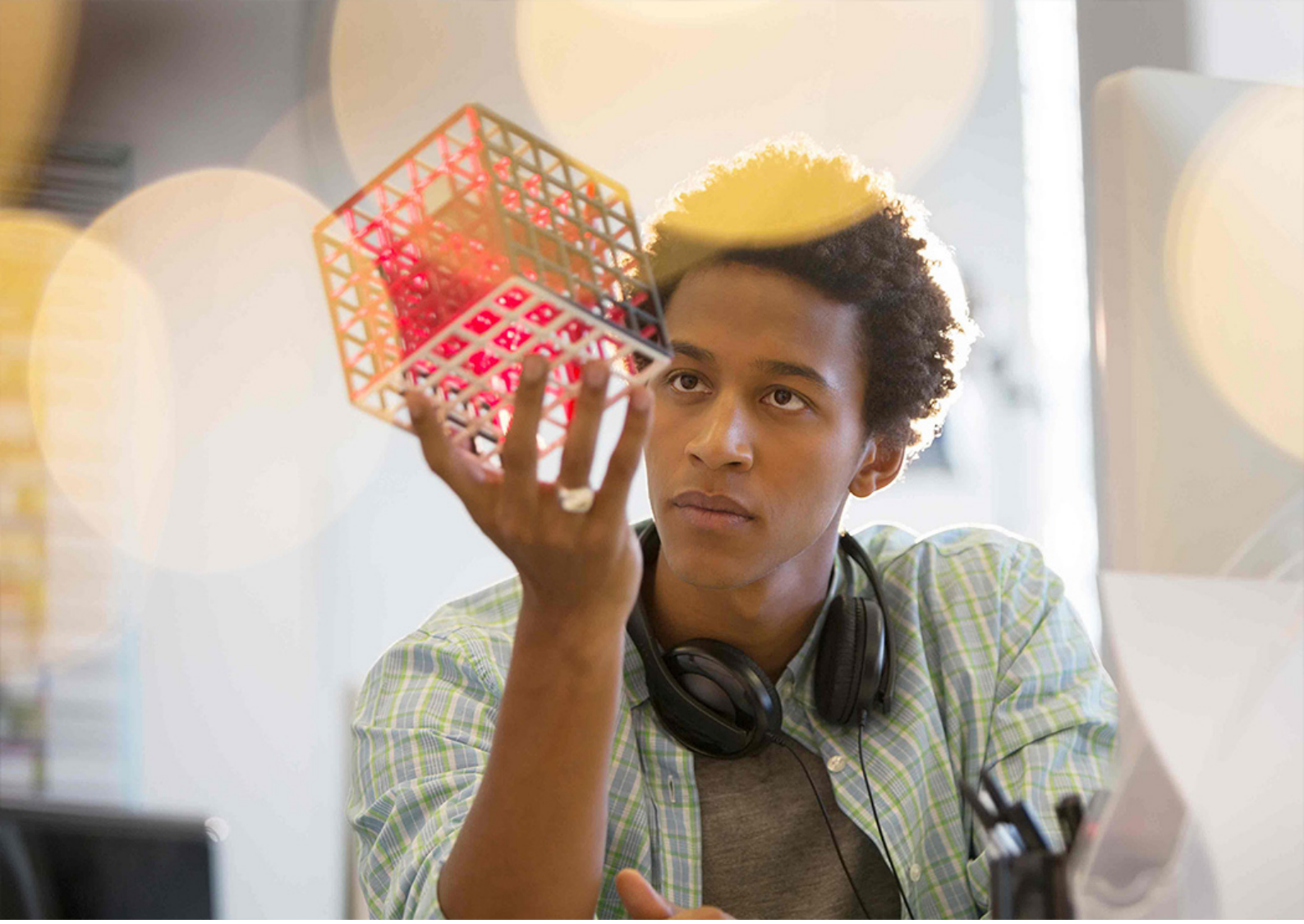

PUBLIC
SAP Lumira
Document Version: 2.4 SP00 – 2020-08-27

Developer Guide: Component SDK

THE BEST RUN SAP

Content

1 About This Guide

1.1 Who Should Read This Guide?

This guide is intended for developers.

1.2 What is the Component SDK?

The Component SDK is a software development kit that allows developers to develop 3rd party components, known as SDK extension components. Application designers can enhance their Lumira documents and analysis applications using these custom components, as well as the standard palette of components in SAP Lumira Designer. You can store and provide access to the Lumira documents and analysis applications, which contain 3rd party components, on the BI platform.

> **i** Note
>
> You can also create new chart types using the Visualization SDK. These chart types are also known as Visualization SDK extensions. Application designers can use them together with the components created with the Component SDK and the standard components of SAP Lumira Designer in their Lumira documents and analysis applications. You can store and provide access to the Lumira documents and analysis applications containing Visualization SDK extensions on the BI platform.
>
> For more information about creating Visualization SDK extensions using the Visualization SDK, see the *Visualization Extension Plugin for SAP Web IDE Guide* on SAP Help Portal.

For more information about deploying extensions to Lumira Designer and BI platform, see "Deploying SDK Extensions" in the *Administrator Guide: SAP Lumira* on SAP Help Portal at https://help.sap.com

2 Introduction to SDK Concepts

2.1 SDK Extensions

Component SDK extensions contain extension components, which are custom components developed by partners and customers.

These extension components integrate seamlessly into Lumira Designer: Like Lumira Designer's standard components, extension components appear in the *Component* view. When placed into an analysis application, these extension components also appear in the editor area and in the *Outline* view. Their properties can be examined and changed in the *Properties* view. If extension components provide properties for which the *Properties* view is not sufficient, they can provide their own *Property* view; the *Additional Properties Sheet*. Extension components can also supply their own Lumira Designer script methods. The visualization of extension components is based on HTML, JavaScript, and CSS – but can be also based on existing SAPUI5 controls to leverage its look-and-feel. Extension components can be data-bound to consume and visualize data from SAP BW and SAP HANA systems. Another flavor of extension components can act like data sources, which produce data for other extension components.

2.2 Client-Server Architecture

Like Lumira Designer standard components, extension components use a client-server architecture. An extension component contains a JavaScript part that runs in the browser (client), which talks to the SDK framework on the back end (server). At the heart of an extension component are its properties, which are stored on the server. The SDK framework provides notification methods to propagate property changes from the server to the extension component on the client, and in the other direction.

2.3 Restrictions

Extension components behave like standard components with the following restrictions:

- They cannot act as container components.
- They cannot use all available property types; they are restricted to a subset of property types.
- They cannot use large result sets.

> **i Note**
>
> The default limit is 10,000 data cells per data-bound property. You can adjust this limit.

- They cannot extend standard components (standard components are technically different from extension components).

3 Creating an SDK Extension

3.1 Getting Started

You can create an SDK extension using any XML and JavaScript editor. However, we recommend Eclipse as an integrated development environment (IDE). This makes SDK extension development much easier. You can create an SDK extension with the Eclipse IDE and test it by launching SAP Lumira Designer from the Eclipse IDE. When launched, SAP Lumira Designer will automatically contain the SDK extension that you have developed.

3.1.1 Prerequisites

- You have installed SAP Lumira Designer (64-bit).
- You have installed the Java Development Kit 7 (or higher) (64-bit). You can download the JDK 7 (64-bit) from the Oracle Website.
- You have basic knowledge of SAP Lumira Designer concepts.
- You have solid knowledge of HTML and JavaScript. Knowledge of CSS and the jQuery JavaScript framework is very helpful.

3.1.2 Extracting the Component SDK Samples and Templates ZIP File

Procedure

1. Download the *Component SDK Templates and Samples* on SAP Help Portal at http://help.sap.com.
2. Extract the downloaded file to a folder, for example `C:\ds_sdk`.

3.1.3 Installing the Eclipse IDE

Procedure

1. Download *Eclipse IDE for Java EE Developers (64 bit)* from download.eclipse.org.

   This edition contains the tools needed to work with the SDK, for example, Plugin Development Tools, XML Editor and JavaScript tools.

   > ⚠ Caution
   >
   > Make sure that you only download this Eclipse version. Other versions, especially 32-bit versions, may not work correctly with the Component SDK.

2. Extract the downloaded file to a folder.
3. Locate and run the file `eclipse.exe`.
4. Close the welcome page.
5. Create a workspace, for example `C:\ds_sdk_workspace`.

   The workspace will contain all your SDK extension projects and the Eclipse IDE settings.

3.1.4 Registering the Component SDK XML Schema Definition

Procedure

1. Choose *Window* > *Preferences*.
2. In the *Preferences* dialog box, choose *XML* > *XML catalog*.
3. Choose *Add...*.
4. In the *Add XML Catalog Element* dialog box, choose *File System...*.
5. Navigate to file `sdk.xsd` in your SDK Templates and Samples folder, for example `C:\ds_sdk\sdk.xsd`.
6. Choose *OK* twice.

3.1.5 Importing a Sample SDK Extension

Procedure

1. Choose *File* > *Import*.
2. In the *Import* dialog box, choose *General* > *Existing Projects into Workspace*.
3. Choose *Next*.
4. Under *Select root directory*, choose *Browse...*.
5. Select the SDK Templates and Samples folder, for example `C:\ds_sdk`.
6. Select sample SDK extension *com.sap.sample.coloredbox*.
7. Select the *Copy projects into Workspace* checkbox.
8. Choose *Finish*.

3.1.6 Setting the Target Platform

Context

The target platform points to your SAP Lumira Designer installation. This enables your Eclipse IDE to access the SDK framework included with SAP Lumira Designer.

The default installation path for SAP Lumira Designer is `C:\Program Files\SAP Lumira\Lumira Designer`.

- If you have installed SAP Lumira Designer in the default folder, follow these steps:
  1. Choose *Window* > *Preferences*.
  2. In the *Preferences* dialog box, choose *Plug-In Development* > *Target Platform*.
  3. Select the checkbox next to the list entry *designstudio*.
  4. Choose *OK*.
  5. Choose *Project* > *Clean*.
  6. Choose *Clean all projects*.
  7. Choose *OK*.
     This removes all error markers.
- If you have not installed SAP Lumira Designer in the default folder, follow these steps:
  1. Choose *Window* > *Preferences*.
  2. In the *Preferences* dialog box, choose *Plug-In Development* > *Target Platform*.
  3. Select the checkbox next to the list entry *designstudio*.
  4. Choose *Edit*.
  5. Choose the *Definition* tab.

6. Choose *Add...*.
7. Choose *Directory* and then choose *Next*.
8. Choose *Browse...* and select the folder of your SAP Lumira Designer installation that contains the file `SapLumiraDesigner.exe`.
9. Choose *OK*.
   A new folder appears in the *Locations* list.
10. Delete the list entry with the red error marker.
11. Save your changes.
12. Choose *Project* > *Clean*.
13. Choose *Clean all projects*.
14. Choose *OK*.
    This removes all error markers.

3.1.7 Testing a Sample SDK Extension

Procedure

1. The first time you test a sample SDK extension in your Eclipse IDE, create a Launch Configuration:
   a. In the Eclipse IDE, choose menu item *Run* > *Run Configurations...*.
   b. Double-click *Eclipse Application* on the left.
   c. In input field *Name*, enter **SDK**.
   d. Click the *Main* tab.
   e. In group *Program to Run*, choose *Run a product* and verify that the text in the adjacent input field reads *com.sap.ip.bi.zen*.
   f. Click the *Arguments* tab.
   g. In input field *VM arguments*, enter:

```
-Xmx1024m
-Xms256m
-XX:PermSize=32m
-XX:MaxPermSize=512m
```

   h. Choose *Apply*, then choose *Close*.
   i. Choose the *Organize Favorites...* menu item in the toolbar in the *Run* popdown (triangle to the right of the green *Play* button).
   j. Choose *Add...*.
   k. Add *SDK*.
   l. Close all dialog boxes with *OK*.
2. Choose the *SDK* menu item in the Eclipse IDE toolbar in the *Run* popdown (triangle to the right of the green *Play* button).

   SAP Lumira Designer starts. The *Components* view contains the extension component *Colored Box*.

> **i Note**
>
> If a message is displayed after you start SAP Lumira Designer informing you that Internet Explorer does not have the required version, add the following registry key to your system:
>
> - Windows (32-bit version):
> `[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer\MAIN \FeatureControl\FEATURE_BROWSER_EMULATION] "javaw.exe"=dword:00002328`
> - Windows (64-bit version):
> `[HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Internet Explorer\MAIN \FeatureControl\FEATURE_BROWSER_EMULATION] "javaw.exe"=dword:00002328`
>
> Your system contains registry key `Security_HKLM_only`. Adding the above registry key forces all `javaw.exe` processes on your system (such as SAP Lumira Designer) to run the Internet Explorer control in the required IE9 mode.

> **i Note**
>
> If you are using Windows 8.x and a 64-bit Eclipse IDE, and the message *no sapjco3 in java.library.path* appears when you launch an application from your Eclipse IDE, follow these steps:
>
> 1. Navigate to the SAP Lumira Designer installation folder, for example `C:\Program Files\SAP Lumira\Lumira Designer`.
> 2. In the `plugins` folder, locate the file `com.sap.conn.jco.win32.x86_64_XXX.jar`.
> 3. Copy this file to a temporary location.
> 4. Change the extension from `.jar` to `.zip`.
> 5. Unzip the file.
> 6. Navigate to the `lib` folder of the unzipped folder.
> 7. Copy the file `sapjco3.dll` to the installation folder of your Eclipse IDE (the folder that contains `eclipse.exe`).

3.2 Creating a New SDK Extension

Context

You can create a new SDK extension by copying the sample SDK extension *Colored Box* and renaming specific parts of it.

Assume your company name is **Sample Company**, your reversed Internet domain name is **com.samplecompany** and your custom component name is **Box**. Perform the operations listed below.

> **i Note**
>
> During these operations, Eclipse may ask you for read/write access to some files. Allow access.

Procedure

1. In the *Package Explorer*, copy project `com.sap.sample.colorbox` to `com.samplecompany.box`.
2. In file `MANIFEST.MF`, replace the following property values:
   - Bundle-Name `Component SDK Extension Sample Colored Box` with `Sample Company Box`
   - Bundle-SymbolicName `com.sap.sample.coloredbox` with `com.samplecompany.box`
   - Bundle-Vendor `SAP` with `Sample Company`
3. In file `.project` in element `<name>`, replace *com.sap.sample.coloredbox* with `com.samplecompany.box`.

