of the argument is returned).

In the following example, an array that is passed to two script methods is returned either by reference (the returned array is the identical array that was passed into the method) or returned by value (the returned array is a copy of the array that was passed into the method):

Custom Widget JSON

> **‹≡› Sample Code**

```
{
    ...
    "methods": {
        "getArrayByRef": {
            "parameters": [{"name": "values", "type": "integer[]"}],
            "returnType": "integer[]",
            "body": "return values;"
        },
        "getArrayByValue": {
            "parameters": [{"name": "values", "type": "integer[]"}],
            "returnType": "integer[]"
        }
}
```

Web Component JavaScript

> **‹≡› Sample Code**

```
class CustomWidget extends /* ... */ {
```

```
    ...
    getArrayByValue(values: integer[]) {
          return values; // "values" is not the original, but already a copy
    }
}
```

Sample Analytics Designer Script

Sample Code

```
var array = [1, 2, 3];
var result1 = customWidget.getArrayByRef(array);
console.log(array === result1); // -> true
var result2 = customWidget.getArrayByValue(array);
console.log(array === result2); // -> false
```

Important Disclaimers and Legal Information

Hyperlinks

Some links are classified by an icon and/or a mouseover text. These links provide additional information.
About the icons:

- Links with the icon : You are entering a Web site that is not hosted by SAP. By using such links, you agree (unless expressly stated otherwise in your agreements with SAP) to this:
  - The content of the linked-to site is not SAP documentation. You may not infer any product claims against SAP based on this information.
  - SAP does not agree or disagree with the content on the linked-to site, nor does SAP warrant the availability and correctness. SAP shall not be liable for any damages caused by the use of such content unless damages have been caused by SAP's gross negligence or willful misconduct.
- Links with the icon : You are leaving the documentation for that particular SAP product or service and are entering a SAP-hosted Web site. By using such links, you agree that (unless expressly stated otherwise in your agreements with SAP) you may not infer any product claims against SAP based on this information.

Videos Hosted on External Platforms

Some videos may point to third-party video hosting platforms. SAP cannot guarantee the future availability of videos stored on these platforms. Furthermore, any advertisements or other content hosted on these platforms (for example, suggested videos or by navigating to other videos hosted on the same site), are not within the control or responsibility of SAP.

Beta and Other Experimental Features

Experimental features are not part of the officially delivered scope that SAP guarantees for future releases. This means that experimental features may be changed by SAP at any time for any reason without notice. Experimental features are not for productive use. You may not demonstrate, test, examine, evaluate or otherwise use the experimental features in a live operating environment or with data that has not been sufficiently backed up.
The purpose of experimental features is to get feedback early on, allowing customers and partners to influence the future product accordingly. By providing your feedback (e.g. in the SAP Community), you accept that intellectual property rights of the contributions or derivative works shall remain the exclusive property of SAP.

Example Code

Any software coding and/or code snippets are examples. They are not for productive use. The example code is only intended to better explain and visualize the syntax and phrasing rules. SAP does not warrant the correctness and completeness of the example code. SAP shall not be liable for errors or damages caused by the use of example code unless damages have been caused by SAP's gross negligence or willful misconduct.

Bias-Free Language

SAP supports a culture of diversity and inclusion. Whenever possible, we use unbiased language in our documentation to refer to people of all cultures, ethnicities, genders, and abilities.

www.sap.com/contactsap

THE BEST RUN SAP