   > → Tip
   >
   > To show the `.project` file, open the view menu in *Package Exporer*, choose *Filters...* and deselect *.*resources*.

4. In file `contribution.xml`, replace the following attribute values in element `<sdkExtension>`:
   - attribute `id` `com.sap.sample.coloredbox` with `com.samplecompany.box`
   - attribute `vendor` `SAP` with `Sample Company`
   - attribute `title` `Component SDK Extension Sample Colored Box` with `Sample Company Box`

   > i Note
   >
   > If you choose a value other than `1.0` for element `<sdkExtension>`, attribute `version`, then adjust the first two numbers of the version number (major and minor version number) in the `Bundle-Version` entry in the `MANIFEST.MF` file accordingly. The first two numbers of the `Bundle-Version` and the version of the SDK extension must match.

5. In file `contribution.xml`, replace the following attribute values in element `<component>`:
   - attribute `id` `ColoredBox` with `Box`
   - attribute `title` `Colored Box` with `Box`
6. In file `contribution.ztl`, replace `class com.sap.sample.coloredbox.ColoredBox...` with `class com.samplecompany.box.Box...`.
7. In file `component.js`, after `Component.subclass("...` replace `com.sap.sample.coloredbox.ColoredBox", ...` with `com.samplecompany.box.Box", ...`
8. In file `additional_properties_sheet.html`, replace `new com.sap.sample.coloredbox.ColoredBoxPropertyPage()` with `new com.samplecompany.box.BoxPropertyPage()`.
9. In file `additional_properties_sheet.js`, after `sap.designstudio.sdk.Component.subclass("...`, replace `com.sap.sample.coloredbox.ColoredBoxPropertyPage` with `com.samplecompany.box.BoxPropertyPage`.

Next Steps

To quickly reload the modified contents of your SDK component's `contribution.xml` and `contribution.ztl` (and JavaScript and CSS files) in Lumira Designer during development, follow these steps:

1. Activate the debug mode of Lumira Designer by pressing CTRL + SHIFT + ALT + D.
2. Choose *Tools* > *Refresh SDK Extensions*.
3. Deactivate the debug mode of Lumira Designer by pressing CTRL + SHIFT + ALT + D

3.3 Adding an SDK Extension to an SAP Lumira Designer Installation

Context

Adding an SDK extension to an SAP Lumira Designer installation enables you to create and execute local analysis applications, which contain components of this SDK extension.

Procedure

1. Pack the SDK extension into an archive file that can be installed in SAP Lumira Designer. This involves the following steps:
   - configuring the SDK extension *plug-in*
   - creating a *feature project* (wrapping the SDK extension),
   - creating a *category* (adding texts that represent the SDK extension in the Eclipse installation wizard), and
   - creating a *deployable feature* (wrapping the SDK extension and its category into an installable format).
2. Add the archive file containing the SDK extension to an SAP Lumira Designer installation.

Related Information

Configuring the SDK Extension Plug-In [page 15]
Creating a Feature Project [page 15]
Creating a Category [page 16]
Creating a Deployable Feature [page 16]

3.3.1 Configuring the SDK Extension Plug-In

Context

Procedure

1. Open the `plugin.xml` file of the SDK extension.
2. Choose the *Overview* tab.
3. In the *Version* input field, enter the version number **`1.0.0.qualifier`**.
4. Save your changes by pressing `Ctrl` + `S`.

3.3.2 Creating a Feature Project

Procedure

1. In your Eclipse IDE, choose *File* > *New* > *Project...*.
2. Choose *Plug-In Development* > *Feature Project*.
3. Choose *Next*.
4. Under *Project name*, enter the feature name, for example **`SampleExtensionFeature`**.
5. Choose *Finish*.
6. Select the *Included Plug-ins* tab and choose *Add...*.
7. Add your SDK extension, for example *com.sap.sample.coloredbox*.

   > → Tip
   >
   > Start typing a part of your SDK extension name in the text field. Your SDK extension appears in the list.

8. Unselect the *Unpack the plugin-archive after the installation* checkbox.
9. Save your changes (by pressing `CTRL` + `S`).

3.3.3 Creating a Category

Procedure

1. Choose *File* > *New* > *Other...*.
2. Choose *Plug-In Development* > *Category Definition*.
3. Choose *Next*.
4. Enter the feature that you created above, for example **`SampleExtensionFeature`**.
5. Choose *Finish*.
6. Choose *New Category*.
7. Under *ID**, enter the category ID **`com.sap.ip.bi.zen.sdk`**. This is the common feature ID of SDK extensions.
8. Choose *Add feature...*.
9. Select the feature that you created above, for example *SampleExtensionFeature*.
10. Choose *OK*.
11. Save your changes (by pressing CTRL + S).

3.3.4 Creating a Deployable Feature

Procedure

1. In the *Package Explorer*, select the created feature, for example *SampleExtensionFeature*.
2. Choose *File* > *Export...*.
3. Choose *Plug-in Development* > *Deployable features*.
4. Choose *Next*.
5. Under *Available Features*, select your feature, for example *SampleExtensionFeature*.
6. On the *Destination* tab, choose *Archive file* and enter the name of the archive file, for example **`C:\SampleExtension.zip`**.
7. On the *Options* tab, choose *Browse...* and select the category file of the feature, for example `C:\ds_sdk_workspace\SampleExtensionFeature\category.xml`.
8. Choose *OK*.
9. Choose *Finish*.

Results

The archive file is created, for example `C:\SampleExtension.zip`.

3.3.5 Installing Component SDK Extensions to SAP Lumira Designer

Context

You can add extensions developed with the Component SDK to your SAP Lumira Designer installation as new components.

Procedure

1. In SAP Lumira Designer, choose *Tools* > *Install Extension to Lumira Designer...*.
2. Depending on where the SDK extension is located, proceed as follows:
   - For locally saved extensions, choose *Archive...* and select the archive file containing the SDK extension, under `C:\SampleExtension.zip`, for example.
   - For extensions stored on a Web server, enter the URL of the Web server.
3. Choose *OK*.
4. Select the required feature, for example, *SampleExtensionFeature*.
5. Select the Component SDK extensions that you want to install.
6. Choose *Finish* to proceed with the installation.
7. Choose *Next* and again *Next* to confirm the installation.
8. Accept the terms of the license agreement and choose *Finish*.
9. Choose *Yes* to allow SAP Lumira Designer to restart.

Results

The SDK extension components appear in the *Components* view of SAP Lumira Designer as new components.

The components are stored under `<user home directory>\LumiraDesigner-config\plugins .`

3.4 Removing Extensions from SAP Lumira Designer

Context

You can remove SDK extensions that you have added to your SAP Lumira Designer installation as follows:

Procedure

1. In SAP Lumira Designer, choose *Help* > *About...*.
2. Click the *Installation Details* button.
3. Select the feature containing the SDK extension, for example, *SampleExtensionFeature*.
4. Choose *Uninstall...*.
5. In the *Uninstall* wizard, choose *Finish*.
6. Choose *Yes* to allow SAP Lumira Designer to restart.

Results

The SDK extension components are removed from the *Components* view of SAP Lumira Designer. Visualization SDK extensions are removed from the list in the *Additional Charts* dialog box.

3.5 Updating SDK Extensions of an SAP Lumira Designer Installation

Context

You can update SDK extensions in your SAP Lumira Designer installation as follows:

Procedure

1. Remove the old SDK extension.
2. Add the new SDK extension.

Related Information

Removing Extensions from SAP Lumira Designer [page 18]
Adding an SDK Extension to an SAP Lumira Designer Installation [page 14]

4 SDK Extensions

An SDK extension contains the following files (any other, more technical files are omitted):

File	required/optional	Description
Contribution XML file	required	Defines the SDK extension and its extension components
Component JavaScript file	optional	Implements an extension component's functional behavior (this includes creating its visual appearance)
Component CSS file	optional	Defines a Cascading Style Sheet (CSS) for an extension component
Icon file	optional	Represents an extension component's icon (16 x 16 pixels)
Script Contribution file	optional	Implements the methods that extension components contribute to the Lumira Designer script editor
Additional Properties Sheet HTML file	optional	Implements the visual appearance of an extension component's Additional Properties Sheet
Additional Properties Sheet JavaScript file	optional	Implements the functional behavior of an extension component's Additional Properties Sheet

The following documentation chapters explain these files in detail. The examples are taken from the Sample SDK Extension **Colored Box**.

4.1 Contribution XML

The Contribution XML file specifies the SDK extension and all its extension components. SAP provides a documented XML schema definition file (`sdk.xds`) that defines the format of the Contribution XML file.

The example below is the Contribution XML of the SDK extension **Colored Box**. The file specifies the title, version, vendor name, as well as an extension namespace. The SDK extension contains one extension component. Its ID is `ColoredBox` (which is internally combined with the SDK extension's namespace to create the unique extension component ID `com.sap.sample.coloredbox.ColoredBox`). The extension component has a title, an Additional Properties Sheet, it references an icon and so on. The extension component also references its Component JavaScript file, defines two properties (`color` and `onClick`, which automatically appear in Lumira Designer's *Properties* view), and various initial values.

Example

(File `contribution.xml`)

```
<?xml version="1.0" encoding="UTF-8"?>
<sdkExtension xmlns="http://www.sap.com/bi/zen/sdk"
  title="Component SDK Extension Sample Colored Box">
  version="1.0"
  vendor="SAP"
  id="com.sap.sample.coloredbox"
  <component
    id="ColoredBox"
    title="Colored Box"
    icon="res/icon.png"
    handlerType="div"
    modes="commons m"
    propertySheetPath="res/additional_properties_sheet/
additional_properties_sheet.html">
    <requireJs modes="commons m">res/js/component</requireJs>
    <property id="color" type="Color" title="Color" group="Display" />
    <property id="onclick" type="ScriptText" title="On Click" group="Events" />
    <initialization>
      <defaultValue property="LEFT_MARGIN">40</defaultValue>
      <defaultValue property="TOP_MARGIN">40</defaultValue>
      <defaultValue property="WIDTH">100</defaultValue>
      <defaultValue property="HEIGHT">100</defaultValue>
      <defaultValue property="color">red</defaultValue>
    </initialization>
  </component>
</sdkExtension>
```

4.1.1 Elements of the Contribution XML File

See the XML schema definition file `sdk.xsd` for full details of what can be defined in the Contribution XML. Its elements are listed below.

> **i** Note
>
> Element names, attribute names, attribute values, and file paths used in the Contribution XML are case-sensitive.

> **→** Tip
>
> You see the XML schema definition file when you have downloaded and extracted the *Component SDK Templates and Samples* on SAP Help Portal at http://help.sap.com.

Element <sdkExtension>

Specifies an SDK extension. Its attributes are:

Attribute	Required/Optional	Description
`title`	required	Title of the SDK extension
`version`	required	Version number in major.minor format, for example "1.0".
`vendor`	required	Vendor name
`eula`	optional	End user license agreement text
`id`	required	Specifies an SDK extension ID to avoid name conflicts between an SDK extension (and its extension components) and other SDK extensions (and their extension components). The specified string is combined with extension component IDs in this SDK extension, to create a unique extension component ID. Use a Java-like package notation, for example, `com.samplecompany`. Use lowercase letters, digits, and a period (.) as a delimiter.

Child elements are (in the following order):

Element	Cardinality	Description
`license`	0..1	License text
`group`	0..*	Custom group (see Element <group> [page 23])
`component`	0..*	Extension components (see Element <component> [page 23])

Element <group>

Specifies a custom group in Lumira Designer's views. A custom group in the *Component* view contains extension components. A custom group in the *Properties* view contains properties of an extension component. Its attributes are:

Attribute	Required/Optional	Description
`id`	required	ID of the custom group **i Note** Lowercase and uppercase letters are treated the same.
`title`	required	Title of the custom group
`tooltip`	optional	Tooltip of the custom group
`visible`	optional	If true, then the group is visible (default setting: `true`).

Element <component>

Specifies an extension component. Its attributes are:

Attribute	Required/Optional	Description
`id`	required	ID of the extension component **i Note** The ID must not end with the string `"Array"`.
`title`	required	Title of the extension component
`tooltip`	optional	Tooltip of the extension component
`visible`	optional	If `true` then the extension component is visible in the Lumira Designer *Components* view (default setting: `true`).

Attribute	Required/Optional	Description
`group`	optional	Group in the Lumira Designer's *Component* view, where this extension component is displayed. Specify a custom group you have defined in this SDK extension by the group's ID. If no custom group is specified, this extension component is placed in the default Custom Component group. **i Note** In group IDs, lowercase and uppercase letters are treated the same. The group ID `TECHNICAL_COMPONENTS` indicates that this extension component is a technical component. Unlike non-technical components, it is displayed in Lumira Designer's *Outline* view when you select the folder *Technical Components* and choose *Create Child* in the context menu. Technical components are not intended to be used for being rendered. Thus, it does not make sense to use inherited properties like `WIDTH`, `HEIGHT` and margins with technical components and there is no need to initialize these properties in `<defaultValue>` elements in the `contribution.xml` file. In addition, their Script Contribution file `contribution.ztl` should not extend `Component` to forbid Lumira Designer scripts access to these properties.
`propertySheetPath`	optional	References the HTML file of the Additional Properties Sheet. This file must be located in the `/res` folder of the extension component.
`databound`	optional	Indicates that this extension component is data-bound (uses data sources) (default setting: `false`).

Attribute	Required/Optional	Description
`newInstancePrefix`	optional	Prefix for the name of a newly created instance of this extension component. If this attribute is not specified, then a default name in the form "extension component type (uppercase) + number" is used, for example `COLOREDBOX_1`.
`handlerType`	optional	Specifies the technology that implements this extension component. Specify one of the following values: `div`, `sapui5`, `datasource` (default setting: `div`). **i Note** The value `datasource` marks this extension component as an SDK data source (see SDK Extensions as Data Sources (Data Source SDK) [page 112].
`icon`	optional	References an icon (16 x 16 pixels) displayed with this extension component in Lumira Designer's *Component* and *Outline* views. The path is relative to the root folder of the SDK extension.

Attribute	Required/Optional	Description
`modes`	optional	Indicates which SAPUI5 libraries this extension component supports. This extension component is only shown in Lumira Designer's *Component* view when you are editing analysis applications based on supported SAPUI5 libraries. This attribute is relevant for extension components with `handlerType` of `sapui5`. It is not so relevant for `div`, unless the components are based on SAPUI5 libraries. **i Note** This attribute is ignored for extension components with a `handlerType` of `datasource` (Lumira Designer SDK Data Sources) Specify one or more of the following values separated by a space: `commons`, `m` (default setting: `commons`).
`cardinality`	optional	Indicates how many instances of this extension component can be created. This only applies to extension components that are technical components. See attribute group. Specify one of the following values: `0_1` (one instance), `0_n` (many instances) (default setting: `0_n`).
`supportsExportContent`	optional	If `true` then the extension component can be exported by the PDF export framework (default setting: `false`). For more information on exporting extension components by the PDF export framework, see Exporting an SDK Extension Component [page 70].

Attribute	Required/Optional	Description
`loadIncludesOnlyIfVisible`	optional	If `true` then the extension component loads the files to be included with this component when the component becomes visible at run time (default setting: `true`).
`includeInBookmarkDialog`	optional	Indicates whether this extension component is included in the *Edit Bookmark Definition* dialog box (default setting: `true`).

Child elements are (in the following order):

Element	Cardinality	Description
`requireJs`	0..*	References a resource file to be loaded with this extension component at runtime. This is typically the Component JavaScript file, for example `contribution.js` of this extension component. The reference is a path relative to the root folder of the SDK extension or a fully qualified URL. This element combines and replaces the functionality of elements `<stdInclude>`, `<jsInclude>`, and `<cssInclude>` (see Element <requireJs> [page 29])
`stdInclude`	0..*	Includes a JavaScript framework at runtime (see Element <stdInclude> [page 29]). This element is deprecated, see also: Loading Resources in a Specific Order [page 45].

Element	Cardinality	Description
`jsInclude`	0..*	References a JavaScript file to be included with this extension component at runtime. It is either a relative path to the root folder of the SDK extension or a fully qualified URL. **i Note** It is not necessary to include the following JavaScript frameworks with this element: • jQuery • underscore They are already included in the SDK framework. **i Note** It is not necessary to include the D3 JavaScript framework with this element. See Element <stdInclude> [page 29] for more information. This element is deprecated, see also: Element <requireJs> [page 29] .
`cssInclude`	0..*	References a CSS file to be included with this extension component at runtime. It is either a relative path to the root folder of the SDK extension or a fully qualified URL. This element is deprecated, see also: Loading Resources in a Specific Order [page 45].
`property`	0..*	Property of the extension component (see Element <property> [page 30])
`initialization`	0..1	Initialization values of properties (see Element <initialization> [page 36])
`supportedBackend`	0..*	Specifies which platform this extension component supports. Specify one of the following values: `LUMX`, `BIPLATFORM` or `LOCAL`. If this element is not specified, then all platforms support this extension component.

Element <stdInclude>

Includes a JavaScript framework. Its attributes are:

Attribute	Required/Optional	Description
`kind`	required	JavaScript framework to include at runtime. Specify one of the following values: `d3`, `cvom`.

This element is deprecated, see also: Loading Resources in a Specific Order [page 45].

Element <requireJs>

References a resource file to be loaded with this extension component at runtime. This is typically the Component JavaScript file, for example `contribution.js`, of this extension component. The reference is a path relative to the root folder of the SDK extension or a fully qualified URL.

This element combines and replaces the functionality of elements `<stdInclude>`, `<jsInclude>`, and `<cssInclude>`. For more information, see Loading Resources in a Specific Order [page 45].

> **i Note**
>
> When referencing a JavaScript file, omit the `.js` file extension.

Its attributes are:

Attribute	Required/Optional	Description
`modes`	required	Indicates which SAPUI5 libraries this resource supports. This resource is only loaded when the analysis application that hosts this extension component is based on supported SAPUI5 libraries. Specify one or more of the following values separated by a space: `commons`, `m`. Example: You have two different Component Javascript files. One of them should be used with analysis applications based on the SAPUI5 library, the other one with the SAPUI5 m library. Specify the first Component JavaScript file with element `<requireJs modes="commons">`, and the other one with element `<requireJs modes="m">`.

In this example, the Component JavaScript file `component.js` of the `ColoredBox` sample is referenced. It is located in folder `res/js` of the extension component. It is used with the SAPUI5 and SAPUI5 m libraries.

Sample Code

```
<requireJs modes="commons m">res/js/component</requireJs>
```

Element <property>

Specifies an extension component property. Its attributes are:

Attribute	Required/Optional	Description
`id`	required	ID of the property → Tip Use IDs with a lowercase first letter.
`title`	required	Title of the property

Attribute	Required/Optional	Description
`tooltip`	optional	Tooltip of the property
`visible`	optional	If `true` then the property is visible in Lumira Designer (default setting: `true`)

Attribute	Required/Optional	Description
`type`	required	Type of the property. Specify one of the following: • `int` • `float` • `boolean` • `String` • `ScriptText` • `Color` • `Url` • `ResultCell` • `ResultCellList` • `ResultCellSet` • `ResultSet` • `MultiLineText` • `Array` • `Object` Properties of certain types have a matching property dialog box (value help) in Lumira Designer's *Properties* view. **i Note** • The type `Text` marks the property as a translatable text. • The type `MultiLineText` marks the property as a translatable, multi-line text. • Properties of type `Url`, `ResultCell`, `ResultCellList`, `ResultCellSet`, and `ResultSet` may contain nested options (see Element <option> [page 37]). • The type `Array` marks the property as an array of properties, for example of type `int`, `String`, but also `Object`. Arrays are stored in the property in the usual JavaScript notation.

Attribute	Required/Optional	Description

		In the following example, the property `names` stores an array of `String` elements:
		Sample Code
		`<property id="names" type="Array" title="Names"> <property id="name" type="String" title="Name" /> </property>`
		In the following example, the property `persons` stores an array of `Object` elements representing a person:
		Sample Code
		`<property id="persons" type="Array" title="Persons"> <property id="person" type="Object" title="Person" > <property id="name" type="String" title="Name" /> <property id="age" type="int" title="Age" /> </property> </property>`
		Properties nested in an `Array` property may contain nested options (see Element <option> [page 37]) to enable input validation in the dialog box (value help) of Lumira Designer's *Properties* view.

Attribute	Required/Optional	Description
		• The type `Object` marks the property as an object containing a nested structure of primitive properties like `int`, `String`, and so on, **but not** `Object`, `ResultCell`, `ResultCellList`, `ResultCellSet`, `ResultSet`, and `ScriptableText`. Objects are stored in the property in the usual JavaScript JSON notation. In the following example the property `person` stores information about a person: **Sample Code** `<property id="person" type="Object" title="Person" > <property id="name" type="String" title="Name" /> <property id="age" type="int" title="Age" /> <property id="city" type="String" title="City" /> </property>` Properties nested in an `Object` property may contain nested options (see Element <option> [page 37]) to enable input validation in the dialog box (value help) of Lumira Designer's *Properties* view.

Attribute	Required/Optional	Description
`group`	optional	Group in the Lumira Designer's *Properties* view where this property is displayed. Specify a custom group you have defined in this SDK extension by the group's ID or one of the following values: `Display`, `DataBinding`, or `Events` (default setting: `Display`).
`bindable`	optional	If `true` then the property can be bound in Lumira Designer's *Properties* view using property binding (not to be confused with SDK data-binding) (default setting: `false`). **i Note** This does not apply to technical components (a technical component contains the entry `group=TECHNICAL_COMPONENTS` in its `contribution.xml` file).
`modes`	optional	Indicates which SAPUI5 libraries this property supports. This property is only shown in Lumira Designer's *Properties* view when you are editing analysis applications based on supported SAPUI5 libraries. This attribute is relevant for components with `handlerType` of `sapui5`. It is not so relevant for `div`, unless the components are based on SAPUI5 libraries. Specify one or more of the following values separated by a space: `commons`, `m` (default setting: `commons m`).

Child elements are (in the following order):

Element	Cardinality	Description
`property`	0..*	Nested property of an Array or Object property (see Element <option> [page 37])

Element	Cardinality	Description
`possibleValue`	0..*	Contains a possible value of this property. Use multiple elements to create an enumeration of possible values for this property.
`option`	0..*	Contains options for data-bound properties of type `ResultCell`, `ResultCellList`, `ResultCellSet` or `ResultSet` (see Element <option> [page 37]). Contains options for input validation in Lumira Designer's *Properties* view dialog box (value help), for example, for properties nested in `Array` or `Object` properties (see Element <option> [page 37]).

Element <initialization>

Initial values of properties (predefined and custom) for this extension component, when a new instance of this extension component is created.

Child elements are:

Element	Cardinality	Description
`defaultValue`	0..*	Default values of properties (see Element <defaultValue> [page 36]).

Element <defaultValue>

This element contains a default value of a property (predefined and custom) for the extension component, when a new instance of this extension component is created. Its attribute values are:

Attribute	Required/Optional	Description
`property`	required	Property ID

Element <possibleValue>

This element contains a possible value of a property. Its attribute values are:

Attribute	Required/Optional	Description
`title`	optional	Title of the possible value displayed in the Lumira Designer *Properties* view.

Element <option>

This element contains specific additional information of a property. Its attributes are:

Attribute	Required/Optional	Description
`name`	required	Option name (see table below)
`value`	required	Option value (see table below)

The following table lists the available option names of data-bound properties `ResultCell`, `ResultCellList`, `ResultCellSet`, and `ResultSet` to fine tune the content and size of the Data Runtime JSON and Metadata Runtime JSON returned by the SDK framework (see “MetadataRuntime JSON” and “Data Runtime JSON” under Runtime Property Values [page 73]):

Option Name	Description
`includeAxesTuples`	If `true` then the JSON properties `axis_rows` and `axis_columns` are included in the Data Runtime JSON. They contain the tuples of the row axis and column axis. The following table lists the default setting depending on the data-bound property type: **ResultCell** / **ResultCell-List** / **ResultCell-Set** / **ResultSet** `false` / `false` / `false` / `true`
`includeTuples`	If `true` then the JSON property `tuples` is included in the Data Runtime JSON. It contains the tuples of the data (default setting: `true`).
`includeResults`	If `true` then the result values, for example totals, are included in the Data Runtime JSON (default setting: `true`).
`presentationDelimiter`	String that separates presentations of dimension member values in the `text` JSON property of dimension members in the Metadata Runtime JSON (default setting: `|`).

<table>
<tr><th>Option Name</th><th>Description</th></tr>
<tr><td>selectionShape</td><td>Integer value that indicates the geometry of the data in the Data Runtime JSON. Possible values: 0 (ResultCell), 1 (ResultCellList), 2 (ResultCellSet or ResultSet).

The following table lists the default setting depending on the type of the data-bound property:

<table>
<tr><th>ResultCell</th><th>ResultCell-List</th><th>ResultCell-Set</th><th>ResultSet</th></tr>
<tr><td>0</td><td>1</td><td>2</td><td>2</td></tr>
</table>

i Note
The value of selectionShape corresponds to the type of a data-bound property. By explicitly setting a value of selectionShape, you basically overrule the type of the data-bound property.

Possible value: 3 (Master Data)

This is the same as value 2 with additional support of data from master data data sources. A master data data source contains dimension member values on one axis (either row or column axis) and no key figures. For more information on master data data sources, see Master Data [page 92].</td></tr>
<tr><td>swapAxes</td><td>If true then the axes (and the relevant data) are swapped (transposed) in the Data Runtime JSON and Metadata Runtime JSON (default setting: false).</td></tr>
<tr><td>includeData</td><td>If true then the JSON property data is included in the Data Runtime JSON. It contains the data values (float numbers or null) (default setting: true).</td></tr>
<tr><td>includeFormattedData</td><td>If true then the JSON property formattedData is included in the Data Runtime JSON. It contains the formatted data values as strings (default setting: false).</td></tr>
<tr><td>includeMetadata</td><td>If true then the Metadata Runtime JSON is included as a part of the Data Runtime JSON.

The following table lists the default setting depending on the data-bound property type:

<table>
<tr><th>ResultCell</th><th>ResultCell-List</th><th>ResultCell-Set</th><th>ResultSet</th></tr>
<tr><td>false</td><td>false</td><td>false</td><td>true</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Option Name</th><th>Description</th></tr>
<tr><td>fillMetadataProperty</td><td>If true then the SDK component's implicit property metadata contains the Metadata Runtime JSON.

The following table lists the default setting depending on the data-bound property type:

<table><tr><th>ResultCell</th><th>ResultCell-List</th><th>ResultCell-Set</th><th>ResultSet</th></tr><tr><td>true</td><td>true</td><td>true</td><td>false</td></tr></table></td></tr>
<tr><td>includeAttributes</td><td>If true then the JSON properties attributes and attributeMembers are added to the Metadata Runtime JSON (default setting: false). They contain information about the display attributes of a result set. If the result set does not contain attributes then these JSON properties are not added, regardless of the value of includeAttributes.</td></tr>
<tr><td>includeConditionalFormats</td><td>If true then the JSON property conditionalFormats is added to the Metadata Runtime JSON and the JSON property conditionalFormatValues is added to the Data Runtime JSON (default setting: false). They contain information about conditional formatting of data values of the result set. If the result set does not contain conditional formats then these JSON properties are not added, regardless of the value of includeConditionalFormats.</td></tr>
<tr><td>allDataOnEmptySelection</td><td>If true then an empty selection string ("" or {}) set to this data-bound property returns the entire result set in the Metadata and Data Runtime JSONs (default setting: true). If false then an empty selection string ("" or {}) set to this data-bound property returns a Metadata Runtime JSON with no dimension information and an empty Data Runtime JSON ("").</td></tr>
</table>

Option Name	Description
`maxCells`	The maximum number of selected result set cells that are sent to this data-bound property (default setting: `10000`). If the number of selected result set cells is greater than the maximum number, then no result set cells are sent to this data-bound property. **i Note** You can also globally set the maximum number of selected result set cells on BW systems (and SAP R/3 systems in general) with the RSADMIN parameter `AAD_SDK_MAX_CELLS` (default value: `50000`). If both a value for the `maxCells` option and the RSADMIN parameter `AAD_SDK_MAX_CELLS` have been set, then the lower value is used. **⚠ Caution** Keep in mind that increasing the maximum number of selected result set cells that are sent to data-bound properties can severely degrade the performance of your application: Not only is the amount of data larger that is sent over the network to the browser, but also the memory consumption and processing load of the browser is increased. If the performance of your analysis application is too slow, check if the maximum number of selected result set cells has been changed - either with the `maxCells` option or the RSADMIN parameter `AAD_SDK_MAX_CELLS` in the relevant BW system.

<table>
<tr><th>Option Name</th><th>Description</th></tr>
<tr><td>includeDataSourceInfo</td><td>If true then the JSON property dataSourceInfo is included in the Data Runtime JSON (default setting: false). This property contains the following JSON properties.<table>
<tr><th>Property Name</th><th>Description</th><th>Example</th></tr>
<tr><td>dataSourceComponent</td><td>Name of the data source component</td><td>DS_1</td></tr>
<tr><td>name</td><td>Name of the data source. In the case of a blended data source, this is an array containing the names of all blended data sources.</td><td>FINANCIALS_QUERY</td></tr>
<tr><td>connection</td><td>Name of the connection</td><td>AB1</td></tr>
<tr><td>type</td><td>Type of the data source, for example QUERY or INFOPROVIDER</td><td>QUERY</td></tr>
</table></td></tr>
<tr><td>repeatHierarchyNodes</td><td>If true and the data source contains an active hierarchy, then the Metadata Runtime JSON may contain hierarchy members with the same ID (default setting: false).</td></tr>
<tr><td>useResultWhenUnspecified</td><td>If true then the SDK framework will pick the aggregate member of that dimension for unspecified dimensions in an underspecified selection string (default setting: false).</td></tr>
</table>

Option Name	Description

`includeAdditionalResults`	If `true` then for each dimension with one or more additional results, the Metadata Runtime JSON contains specific dimension members for these additional results (default setting: `false`).
	
	These specific dimension members have a `type` attribute of `"RESULT"`, the key attribute has one of the values `"COUNT"`, `"SUM"`, `"AVERAGE"`, `"MIN"`, or `"MAX"`, depending on the aggregation mode of the additional results.
	
	In the following example, a snippet from a Metadata Runtime JSON shows a dimension member of a country dimension (with key `8050322`), followed by two dimension members of additional results (with keys `"COUNT"` and `"AVERAGE"`). The first dimension member of additional results (with key `"COUNT"`) uses a count of members as its aggregation mode, the second dimension member of additional results (with key `"AVERAGE"`) uses an average as its aggregation mode.
	
	Sample Code
	
	`...`
	`{`
	`  "key": "8050322",`
	`  "text": "Italy"`
	`},`
	`{`
	`  "key": "COUNT",`
	`  "text": "Count",`
	`  "type": "RESULT"`
	`},`
	`{`
	`  "key": "AVERAGE",`
	`  "text": "Average",`
	`  "type": "RESULT"`
	`},`
	`...`
	
	Also, the Data Runtime JSON contains the corresponding data values for the dimension members of the additional results in the JSON properties `tuples`, `data`, and `formattedData`.
	
	Caution
	
	When one of the values `"COUNT"`, `"SUM"`, `"AVERAGE"`, `"MIN"`, or `"MAX"` is used as `key` attributes in a selection string (see Design Time Property Values [page 72]), they may not lead to a unique selection

Option Name	Description
	when your data source has regular dimension members with the same `key` attribute. **i Note** The options `includeAdditionResults` and `includeResults` are completely independent from each other.
`includeAllDimensionsAndMeasures`	If `true`, then all dimensions and members are included in the Data Runtime JSON (default setting: `false`). Dimensions that are neither on the row nor on the column axis but are on the free axis are appended to the array of dimensions of the JSON property `externalDimensions`. Measures that are not part of the result set are added to the array of members of the JSON property members. Such members contain the nested JSON property `isExcluded` with a value of `true`.
`keyfield`	If `true` and the corresponding property is part of an `Array` property, then Lumira Designer checks if the property value is unique when it is entered in Lumira Designer's *Properties* view dialog box (value help). Lumira Designer's value help does not accept the property value if another property of the array has the same value (default setting: `false`).
`optional`	If `true` then Lumira Designer accepts an empty or no property value when entering it in Lumira Designer's *Properties* view dialog box (value help) (default setting: `false`).
`kind`	Indicates the MIME type of properties of type `Url` so that Lumira Designer's *Properties* view dialog box (value help) can provide the appropriate value help dialog box. Specify one of the following: `GeoJSON`, `CSS`, `Image`, `Font`, `CSV` or `SVG` for the `value` attribute.
`minValue`	Minimum value of a property of type `int` or `float`. If you enter a value for this property (in Lumira Designer's *Properties* view) that is less than this value, then an error message appears in the status bar at the bottom of the screen in Lumira Designer and the entered value is discarded.
`maxValue`	Maximum value of a property of type `int` or `float`. If you enter a value for this property (in Lumira Designer's *Properties* view) that is greater than this value, then an error message appears in the status bar at the bottom of the screen in Lumira Designer and the entered value is discarded.

Option Name	Description
`type`	If `true` and the corresponding property is of type `ComponentReference`, then the value indicates the component type of referenced component, for example `com.sap.sample.coloredbox.ColoredBox`.

4.2 Component JavaScript

You implement a Component JavaScript class for each extension component. You can implement the Component JavaScript class using both JavaScript and jQuery, as jQuery is included in the Design Studio SDK framework. The Lumira Component SDK includes jQuery 2.2.3.

The **class name** of the JavaScript class is the combination of the SDK extension namespace and the extension component ID, for example `com.sap.sample.coloredbox.ColoredBox`.

Below is an example; the Component JavaScript class of the extension component **Colored Box**. Its class name is `com.sap.sample.coloredbox.ColoredBox` and subclasses the generic JavaScript class `sap.designstudio.sdk.Component`. It implements an `init` function, which adds a CSS style `coloredBox` and attaches an event handler to the click event of the extension component. When clicked, the extension component executes the script assigned to the extension component property `onclick`. It also defines a `color` function that acts as a combined setter and getter function for the extension component property `color`; in other words, the function sets and gets the background color of the extension component.

Example

(File: `component.js`)

```
define(["sap/designstudio/sdk/component", "css!../css/component.css"],
function(Component, css) {
  Component.subclass("com.sap.sample.coloredbox.ColoredBox", function() {

    var that = this;

    this.init = function() {
      this.$().addClass("coloredBox");
      this.$().click(function() {
        that.fireEvent("onclick");
      });
    };

    this.color = function(value) {
      if (value === undefined) {
        return this.$().css("background-color");
      } else {
        this.$().css("background-color", value);
        return this;
      }
    };
  });
```

```
});
```

The code in the Component JavaScript class controls important aspects of an extension component:

- loading resources in a specific order
- creating the HTML of the extension component
- getting and setting extension component properties
- firing events

Related Information

Loading Resources in a Specific Order [page 45]
Creating the HTML of the Extension Component [page 48]
Getting and Setting Extension Component Properties [page 69]
Events [page 55]

4.2.1 Loading Resources in a Specific Order

The SDK framework lets you specify the order in which resource files of your extension component, like JavaScript and CSS files, are loaded before the Component JavaScript is executed. The SDK framework uses the loading mechanism of the RequireJS library, which is included with the SDK.

The loading order is defined by the `define` function of RequireJS. In the SDK, this function is used with the following syntax:

```
define(["sap/designstudio/sdk/component", sResourcePath1, sResourcePath2, ...],
function(Component, ref1, ref2, ...) {
  Component.subclass(sExtensionId.sComponentId, function() {
    ...
  });
});
```

The first argument of the `define` function is an array of resource paths. They are loaded in the order in which they are listed. The first resource path `"sap/designstudio/sdk/component"` is an alias of the JavaScript file of the parent class of your SDK component's JavaScript class. It is always present.

You can add zero, one, or multiple resource paths `sResourcePath1, sResourcePath2,...`. Depending on the type of resource, certain rules may apply to the resource paths:

- For JavaScript files, see Loading JavaScript Files as Resources [page 46].
- For CSS files, see Loading CSS Files as Resources [page 47].
- For standard JavaScript frameworks, see Loading Standard JavaScript Frameworks as Resources [page 48].

The second argument of the `define` function is an anonymous function, which extends your SDK component's JavaScript class as a subclass of its parent JavaScript class.

The SDK framework passes references to the loaded resources as arguments to the anonymous function. The order of the arguments corresponds to the order of the resource paths in the array. For example, in the first

argument `Component`, the SDK framework passes a reference to the parent class of your SDK component's JavaScript class.

The actual class extension is achieved by calling the `subclass` function and passing the fully qualified component ID of your SDK component and a function that contains the actual Component JavaScript code of your SDK component.

> **i Note**
>
> When using RequireJS, it is good practice to provide a matching argument in the function signature for each resource listed in the array. This enables you to access all the loaded resources in the Component JavaScript body.

4.2.1.1 Loading JavaScript Files as Resources

JavaScript files are loaded using RequireJS by adding the file resource path to the resource path array of the `define` function.

> **i Note**
>
> Relative resource paths to a JavaScript file are relative to the folder of the Component JavaScript file. Relative resource paths must start with `./ or ../`.

> **i Note**
>
> At the end of the resource path, omit the `.js` extension.

In the following example, a JavaScript file `sample.js`, located in the `res/js/folder1` folder of the `ColoredBox` sample component, is loaded in the Component JavaScript file `contribution.js`.

> **‹≡› Sample Code**

```
define(["sap/designstudio/sdk/component", ..., "./folder1/sample"],
function(Component, ..., sample) {
  Component.subclass("com.sap.sample.coloredbox.ColoredBox", function() {
    // ...
  });
define(["sap/designstudio/sdk/component", ..., "./folder1/sample"],
function(Component, ..., sample) {
  Component.subclass("com.sap.sample.coloredbox.ColoredBox", function() {
    // ...
  });
});
});
```

> **→ Tip**
>
> **How to Call Functions of Your JavaScript Resources**
>
> In the Component JavaScript of your SDK component, you want to call a function of one of your JavaScript resources.

The following example shows you how to call the function `greet` of your JavaScript resource file `sample.js` in the Component JavaScript file `contribution.js` of the `ColoredBox` sample component. Note how the resource's path in the resource path array of the `define` function indicates that the JavaScript resource file `sample.js` is located in the folder `res/js` of the `ColoredBox` sample component. Note that the extension `.js.` is omitted in the resource path. Note also the argument `sample` in the anonymous function. This argument receives a reference to the JavaScript resource at runtime, which is then used in the code to call the `greet` function.

File `contribution.js`

Sample Code

```
define(["sap/designstudio/sdk/component", ..., "./sample"],
function(Component, ..., sample) {
  Component.subclass("com.sap.sample.coloredbox.ColoredBox", function() {
    // ...
    this.init = function() {
      sample.greet("Hello, world!");
      // ...
    }
    // ...
  });
});
```

This is the implemetation of the `greet` function in the JavaScript resource file `sample.js`.

File `sample.js`

Sample Code

```
defined([], function() {
  var result = {};
  result.greet = function(message) {
    alert(message);
  }
  return result;
});
```

4.2.1.2 Loading CSS Files as Resources

CSS files are loaded using RequireJS by prepending the file resource path with the string `"css!"` in the resource path array of the `define` function.

Note

Relative resource paths to a CSS file are relative to the folder of the Component JavaScript file. Relative resource paths must start with `./ or ../`.

In the following example, a CSS file is loaded in the Component JavaScript file `contribution.js` of the `ColoredBox` sample component. Note the resource path of the CSS file. A `css` argument was added to the signature of the anonymous function, as it is good practice with RequireJS (although it is not used in this JavaScript code).

Sample Code

```
define(["sap/designstudio/sdk/component", "css!../css/component.css"],
function(Component, css) {
  Component.subclass("com.sap.sample.coloredbox.ColoredBox", function() {
    ...
  });
});
```

4.2.1.3 Loading Standard JavaScript Frameworks as Resources

The d3 JavaScript library is included with the SDK and is loaded with the resource path `"d3"` using RequireJS.

In the following example, the d3 JavaScript library is loaded in the Component JavaScript file `contribution.js` of the `Sparkline` sample component.

Sample Code

```
define(["sap/designstudio/sdk/component", "d3"], function(Component, d3) {
  Component.subclass("com.sap.sample.sparkline.Sparkline", function() {
    // ...
    var graph = d3.select(...)....
    // ...
  });
});
```

4.2.2 Creating the HTML of the Extension Component

You create the HTML of the extension component in the Component JavaScript.

At runtime the SDK framework provides a `<div>` element, which acts as a root element. The HTML of the extension component can then be placed into this element. You access this root element as a jQuery object with `this.$()`.

Caution

Ensure that your extension component does not rely on HTML DOM content outside the provided `<div>` element and only modifies HTML DOM content inside the provided `<div>` element. Otherwise your extension component may not work in future Lumira Designer versions as the HTML DOM outside the provided `<div>` element is subject to change without further notice.

Extension Component Lifecycle

When the extension component is rendered for the first time, the SDK framework performs the following sequence of JavaScript function calls:

- `init()`
- `beforeUpdate()`
- Update all extension component properties using their setter/getter functions (see next section)
- `afterUpdate()`

When the extension component is only updated (after it has already been rendered once), the SDK framework performs the following sequence of JavaScript function calls:

- `beforeUpdate()`
- Update all extension component properties using their setter/getter functions (see next section)
- `afterUpdate()`

When the extension component is deleted from the application, the SDK framework calls JavaScript function

- `componentDeleted()`

> **i** Note
>
> If you want to run certain parts of JavaScript code of your extension component only when the extension component is in the Lumira Designer canvas (as opposed to an analysis application in a browser), then nest your code in the following JavaScript condition:

```
if (window.sap && sap.zen && sap.zen.designmode) {
  // ...
}
```

> **→** Tip
>
> If you want to find out if *Authoring* mode is enabled, then use the following function:

```
sap.zen.designmode.isRuntimeAuthoringMode()
```

The function returns `true` if all of the following checks are met:

- The function `sap.zen.designmode.isEmbeddedDesignMode()` returns `false`
- The property *Authoring Area* on the *Authoring* component is set
- The property *Enabled* on the *Authoring* component is set to *true*
  A component can be outside the authoring area and thus not in *Authoring* mode when the above function returns true. This can be checked by passing the component to the `isRuntimeAuthoringMode()` function:

  ```
  sap.zen.designmode.isRuntimeAuthoringMode(component)
  ```

The function returns `true` if all the following checks are met:

- The function `sap.zen.designmode.isEmbeddedDesignMode()` returns `false`
- The property *Authoring Area* on the authoring component is set
- The property *Enabled* on the Authoring component is set to *true*

- The passed in component is a child of the composite set in the property *Authoring Area*

Related Information

JavaScript Function Calls [page 50]

4.2.2.1 JavaScript Function Calls

Function `init`

Syntax: `init()`

Implement this function to execute JavaScript code after the extension component's root `<div>` element has been created.

> **i** Note
>
> If your extension component is a technical component (it contains the entry `group=TECHNICAL_COMPONENTS` in its `contribution.xml` file), use the following `init` function to avoid rendering the extension component:

```
this.init = function() {
  this.$().css("display", "none");
}
```

Function `beforeUpdate`

Syntax: `beforeUpdate()`

Implement this function to execute JavaScript code before the properties of the extension component are updated.

Property Getter and Setter Functions

For each extension property, you can implement a function that acts as a combined setter and getter function.

- The **function name** is the property's name.
- The function's **setter clause** must return `this` to allow function calls to be chained, thus creating a fluent interface.

Example: (File `component.js`)

```
this.color = function(value) {
  if (value === undefined) {
    return this.$().css("background-color");
  } else {
    this.$().css("background-color", value);
    return this;
  }
};
```

- Properties of type `Array` can be accessed like normal JavaScript arrays.
  In the following example, a property containing an array of city names is defined in the Contribution XML:

  ```
  <property id="cities" type="Array" title="Cities">
    <property id="city" type="String" title="City"/>
  </property>
  ```

  The following excerpt of a Component JavaScript shows how to implement the property getter and setter for the array property `cities`. When setting the property `cities`, the passed array, for example [`"Cairo"`, `"Moscow"`, `"New York"`, `"Sydney"`, `"Tokyo"`], is stored in the variable `aCities` in the browser.

  ```
  var aCities;

  this.cities = function(value) {
    if (value === undefined) {
      return this.aCities;
    } else {
      this.aCities = value;
      return this;
    }
  };
  ```

  With the following functions you can get or set a city name within the Component JavaScript. Note that `setCity` changes the value in the browser only but not yet in the runtime.

  ```
  function getCity(index) {
    return this.aCities[index];
  }

  function setCity(value, index) {
    this.aCities[index] = value;
  }
  ```

- Properties of type `Object` can be accessed like normal JavaScript objects.
  In the following example, a property containing an object with information about a person is defined in the Contribution XML:

  ```
  <property id="person" type="Object" title="Person">
    <property id="name" type="String" title="Name" />
    <property id="age" type="int" title="Age" />
    <property id="city" type="String" title="City" />
  </property>
  ```

  The following excerpt of a Component JavaScript shows how to implement the property getter and setter for the object property `person`. When setting the property `person`, the passed JSON, for example {`"name"`: `"John"`, `"age"`: `"35"`, `"city"`: `"London"`}, is stored in the variable `oPerson` in the browser.

  ```
  var oPerson;

  this.person = function(value) {
  ```

```
  if (value === undefined) {
    return this.oPerson;
  } else {
    this.oPerson = value;
    return this;
  }
};
```

With the following functions you can get or set an element of the object, the person's name, within the Component JavaScript. Note that `setName` changes the value in the browser only but not yet in the runtime.

```
function getName() {
  return this.oPerson.name;
}

function setName(value) {
  this.oPerson.name = value;
}
```

Function `afterUpdate`

Syntax: `afterUpdate()`

Implement this function to execute JavaScript code after all properties of the extension component have been updated.

Function `componentDeleted`

Syntax: `componentDeleted()`

Implement this function to execute JavaScript code after your extension component has been deleted from the analysis application (for cleanup operations, for example). Next, the extension component's root `<div>` element and its children are removed from the HTML DOM.

Function `Function sap.zen.createStaticSdkMimeUrl Syntax:`

`sap.zen.createStaticSdkMimeUrl(sExtensionId, sMimePath)`

This function returns a URL to a MIME resource, for example an image or a CSS file, that is contained in the SDK extension.

The argument `sExtensionId` is the extension ID of your SDK component. The argument `sMimePath` is the path to the MIME resource relative to the root folder of your SDK extension.

In the example below, when this code is added to the `init` function of the *Colored Box*, an icon appears within this component:

```
var url = sap.zen.createStaticSdkMimeUrl("com.sap.sample.coloredbox", "res/
icon.png");
this.$().append($("<img src=\""+ url + "\"></img>"));
```

Function `callZTLFunction`

Syntax: `callZTLFunction(sMethodname, function, arg1, arg2, ...)`

Call this function to execute a method of the Lumira Designer Script contribution file `contribution.ztl`.

The argument `sMethodname` is the name of the method.

The argument `function` is a JavaScript function that is executed after the method call and the result of the method call is passed.

The arguments `arg1, arg2, ...` are arguments of the method. Arguments should be strings, JSONs, or arrays.

> **i Note**
>
> You can also call private Lumira Designer Script contribution methods.

> **⚠ Caution**
>
> Do not modify data sources during a call of `callZTLFunction`, for example by calling `setFilter`. This adds an error message to the Lumira Designer error log.

In the example below, the private Lumira Designer Script contribution method `getDimension` is called (without arguments). The result is passed to a component setter.

Example:

(`contribution.ztl`)

> **‹≡› Sample Code**

```
@Visibility(private)
String getDimensions() {*
  //...
  return ...;
*}
```

(`contribution.js`)

> **‹≡› Sample Code**

```
//...
that.callZTLFunction("getDimensions", function(result) {
  that.setItems(result);
});
```

For more information on how to optimize your application's roundtrip performance using this function see Roundtrip Optimization [page 59].

Function `callZTLFunctionNoUndo`

Syntax: `callZTLFunctionNoUndo(sMethodname, function, arg1, arg2, ...)`

This is similar to the `callZTLFunction` function but it doesn't record the resulting state changes made by the application's undo stack.

Function `showMessage`

Syntax: `showMessage(sText, bSuppressible)`

Implement this function to replace the standard error notification of the SDK framework (see also hideMessage [page 54]).

The argument `sText` contains the error message provided by the SDK framework.

The optional argument `bSuppressible` indicates to your implementation of the error notification whether the user is able to close the error notification.

The function must return a non-`null` value to replace the standard error notification with your own implementation. It is up to your implementation what value to return as long as it is not `null`. The return value of this function is later passed to the `hideMessage` function. This can be used to identify corresponding `showMessage` and `hideMessage` function calls.

When the SDK framework detects error situations, especially in combination with data-bound properties of an SDK component, it renders a standard error notification on top of your SDK component. You can replace this error notification by implementing your own visualization of the error visualization: Implement the function `showMessage` to display the error visualization and the function `hideMessage` to remove the error visualization.

Function `hideMessage`

Syntax: `hideMessage(value)`

Implement this function to replace the standard error notification of the SDK framework (see also function showMessage [page 54]).

The argument value is the return value of the previous `showMessage` function call. This can be used to identify corresponding `showMessage` and `hideMessage` function calls.

When the SDK framework detects error situations, especially in combination with data-bound properties of an SDK component, it renders a standard error notification on top of your SDK component. You can replace this error notification by implementing your own visualization of the error visualization: Implement the function `showMessage` to display the error visualization and the function `hideMessage` to remove the error visualization.

4.2.2.2 Events

The following functions trigger execution of JavaScript code (events):

Function `firePropertiesChanged`

Syntax: `firePropertiesChanged([sPropertyname1, sPropertyname2, ...])`

Call this function to inform the SDK framework when one or more properties of your extension component have changed in the browser.

> ⚠ Caution
>
> Do not confuse the `firePropertiesChanged` function of the Component JavaScript with the `firePropertiesChanged` function of the Additional Properties Sheet JavaScript.

Example

```
this.firePropertiesChanged(["color"]);
```

This performs the following steps in detail:

1. The runtime is informed that the property `color` (maintained by the runtime) needs to be updated with the new property value now available in the Component JavaScript.
2. The runtime retrieves the new property value by calling the `color()` getter function of the Component JavaScript.
3. The runtime stores this property value in the property `color`.

> ℹ Note
>
> Calling `firePropertiesChanged` triggers a server roundtrip. Therefore, frequent use of this function may decrease the performance of your analysis application. We recommend that this function should only be called upon user interaction. We do not recommend calling this function to implement implicit changes to properties (so-called event cascading), as this may lead to a large number of (or even infinite) server roundtrips. Lumira Designer's standard components only trigger server roundtrips upon user interaction. This ensures efficient use of server roundtrips, which leads to better performance and avoids the threat of indeterministic (or even infinite) server roundtrips through event cascading.

For more information on how to optimize your application's roundtrip performance using this function see Roundtrip Optimization [page 59].

Function `fireEvent`

Syntax: `fireEvent(sPropertyname)`

Call this function to execute the Lumira Designer script that is stored in a property of type `ScriptText` of this extension component.

Example: (File `component.js`)

```
this.fireEvent("onclick");
```

> **i** Note
>
> Calling `fireEvent` triggers a server roundtrip. Therefore, frequent use of this function may decrease the performance of your analysis application. We recommend that this function should only be called upon user interaction. We do not recommend calling this function to implement implicit changes to properties (so-called event cascading), as this may lead to a large number of (or even infinite) server roundtrips. Lumira Designer's standard components only trigger server roundtrips upon user interaction. This ensures efficient use of server roundtrips, which leads to better performance and avoids the threat of indeterministic (or even infinite) server roundtrips through event cascading.

> **→** Tip
>
> Using Default Values of Properties of Type `ScriptText`
>
> With the `fireEvent` function, you can execute a Lumira Designer script that was assigned to a property of type `ScriptText` at design time. However, you can also assign a default value to this property as a string that contains the Lumira Designer script.
>
> If you set the visibility of this property to `false`, then you can use the keyword `this` in this string to refer to the "current" component to which this Lumira Designer script is applied.
>
> **Example:**
>
> The Contribution XML file `contribution.xml` of your component contains a property `onclick` with a default value of `this.doSomething()`:
>
> ```
> <sdkExtension ...>
>   <component ...
>     <jsInclude>res/js/component.js</jsInclude>
>     <property
>       id="onclick"
>       type="ScriptText"
>       visible="false" .../>
>     <initialization>
>       <defaultValue property="onclick">this.doSomething();</defaultValue>
>     </initialization>
>   </component>
> </sdkExtension>
> ```
>
> The Script Contribution file `contribution.ztl` of your component contains the method `doSomething`. The method is marked as private and does not show up in the content assistance of the Lumira Designer script editor:
>
> ```
> @Visibility(private)
> void doSomething() {*
>     // ...
> *}
> ```

Whenever you fire an event on the property `onclick` in one of the functions of the Component JavaScript file `component.js` with

```
fireEvent("onclick");
```

then this will execute the Lumira Designer script stored in the property `onclick`. This is the default value `this.doSomething();`. This in turn executes the `doSomething()` Lumira Designer script method of your component.

→ Tip

Using the Implicit Property `onBeforeRender`

SDK components have an implicit property `onBeforeRender` of type `ScriptText`. The Lumira Designer script assigned to this property is always executed before the SDK component is rendered in the browser. This makes this property an ideal place for initialization code.

The property `onBeforeRender` is not editable in Lumira Designer. However you can assign a default value, a string containing a Lumira Designer script, to this property in the SDK component's Contribution XML.

Example:

In the following example, the Contribution XML file `contribution.xml` of your component defines a property `myDimension`, as well as the property `onclick` of type `ScriptText`. Both properties are not visible in the *Properties* view of Lumira Designer. The default value of the `onclick` property is `this.myHandleClick();`. The default value of the `onBeforeRender` property is `this.myOnBeforeRender();`:

```
<sdkExtension ...>
  <component ...>
  ...
    <property
      id="myDimension"
      type="String"
      visible="false" .../>
    <property
      id="onclick"
      type="ScriptText"
      visible="false" .../>
    <initialization>
      <defaultValue property="onBeforeRender">this.myOnBeforeRender();</
defaultValue>
      <defaultValue property="onClick">this.myHandleClick();</defaultValue>
    </initialization>
  </component>
</sdkExtension>
```

The Script Contribution file `contribution.ztl` of your component contains the Lumira Designer script methods `myOnBeforeRender` and `myHandleClick`. Both methods are marked as private and do not appear in the content assistance of the Lumira Designer script editor:

```
@Visibility(private)
void myOnBeforeRender() {*
  this.myDimension = this.getDataSource().getDimensions()[0].name;
*}
@Visibility(private)
void myHandleClick() {*
  this.getDataSource().setFilter(this.myDimension, ...);
```

```
*}
```

Every time your SDK component is rendered, the Lumira Designer script method `myOnBeforeRender` is executed beforehand. This method retrieves the dimensions of the data source of the component, picks the name of the first dimension, and stores it in property `myDimension`. Now, whenever you fire an event in the Component JavaScript of your SDK component by calling `fireEvent("onclick");`, the Lumira Designer script stored as the default value of the `onlick` property is executed: `this.myHandleClick();`. This script sets a filter on the dimension that was retrieved before the rendering of your SDK component was started.

→ Tip

You can retrieve the data source alias (the Lumira Designer script `DataSourceAlias` object) of your data-bound SDK component with `<componentname>.getDataSource()`.

For more information on how to optimize your application's roundtrip performance using this function see Roundtrip Optimization [page 59].

Function `firePropertiesChangedAndEvent`

Syntax: `firePropertiesChangedAndEvent([sPropertyname1, sPropertyname2, ...], sPropertyname);`

This function is equivalent to

```
firePropertiesChanged([sPropertyname1, sPropertyname2, ...]);
fireEvent(sPropertyname);
```

Function `firePropertiesChangedAndEvent` a faster implementation of this frequent combination of function calls and requires only one server round-trip.

ℹ Note

Calling `firePropertiesChangedAndEvent` triggers a server roundtrip. Therefore, frequent use of this function may decrease the performance of your analysis application. We recommend that this function should only be called upon user interaction. We do not recommend calling this function to implement implicit changes to properties (so-called event cascading), as this may lead to a large number of (or even infinite) server roundtrips. Lumira Designer's standard components only trigger server roundtrips upon user interaction. This ensures efficient use of server roundtrips, which leads to better performance and avoids the threat of indeterministic (or even infinite) server roundtrips through event cascading.

For more information on how to optimize your application's roundtrip performance using this function see Roundtrip Optimization [page 59].

4.2.2.3 Roundtrip Optimization

To optimize roundtrip performance, the number of server roundtrips has reduced by a queuing mechanism.

In previous releases of Lumira Designer, calling one of the following SDK JavaScript functions executed a server roundtrip immediately:

- `firePropertiesChanged`
- `fireEvent`
- `firePropertiesChangedAndEvent`
- `callZTLFunction`

However, too many server roundtrips reduce the performance of the application. The queuing mechanism that optimizes roundtrip performance works by queuing (holding back) changes caused by calls of the above functions in the browser. These changes are sent with a roundtrip to the server only when necessary to keep the application's state on the server consistent.

Roundtrip optimization affects the roundtrip behavior of the SDK JavaScript functions as follows:

Function	Roundtrip Behavior
`firePropertiesChanged`	Queued
`fireEvent`	Immediate
`firePropertiesChangedAndEvent`	Immediate
`callZTLFunction`	Immediate

> **i Note**
>
> For best performance, use function `firePropertiesChanged`.
>
> To continue to force a roundtrip on a property change, use function `firePropertiesChangedAndEvent` instead of function `firePropertiesChanged`. You can pass a dummy text for the required event name to function `firePropertiesChangedAndEvent`.
>
> Roundtrip optimization can be configured by administrators and application designers. By default it is activated.
>
> For more information, see the following chapters on SAP Help Portal at http://help.sap.com:
>
> - “Configuring Roundtrip Optimization for Analysis Applications on the BI Platform” and “Configuring Roundtrip Optimization for Analysis Applications in Lumira Designer” in the *Administrator Guide: SAP Lumira*
> - “Using Roundtrip Optimization” in the *Application Designer Guide: Designing Analysis Applications*

4.3 Script Contributions

In analysis applications and in the Lumira Designer script editor, you can access the properties of an extension component with Lumira Designer scripts by adding a Script Contribution file `contribution.ztl` to the same folder as the Contribution XML.

- The content of `contribution.ztl` is a mix of **Java syntax** (script method signatures) and **JavaScript syntax** (script method bodies).

  > → Tip
  >
  > To open this file in Eclipse with the Java Editor, right-click on `contribution.ztl`, and choose *Open with* > *Other*. In the *Editor Selection* dialog box choose *Java Editor*.

- The JavaScript parts (script method bodies) are executed in the Lumira Designer script engine **on the server** and not in the browser. This means you are restricted to "sand-boxed" JavaScript, without access to the HTML DOM.
- Enclose **script method bodies** in `{**}` pairs.
- Enclose **method blocks** within script body methods in regular braces (`{}`).
- Access **properties** defined in the Contribution XML file with the notation `this.<propertyName>`.
- The following **types** are available:
  - `String`
  - `int`
  - `float`
  - `boolean`
  - `Array`
  - `Object`

  > ⚠ Caution
  >
  > When you change a property of type `Array` or `Object` in a Lumira Designer script method, you need to explicity assign the changed property value to the property after the change.
  >
  > In the following example, a property of type `Array` is defined in the Contribution XML:
  >
  > ‘≡, Sample Code
  >
  > ```
  > <property id="cities" type="Array" title="Cities">
  >   <property id="city" type="String" title="City" />
  > </property>
  > ```
  >
  > The following Lumira Designer script method adds a new city name to the property `cities`. Note how the changed value `current` is assigned to the property `cities` at the end of the method body.
  >
  > ```
  > void add(String name) {*
  >   var current = this.cities || [];
  >   current.push(name);
  >   this.cities = current;
  > *}
  > ```

- **Comments** are automatically included in content assistance and tooltips of the Lumira Designer script editor.
- By **extending your SDK component class** in the Lumira Designer contribution file with `extends Component` your SDK component automatically inherits Lumira Designer script methods that are common to all SDK components, for example:

```
void    setWidth(int width)
int     getWidth()
void    setHeight(int height)
int     getHeight()
void    setBottomMargin(int bottomMargin)
int     getBottomMargin()
void    setTopMargin(int topMargin)
int     getTopMargin()
void    setLeftMargin(int leftmargin)
int     getLeftMargin()
void    setRightMargin(int rightMargin)
int     getRightMargin()
void    setCSSClass(String className)
String  getCSSClass()
void    setVisible(boolean isVisible)
boolean isVisible()
void    showLoadingState()
void    hideLoadingState()
```

> → Tip
>
> By extending your class with `extends DataBoundComponent` your SDK component automatically aditionally inherits the following Lumira Designer script methods:
>
> ```
> DataSourceAlias getDataSource()
> void            setDataSource(DataSourceAlias dataSourceAlias);
> ```

- The Script contribution file can contain script contributions of **multiple extension components**.

The example below is the Script Contribution file of the extension component **Colored Box**.

Example: (File `contribution.ztl`)

```
class com.sap.sample.coloredbox.ColoredBox extends Component {

  /* Returns the current color of the box */
  String getColor() {*
    return this.color;
  *}
  /* Sets the current color of the box */
  void setColor(/* the  new color */ String newColor) {*
    this.color = newColor;
  *}
}
```

> → Tip
>
> No Script Contibution file vs. Script Contribution file without methods
>
> Although excluding the Script Contribution file completely hides your SDK component in the content assistance of the Lumira Designer Script editor, you may find it useful to provide a Script Contribution file without any methods. In this case, the SDK extension component automatically inherits Lumira Designer script methods that are common to all SDK extension components, for example `setWidth()`, `getWidth()`, etc.

This example shows the empty Script Contribution file of the extension component `Colored Box`:

```
class com.sap.sample.coloredbox.ColoredBox extends Component {
}
```

4.4 Additional Properties Sheet

In Lumira Designer you can provide an extension component with an interactive Additional Properties Sheet, which allows users to set and get extension component property values. The Additional Properties Sheet of the extension component is displayed in Lumira Designer's *Additional Properties* view.

An Additional Properties Sheet consists of:

- an HTML file to specify the visual appearance
- a JavaScript file to implement the functional behavior

> **i Note**
>
> Once your SDK extension has been deployed to BI platform, all HTML and JavaScript files that you package with your SDK extension are available on the BI Platform at runtime. It might happen that application users accidentally open an Additional Properties Sheet HTML page in their Web browser. It would not work, but might be confusing. In addition, such files could be a security risk.
>
> Therefore we recommend that you provide two extension packages:
>
> - one package with Additional Properties Sheet content that you install on Lumira Designer only
> - one package without that content that you deploy to BI platform

4.4.1 HTML

The HTML file specifies the visual appearance of the Additional Properties Sheet.

1. Place the Additional Properties Sheet HTML file in SDK extension's `res` folder or subfolder.
2. Reference the Additional Properties Sheet HTML file in the `propertySheetPath` attribute of the `<component>` element in the Contribution XML file.

The example below is the Additional Properties Sheet HTML file of the extension component **Colored Box**. It defines the visual appearance using `<form>` and `<fieldset>` elements. It also uses an `<input>` element - an input field that allows users to enter a color value.

> **i Note**
>
> - Here two JavaScript files are referenced: the generic Additional Properties Sheet JavaScript file of the SDK framework and the JavaScript file of this Additional Properties Sheet (see JavaScript [page 63]).
> - The Additional Properties Sheet JavaScript class is instantiated here (`new com.sap.sample.coloredbox.ColoredBoxPropertyPage();`)

.

Example

(File `additional_properties_sheet.html`)

```
<html>
  <head>
    <title>Colored Box Property Sheet</title>
    <meta http-equiv="Content-Type" content="text/html;charset=utf-8" />
    <script src="/aad/zen.rt.components.sdk/resources/js/
sdk_propertysheets_handler.js"></script>
    <script src="additional_properties_sheet.js"></script>
  </head>
  <script>
     new com.sap.sample.coloredbox.ColoredBoxPropertyPage();
  </script>
  <body>
    <form id="form">
      <fieldset>
        <legend>Colored Box Properties</legend>
        <table>
          <tr>
            <td>Color</td>
            <td><input id="aps_color" type="text" name="color" size="40"
maxlength="40"></td>
          </tr>
        </table>
      </fieldset>
    </form>
  </body>
</html>
```

4.4.2 JavaScript

For each Additional Properties Sheet HTML file, you can implement a complementing Additional Properties Sheet JavaScript class to make the extension component's Additional Properties Sheet interactive. You can implement this JavaScript class using both JavaScript and jQuery, as jQuery is included in the Design Studio SDK framework. The Lumira Component SDK includes jQuery 2.2.3.

1. Place the Additional Properties Sheet JavaScript file in the SDK extension's `res` folder or subfolder.
2. Reference this file in the complementing Additional Properties Sheet HTML file (see HTML [page 62]).

The example below is the Additional Properties Sheet JavaScript class of the extension component **Colored Box**. Its class name is `com.sap.sample.coloredbox.ColoredBoxPropertyPage` and subclasses the generic JavaScript class `sap.designstudio.sdk.PropertyPage`. It implements an `init` function, which attaches an event handler for the submit event to the `<form>` element with ID `form`. When the coloured box has been clicked, the SDK framework is notified that the extension component's `color` property has changed in the browser. Furthermore, the JavaScript defines a `color` function, which acts as a combined setter and getter function for the input field with ID `aps_color`. This enables the SDK framework to get and set the value of the input entered in the Additional Properties Sheet HTML.

Example

(File `additional_properties_sheet.js`)

```
sap.designstudio.sdk.PropertyPage.subclass("com.sap.sample.coloredbox.ColoredBoxP
ropertyPage", function() {
  var that = this;
  this.init = function() {
    $("#form").submit(function() {
      that.firePropertiesChanged(["color"]);
      return false;
    });
  };
  this.color = function(value) {
    if (value === undefined) {
      return $("#aps_color").val();
    } else {
      $("#aps_color").val(value);
      return this;
    }
  };
});
```

Related Information

JavaScript Functions for the Additional Properties Sheet [page 65]
Getting and Setting Extension Component Properties [page 69]

4.4.2.1 Additional Properties Sheet Lifecycle

When the Additional Properties Sheet is rendered, the SDK framework executes the following sequence of Additional Properties Sheet JavaScript function calls:

- `init()`

  > **i Note**
  >
  > This function is called only once when the Additional Properties Sheet is rendered for the first time.

- `beforeUpdate()`
- Update all extension component properties using their setter/getter functions (see Getting and Setting Extension Component Properties [page 69])

  > **i Note**
  >
  > First, all the getter functions are called, then the setter functions (of properties that have changed).

> ⚠ Caution
>
> In a getter or setter function, do not call Additional Properties Sheet JavaScript function `firePropertiesChanged`. This can lead to infinite invocations of getter or setter functions (so-called "event cascading") and can bring your application to a halt.

- `afterUpdate()`

4.4.2.2 JavaScript Functions for the Additional Properties Sheet

Function `init`

Syntax: `init()`

Implement this function to execute JavaScript code after the Additional Properties Sheet HMTML page is associated with the extension component.

Function `beforeUpdate`

Syntax: `beforeUpdate()`

Implement this function to execute JavaScript code before the extension component properties are updated from the Additional Properties Sheet.

> ⚠ Caution
>
> Do not confuse the `beforeUpdate` function of the Additional Properties Sheet JavaScript with the `beforeUpdate` function of the Component JavaScript!

Function `afterUpdate`

Syntax: `afterUpdate()`

Implement this function to execute JavaScript code after the extension component properties have been updated from the Additional Properties Sheet.

> ⚠ Caution
>
> Do not confuse the `afterUpdate` function of the Additional Properties Sheet JavaScript with the `afterUpdate` function of the Component JavaScript!

Function `firePropertiesChanged`

Syntax: `firePropertiesChanged([sPropertyname1, sPropertyname2, ...])`

Call this function to inform the SDK framework when one or more properties of the extension component have changed in the Addtional Properties Sheet.

> ⚠ Caution
>
> Do not confuse the `firePropertiesChanged` function of the Additional Properties Sheet JavaScript with the `firePropertiesChanged` function of the Component JavaScript.

Example

```
this.firePropertiesChanged(["color"]);
```

This performs the following steps in detail:

1. The Runtime is informed that the property `color` (maintained by the Runtime) needs to be updated with the new property value now available in the Additional Properties Sheet JavaScript.
2. The Runtime retrieves the new property value by calling the `color()` getter function of the Additional Properties Sheet JavaScript.
3. The Runtime stores this property value in the property `color`.
4. The Runtime updates the extension component in the browser by calling the `color()` setter function of the Component JavaScript and passing the new property value of property `color`.

> ℹ Note
>
> Calling `firePropertiesChanged` triggers a server roundtrip. Therefore, frequent use of this function may decrease the performance of your analysis application. We recommend that this function should only be called upon user interaction. We do not recommend calling this function to implement implicit changes to properties (so-called event cascading), as this may lead to a large number of (or even infinite) server roundtrips. Lumira Designer's standard components only trigger server roundtrips upon user interaction. This ensures efficient use of server roundtrips, which leads to better performance and avoids the threat of indeterministic (or even infinite) server roundtrips through event cascading.

Function `callRuntimeHandler`

Syntax: `callRuntimeHandler(sFunctionname, sArgument1, sArgument2, ...)`

Call this function to execute a JavaScript function located in the Component JavaScript file. The argument `functionName` is a string with the name of the JavaScript function to be called. The optional arguments `sArgument1`, `sArgument2`, etc. are passed to the JavaScript function.

Example

(File `additional_properties_sheet.js`)

```
this.callRuntimeHandler("getMetadataAsString");
```

Example

You can pass arguments to the JavaScript function located in the Component JavaScript file by adding them to the call of function `callRuntimeHandler()`.

```
this.callRuntimeHandler("sampleFunction", "arg1", "arg2");
```

Function `componentSelected`

Syntax: `componentSelected()`

Implement this function to execute JavaScript code when the extension component has been selected in Lumira Designer.

Function `openPropertyDialog`

Syntax: `openPropertyDialog(sPropertyname)`

Call this function to open a property dialog box (value help) to select a property value. Property dialog boxes are supported for properties of the following types:

- `Color`
- `ScriptText`
- `ResultCell`
- `ResultCellList`
- `ResultCellSet`
- `ResultSet`

Example

```
this.openPropertyDialog("color");
```

Function `callZTLFunction`

Syntax: `callZTLFunction(sMethodname, function, arg1, arg2, ...)`

Call this function to execute a method of the Lumira Designer Script contribution file `contribution.ztl`.

The argument `sMethodname` is the name of the method.

The argument `function` is a JavaScript function that is executed after the method call and the result of the method call is passed.

The arguments `arg1, arg2, ...` are arguments of the method. Arguments should be strings, JSONs, or arrays.

> **i Note**
>
> You can also call private Lumira Designer Script contribution methods.

> **⚠ Caution**
>
> Do not modify data sources during a call of `callZTLFunction`, for example by calling `setFilter`. This adds an error message to the Lumira Designer error log.

In the example below, the private Lumira Designer Script contribution method `getDimension` is called (without arguments). The result is passed to a component setter.

Example:

(`contribution.ztl`)

> **Sample Code**
>
> ```
> @Visibility(private)
> String getDimensions() {*
>   //...
>   return ...;
> *}
> ```

> **Sample Code**
>
> ```
> //...
> that.callZTLFunction("getDimensions", function(result) {
>   that.setItems(result);
> });
> ```

Function `log`

Syntax: `log(sMessage, sSeverity)`

Call this function to add a log message to the Lumira Designer *Error Log* view.

The argument `sMessage` is the log message.

The argument `sSeverity` indicates the severity of the message. Supported values are: `"info"`, `"warn"`, `"error"`, and `"log"`.

> **i Note**
>
> Whether the log message is actually displayed in the Lumira Designer *Error Log* view, depends on the configured log level of Lumira Designer. To display the configured log level, choose *Tools* > *Preferences* > *Application Design* > *Support Settings* and locate the dropdown box *Log Level*.

Example:

> **‹≡› Sample Code**
>
> ```
> this.log("Variable is undefined", "error");
> ```

4.4.2.3 Getting and Setting Extension Component Properties

For each extension component property, you can implement a function that acts as a combined setter and getter function.

- The **function name** is the property's name.
- The function's **setter clause** must return `this` to allow function calls to be chained, thus creating a fluent interface.

Example

(File `additional_properties_sheet.js`)

Note the jQuery notation `$("#aps_color")` to access the `<input>` element.

```
this.color = function(value) {
  if (value === undefined) {
    return $("#aps_color").val();
  } else {
    $("#aps_color").val(value);
    return this;
  }
};
```

Properties of type `Array` can be accessed like normal JavaScript arrays.

Properties of type `Object` can be accessed like normal JavaScript objects in JSON notation.

> **⚠ Caution**
>
> In a getter or setter function, do not call Additional Properties Sheet JavaScript function `firePropertiesChanged`. This can lead to infinite invocations of getter or setter functions (so-called "event cascading") and can bring your application to a halt.

4.5 Exporting an SDK Extension Component

You can enable your extension component for export by the PDF export framework.

Extension components enabled for export can be selected in the *Edit Report Selection* dialog box of the *Export* component.

To enable your extension component for export, add the following attribute to your extension component's `<component>` element in the Contribution XML:

```
supportsExportContent="true"
```

Example

The following example shows how to enable export for the sample Component SDK extension `Colored Box`. The attribute `supportsExportContent` was added in the Contribution XML to the `<component>` element of the `Colored Box`:

(File contribution.xml)

Sample Code

```
<component  id="ColoredBox"
  // ...
  supportsExportContent="true">
  //...
</component>
```

5 SDK Extensions and Data Binding

You can create SDK extensions with extension components that retrieve and display data from the result set of a data source on an SAP BW or SAP HANA system (data binding).

SDK extension components can also retrieve data from result sets of an SDK data source.

5.1 Prerequisites

To enable data binding between an extension component and a data source, add the following attribute to the extension component's `<component>` element in the Contribution XML:

```
databound="true"
```

> **i** Note
>
> - This automatically adds the `Data Source` property to the extension component. It is displayed in the *Properties* view of the extension component in SAP Lumira Designer.
> - This automatically adds the `metadata` property to the extension component.

Related Information

Runtime Property Values [page 73]

5.2 Result Set Terminology

To simplify discussion about data binding, here is a quick review of result set terminology.

The result set of a data source is a two-dimensional table with a **column axis** and a **row axis**.

- Each axis has a list of **dimensions**.
- One dimension can contain **measures**.
- Each dimension has **dimension members** (or simply "**members**").
- The dimension members on an axis form an **axis tuple** at each axis position.
- The intersection of each row and column contains a **data value**.

Example

The table below has two column dimensions (**DATE** and **Measures**) and one row dimension (**CITY**). The dimension **CITY** has the members Berlin, Rio de Janeiro, Tokyo, and Overall Result. The dimension **DATE** has the members 2010-01-01 and 2012-01-01. The dimension **Measures** has the members SALESREVENUE and QUANTITYSOLD. The column axis tuple [2010-01-01, SALESREVENUE] specifies the first column of the result set.

	2010-01-01		2012-01-01	
	SALESREVENUE	QUANTITYSOLD	SALESREVENUE	QUANTITYSOLD
Berlin	190,958.00	1,479	393,902.00	2,721
Rio de Janeiro	139,410.00	1,104	259,345.00	1,752
Tokyo	194,392.00	1,471	412,279.00	2,700
Overall Result	**524,760.00**	**4,054**	**1,065,526.00**	**7,173**

5.3 Data-Bound Properties

Several types of data-bound properties allow you to restrict the selection of data values from a result set. Data-bound property types also help the SDK framework to check the feasability of your selection and restrict the available selections in the *Select Data* dialog box (value help of data-bound properties in Lumira Designer).

The following data-bound property types are available:

Property Type	Data Values
`ResultCell`	A single data value
`ResultCellList`	A single row or column of data values
`ResultCellSet`	A complex selection of data values from rows and columns (a Cartesian selection)
`ResultSet`	All data values of the result set

> **i** Note
>
> A Cartesian selection contains data points in the multidimensional cube that form a connected space.

5.3.1 Design Time Property Values

At design time, you assign a **selection string** to a data-bound property. It specifies which data values of the result set are received by the property. It is expressed in JSON notation and is called the **Design time JSON**.

Example

Selection string to select the cell containing the quantity sold in 2010 in Tokyo, in the result set example under Data-Bound Properties [page 72] (used with a `ResultCell` property):

```
{"DATE": "2010-01-01", "(MEASURES_DIMENSION)": "QUANTITYSOLD", "CITY": "Tokyo"}
```

Example

Selection string to select the second column of the result set, in the result set example under Data-Bound Properties [page 72] (used with a `ResultList` property):

```
{"DATE": "2010-01-01", "(MEASURES_DIMENSION)": "QUANTITYSOLD"}
```

The design time JSON contains a list of dimension-member pairs for selecting a subset of the result set. The dimension-member pairs can be in any order. If a dimension is omitted, then all its members, including aggregate members, are selected.

> **i** Note
>
> Use `"(RESULT_MEMBER)"` as **member** to select the aggregate member of a dimension.

> **i** Note
>
> Use `"(MEASURES_DIMENSION)"` as **dimension** to select a measure structure. It is converted internally into the correct name of the measure structure.

5.3.2 Runtime Property Values

At runtime, the SDK framework retrieves the selected data values and stores them in the data-bound property in JSON format called **Data Runtime JSON**. The Data Runtime JSON contains mostly data value information. To complement this information, the SDK framework automatically creates a `metadata` property and assigns the metadata of the data values to it in a JSON format called **Metadata Runtime JSON** to this property. The Metadata Runtime JSON contains additional, helpful information about the data values. Data-bound extension components can examine the values of both the Metadata Runtime JSON and Data Runtime JSONs in order to create appropriate output.

> **i** Note
>
> For a data-bound property of type `ResultSet`, the Metadata Runtime JSON content is part of the Data Runtime JSON. There is no implicit `metadata` property and thus no separate Metadata Runtime JSON available.

> **i Note**
>
> If an extension component contains multiple data-bound properties, the `metadata` property contains a merged version of the Metadata Runtime JSONs of all data-bound properties.

Metadata Runtime JSON

Here you see a formal representation of the Metadata Runtime JSON:

```
{
  "dimensions": [
    {
       "key": <string>,
       "text": <string>,
       "axis": "COLUMNS"|"ROWS",
       "axis_index": <integer>,
       "containsMeasures": true|false,
       "attributes": [
         {
           "key": <string>,
           "text": <string>
         }, ...
       ],
      "hierarchy" : {
          "key": <string>,
         "text": <string>,
         "nodeAlignment":
"top"|"bottom"
      },
       "members": [
        {
           "key": <string>,
           "text": <string>,
           "formatString": <string>,
           "unitOfMeasure": <string>,
           "scalingFactor": <integer>,
           "type":  "RESULT",
           "nodeState":
"COLLAPSED"|"EXPANDED",
           "level": <integer>
           "attributeMembers": [
             {
               "key": <string>,
               "text": <string>
             } | null, ...
           ]
          "parent": <string>
         "unixTimeMillis": <integer>
         "timeSpanMillis": <integer>
         "isExcluded": true|false
         "valueType":
<string>
        }, ...
       ],
     }, ...
  ],
  "externalDimensions": [
     ...
  ],
  "conditionalFormats": [
    {
      "key": <string>,
      "text": <string>,
      <customObject>
    }, ...
  ],
  "locale": <string>
}
```

Array of dimensions; column dimensions first, then row dimensions

Dimension key

Dimension text

Axis on which the dimension is located

Axis tuple index of the dimension, > = 0

Does dimension contain measures? (omitted when false)

Array of attributes (omitted when result set has no attributes)

Attribute key

Attribute text

Hierarchy info (omitted when hierarchy inactive or not assigned)

Hierarchy key

Hierarchy text

Where child nodes are placed (bottom: below parent node, top: above parent node)

Array of dimension's members

Member key

Member text (text may contain "|" separator depending on used presentation)

Format string, in Java DecimalFormat format (only with measures members)

Unit of measure string (only with measures members)

Scaling factor as exponent to base 10 (omitted when 0, only with measures members)

Member is an aggregate value (omitted when not)

Node state (only with hierarchy members)

Indent level, > 0 (only with hierarchy members)

Array of attribute members (omitted when result set has no attributes)

Attribute member key

Attribute member text

	Attribute member is null for members of type "RESULT"
	Key of parent member (omitted when hierarchy inactive, not assigned, or member has no parent)
	Timestamp of member in milliseconds since January 1, 1970 (only with members representing a date)
	Time span of member in milliseconds (only with members representing a time span)
	Member is not part of the result set (omitted when false, only with measures members)
	Type of member
	Array of external dimensions
	(optional) Array element has same structure as an array element of JSON property dimensions. Currently only one element is supported.
	Array of conditional formats
	Key of conditional format
	Text of conditional format
	(optional) Custom object containing custom parameters
	Browser locale string

> **i** Note
>
> The `dimension` array contains the column dimensions first (sorted by increasing `axis_index`), then the row dimensions (sorted by increasing `axis_index`).

JSON property `externalDimensions`

If the result set contains a measures dimension, but this dimension is not contained in the row or the column dimensions of the `dimensions` JSON property, then this measures dimension is stored in the `externalDimensions` JSON property. It is an array, which can contain the measures dimension as its only element.

A result set can contain one measure dimension. It can be a dimension of the `dimensions` JSON property or the `externalDimension` JSON property. A measure has the `isMeasureDimension` JSON property set to `true`.

JSON property `attributes`

Contains information (key and text) about each attribute of a dimension. It is only present if the result set actually contains attributes and the data-bound property option `includeAttributes` is `true`.

JSON property `attributeMembers`

Contains information (key and text) about each attribute member of a dimension attribute. Is is only present if the result set actually contains attributes and the data-bound property option `includeAttributes` is `true`.

JSON property `conditionalFormats`

Contains information about the conditional formats that have been applied to the result set. It is only present if the result set actually contains conditional formats and the data-bound property option `includeConditionalFormats` is `true`.

JSON property `unixTimeMillis`

Contains a timestamp of the member in milliseconds since January 1, 1970 00:00:00 UTC. It is only present if the member represents a date.

JSON property `timeSpanMillis`

Contains a time span of the member in milliseconds. It is only present if the member represents a time span.

JSON property `isExcluded`

`True` if the member (it is a measure member) is not part of the result set. Requires the data-bound property's option `includeAllDimensionsAndMeasures` (see Elements of the Contribution XML File [page 21]) to be `true`.

JSON property `valueType`

Indicates the type of the value, for example, `DOUBLE, PERCENT, CALENDAR_DAY`, and so on.

Data Runtime JSON

Here you see a formal representation of the Data Runtime JSON:

```
{
  "selection": [<integer>, ...],
   "data": [<float>|null, ...]
   "formattedData": [<String>, ...]
   "tuples": [[<integer>, ...], ...
   "axis_columns":
[[<integer>, ...], ...],
   "axis_rows":
[[<integer>, ...], ...],
   "conditionalFormatValues": [
     {
       <string>: <integer>,
       ...
     } | null, ...
   ],
   "columnCount": <integer>,
   "rowCount": <integer>
   "dataSourceInfo": {
    "dataSourceComponent": "DS_1",
    "name": "FINANCIALS_QUERY",
    "connection": "AB1",
    "type": "QUERY"
  }

}
```

Array of selection dimension member indexes, index = -1 (unspecified by selection)

Array of data values in left-to-right, first-to-last row order, may contain null value

Array of formatted data values in left-to-right, first-to-last row order

Array of tuple arrays, one tuple for each data value. Tuple element = -1 (tuple element unspecified by selection)

Array of tuple arrays, one tuple per column axis position specifying the column axis tuple elements. Tuple element = -1 (dimension is not on the column axis (only with properties of type ResultSet))

Array of tuple arrays, one tuple per row axis position specifying the row axis tuple elements. Tuple element = -1 (dimension is not on the row axis (only with properties of type ResultSet))

Array of conditional format maps, one map for each data value

Array element is a map of conditional formats

String contains key of conditional format, integer contains alert level (1..9)

Array element is null if data value has no conditional format

Number of columns of the data

Number of rows of the data

Data source information

Name of data source component

Name of data source

Name of connection

Type of query

JSON property `formattedData`

Contains the data as an array of formatted strings in left-to-right, first-to-last row order.

JSON property `columnCount`

Contains the number of columns of the data.

JSON property `rowCount`

Contains the number of rows of the data.

JSON property `conditionalFormatValues`

Contains information about which conditional formats are applied to each data value of the result set. This JSON property contains an array of conditional format maps - one map for each data value of the result set or `null` if a data value has no conditional format. The conditional format maps contain one or more sets of key-value pairs. The key is the key of the conditional format in JSON property `conditionalFormats` in the Metadata Runtime JSON. The value is an integer between 1..9 that indicates the alert level (specifically: the highest alert level of this conditional format applied to this data value).

This JSON property is only present if the result set actually contains conditional formats and the data-bound property option `includeConditionalFormats` is `true`.

JSON property `dataSourceInfo`

Contains information about the data source that provides the data. It contains the JSON properties `dataSourceComponent`, `name`, `connection`, and `type`.

This JSON property is only present if the data-bound property option `includeDataSourceInfo` is `true`.

You can find fully-executed examples in the following chapters.

Related Information

Cell Selection [page 79]
Column or Row Selection [page 81]
Columns and Row Selection (Multiple Columns or Rows) [page 84]
Columns and Row Selection ("Checkerboard") [page 86]
Result Set Selection [page 89]

5.3.3 Cell Selection

To get the data value of a single cell in a result set, use a data-bound property of type `ResultCell`.

For example, you have the following result set:

	2010-01-01		2012-01-01	
	SALESREVENUE	QUANTITYSOLD	SALESREVENUE	QUANTITYSOLD
Berlin	190,958.00	1,479	393,902.00	2,721
Rio de Janeiro	139,410.00	1,104	259,345.00	1,752
Tokyo	194,392.00	**1,471**	412,279.00	2,700
Overall Result	524,760.00	4,054	1,065,526.00	7,173

To select the highlighted cell (with value **1,471**) of this result set, use the following selection string:

```
{"DATE": "2010-01-01", "(MEASURES_DIMENSION)": "QUANTITYSOLD", "CITY": "Tokyo"}
```

The SDK framework returns the following Data Runtime and Metadata Runtime JSONs:

Data Runtime JSON

```
{
  "selection": [0, 0, 0],
   "data": [1471],
   "tuples": [[0, 0, 0]],
   "columnCount": 1,
   "rowCount": 1
}
```

The `selection` JSON property reflects the selection. It contains an array of three indexes, corresponding to the three dimensions of the result set, in the order **DATE**, **Measures**, and **CITY** (see Metadata Runtime JSON below). The index values point at the selected dimension members (see Metadata Runtime JSON below):

- `0 = 2010-01-01` for dimension **DATE**
- `0 = QUANTITYSOLD` for dimension **Measures**
- `0 = Tokyo` for dimension **CITY**

The `data` JSON property contains an array with the single data value of the selected result set cell.

For each data value, the `tuples` JSON property contains a tuple of indexes for the selected dimension members (see Metadata Runtime JSON below):

- `0 = 2010-01-01` for dimension **DATE**
- `0 = QUANTITYSOLD` for dimension **Measures**
- `0 = Tokyo` for dimension **CITY**

The `columnCount` and the `columnCount` JSON properties contain the number of rows and columns of the data, respectively.

Metadata Runtime JSON

```
{
  "dimensions": [
    {
       "key": "DATE",
       "text": "DATE",
       "axis": "COLUMNS",
       "axis_index": 0,
       "members": [
        {
           "key": "2010-01-01",
           "text": "01/01/2010"
        }
      ]
    },
    {
       "key": "Measures",
       "text": "Measures",
       "axis": "COLUMNS",
       "axis_index": 1,
```

```
        "containsMeasures": true,
        "members": [
         {
            "key": "QUANTITYSOLD",
            "text": "QUANTITYSOLD",
            "formatString": "#.##0;'-'#.##0"
         }
       ]
     },
     {
        "key": "CITY",
        "text": "CITY",
        "axis": "ROWS",
        "axis_index": 0,
        "members": [
         {
            "key": "Tokyo",
            "text": "Tokyo"
         }
       ]
     }
   ],
    "locale": "en_US"
}
```

The `dimensions` JSON property contains dimension and member information for each dimension relevant for the selection.

The `locale` JSON property contains the browser's locale string.

5.3.4 Column or Row Selection

To get the data values of a single result set column or row, use a data-bound property of type `ResultCellList`.

For example, you have the following result set:

	2010-01-01		2012-01-01	
	SALESREVENUE	QUANTITYSOLD	SALESREVENUE	QUANTITYSOLD
Berlin	190,958.00	**1,479**	393,902.00	2,721
Rio de Janeiro	139,410.00	**1,104**	259,345.00	1,752
Tokyo	194,392.00	**1,471**	412,279.00	2,700
Overall Result	524,760.00	**4,054**	1,065,526.00	7,173

To select the highlighted column (**QUANTITYSOLD**) of this result set, use the following selection string:

```
{"DATE": "2010-01-01", "(MEASURES_DIMENSION)": "QUANTITYSOLD"}
```

> **i** Note
>
> Row selection works in the same way.

The SDK framework returns the following Data Runtime and Metadata Runtime JSONs:

Data Runtime JSON

```
{
  "selection": [0, 0, -1],
   "data": [
    1479,
    1104,
    1471,
    4054
  ]
   "tuples": [
    [0, 0, 0],
    [0, 0, 1],
    [0, 0, 2],
    [0, 0, 3]
  ],
  "columnCount": 1,
  "rowCount": 4
}
```

The `selection` JSON property reflects the selection. It contains an array of three indexes corresponding to the three dimensions of the result set in the order **DATE**, **Measures**, and **CITY** (see Metadata Runtime JSON below). The index values point at the selected dimension members (see Metadata Runtime JSON below):

- `0 = 2010-01-01` for dimension **DATE**
- `0 = QUANTITYSOLD` for dimension **Measures**
- `-1 =` This dimension was not specified by the selection string.

The `data` JSON property contains an array with the data values of the selected result set column.

For each data value, the `tuples` JSON property contains a tuple of indexes of the selected dimensions members. For example, the first tuple `[0, 0, 0]` points at the following dimension members:

- `0 = 2010-01-01` for dimension **DATE**
- `0 = QUANTITYSOLD` for dimension **Measures**
- `0 = Berlin` for dimension **CITY**

The second tuple `[0, 0, 1]` points at the following dimension members:

- `0 = 2010-01-01` for dimension **DATE**
- `0 = QUANTITYSOLD` for dimension **Measures**
- `1 = Rio de Janeiro` for dimension **CITY** (see Metadata Runtime JSON below).

The `columnCount` and the `columnCount` JSON properties contain the number of rows and columns of the data, respectively.

Metadata Runtime JSON

```
{
  "dimensions": [
```

```
    {
       "key": "DATE",
       "text": "DATE",
       "axis": "COLUMNS",
       "axis_index": 0,
       "members": [
        {
           "key": "2010-01-01",
           "text": "2010-01-01"
        }
      ]
    },
    {
       "key": "Measures",
       "text": "Measures",
       "axis": "COLUMNS",
       "axis_index": 1,
       "containsMeasures": true,
       "members": [
        {
           "key": "QUANTITYSOLD",
           "text": "QUANTITYSOLD",
           "formatString": "#.##0;'-'#.##0"
        }
      ]
    },
    {
       "key": "CITY",
       "text": "CITY",
       "axis": "ROWS",
       "axis_index": 0,
       "members": [
        {
           "key": "Berlin",
           "text": "Berlin"
        },
        {
           "key": "Rio de Janeiro",
           "text": "Rio de Janeiro"
        },
        {
           "key": "Tokyo",
           "text": "Tokyo"
        },
        {
           "key": "Result",
           "text": "Overall Result",
           "type": "RESULT"
        }
      ]
    }
  ],
   "locale": "en_US"
}
```


The `dimensions` JSON property contains dimension and member information for each dimension relevant for the selection.

The `locale` JSON property contains the browser's locale string.

5.3.5 Columns and Row Selection (Multiple Columns or Rows)

To get the data values of multiple columns or rows, use a data-bound property of type `ResultCellSet`.

For example, you have the following result set:

	2010-01-01		2012-01-01	
	SALESREVENUE	QUANTITYSOLD	SALESREVENUE	QUANTITYSOLD
Berlin	190,958.00	**1,479**	393,902.00	**2,721**
Rio de Janeiro	139,410.00	**1,104**	259,345.00	**1,752**
Tokyo	194,392.00	**1,471**	412,279.00	**2,700**
Overall Result	524,760.00	**4,054**	1,065,526.00	**7,173**

To select the highlighted columns (both **QUANTITYSOLD** columns) of this result set, use the following selection string:

```
{"DATE": ["2010-01-01", "2012-01-01"], "(MEASURES_DIMENSION)": "QUANTITYSOLD"}
```

> **i** Note
>
> Multiple row selection works in the same way.

The SDK framework returns the following Data Runtime and Metadata Runtime JSONs:

Data Runtime JSON

```
{
   "selection": [[0, 1], 0, -1],
   "data": [
    1479,
    2721,
    1104,
    1752,
    1471,
    2700,
    4054,
    7173
  ]
   "tuples": [
    [0, 0, 0],
    [1, 0, 0],
    [0, 0, 1],
    [1, 0, 1],
    [0, 0, 2],
    [1, 0, 2],
    [0, 0, 3],
    [1, 0, 3]
  ],
  "columnCount": 2,
  "rowCount": 4
```

```
}
```

The `selection` JSON property `[[0, 1], 0, -1]` reflects the selection. It contains an array of three elements corresponding to the three dimensions of the result set in the order `DATE`, `Measures`, and `CITY` (see Metadata Runtime JSON below). The index values point to the selected dimension members (see Metadata Runtime JSON below):

- `0, 1 = 2010-01-01, 2012-01-01` for dimension `DATE`
- `0 = QUANTITYSOLD` for dimension `Measures`
- `-1 =` This dimenson was not specified by the selection string.

If your code assumes that the elements of the `selection` JSON property are always integer numbers, this may lead to incompatible changes.

The `data` JSON property contains an array with the data values of the selected result set columns in the following order: left-to-right cell, first-to-last row.

For each data value, the `tuples` JSON property contains a tuple of indexes to the selected dimensions members. For example, the first tuple `[0, 0, 0]` points at the following dimension members:

- `0 = 2010-01-01` for dimension **DATE**
- `0 = QUANTITYSOLD` for dimension **Measures**
- `0 = Berlin` for dimension **CITY**

The second tuple `[1, 0, 0]` points at the following dimension members:

- `1 = 2012-01-01` for dimension **DATE**
- `0 = QUANTITYSOLD` for dimension **Measures**
- `0 = Berlin` for dimension **CITY** (see Metadata Runtime JSON below)

The `columnCount` and the `columnCount` JSON properties contain the number of rows and columns of the data, respectively.

Metadata Runtime JSON

```
{
  "dimensions": [
    {
       "key": "DATE",
       "text": "DATE",
       "axis": "COLUMNS",
       "axis_index": 0,
       "members": [
        {
           "key": "2010-01-01",
           "text": "01/01/2010"
        },
        {
           "key": "2012-01-01",
           "text": "01/01/2012"
        }
      ]
    },
    {
       "key": "Measures",
       "text": "Measures",
```

```
        "axis": "COLUMNS",
        "axis_index": 1,
        "containsMeasures": true,
        "members": [
         {
            "key": "QUANTITYSOLD",
            "text": "QUANTITYSOLD",
            "formatString": "#,##0;'-'#,##0"
         }
       ]
     },
     {
        "key": "CITY",
        "text": "CITY",
        "axis": "ROWS",
        "axis_index": 0,
        "members": [
         {
            "key": "Berlin",
            "text": "Berlin"
         },
         {
            "key": "Rio de Janeiro",
            "text": "Rio de Janeiro"
         },
         {
            "key": "Tokyo",
            "text": "Tokyo"
         },
         {
            "key": "Result",
            "text": "Result",
            "type": "RESULT"
         }
       ]
     }
   ],
    "locale": "en_US"
}
```

The `dimensions` JSON property contains dimension and member information for each dimension relevant for the selection.

The `locale` JSON property contains the browser's locale string.

5.3.6 Columns and Row Selection ("Checkerboard")

To get the data values of multiple sub columns and rows (also known as a "checkerboard"), use a data-bound property of type `ResultCellSet`.

For example, you have the following result set:

	2010-01-01		2012-01-01	
	SALESREVENUE	QUANTITYSOLD	SALESREVENUE	QUANTITYSOLD
Berlin	190,958.00	**1,479**	393,902.00	**2,721**
Rio de Janeiro	139,410.00	1,104	259,345.00	1,752

	2010-01-01		2012-01-01	
	SALESREVENUE	QUANTITYSOLD	SALESREVENUE	QUANTITYSOLD
Tokyo	194,392.00	**1,471**	412,279.00	**2,700**
Overall Result	524,760.00	4,054	1,065,526.00	7,173

To select the highlighted sub columns and rows (with cells **1,479** , **1,471** , **2,721** and **2,700**) of this result set, use the following selection string:

```
{"DATE": ["2010-01-01", "2012-01-01"], "(MEASURES_DIMENSION)": "QUANTITYSOLD",
"CITY": ["Berlin", "Tokyo"]}
```

The SDK framework returns the following Data Runtime and Metadata Runtime JSONs:

Data Runtime JSON

```
{
  "selection": [[0, 1], 0, [0, 1]],
   "data": [
    1479,
    2721,
    1471,
    2700
  ]
   "tuples": [
    [0, 0, 0],
    [1, 0, 0],
    [0, 0, 1],
    [1, 0, 1]
  ],
  "columnCount": 2,
  "rowCount": 2
}
```

The `selection` JSON property `[[0, 1], 0, [0, 1]]` reflects the selection. It contains an array of three elements corresponding to the three dimensions of the result set in the order `DATE`, `Measures`, and `CITY` (see Metadata Runtime JSON below). The index values point at the selected dimension members (see Metadata Runtime JSON below):

- `0, 1 = 2010-01-01, 2012-01-01` for dimension `DATE`
- `0 = QUANTITYSOLD` for dimension `Measures`
- `0, 1 = Berlin, Tokyo` for dimension `CITY`

If your code assumes that the elements of the `selection` JSON property are always integer numbers this may lead to incompatible changes.

The `data` JSON property contains an array with the data values of the selected result set sub-columns in this order: left-to-right cell, first-to-last row.

For each data value, the `tuples` JSON property contains a tuple of indexes to the selected dimensions members. For example, the first tuple `[0, 0, 0]` points at the following dimension members:

- `0 = 2010-01-01` for dimension **DATE**

- `0 = QUANTITYSOLD` for dimension **Measures**
- `0 = Berlin` for dimension **CITY**

The second tuple `[1, 0, 0]` points at the following dimension members:

- `1 = 2012-01-01` for dimension **DATE**
- `0 = QUANTITYSOLD` for dimension **Measures**
- `0 = Berlin` for dimension **CITY** (see Metadata Runtime JSON below)

The `columnCount` and the `columnCount` JSON properties contain the number of rows and columns of the data, respectively.

Metadata Runtime JSON

```
{
  "dimensions": [
    {
       "key": "DATE",
       "text": "DATE",
       "axis": "COLUMNS",
       "axis_index": 0,
       "members": [
        {
           "key": "2010-01-01",
           "text": "01/01/2010"
        },
        {
           "key": "2012-01-01",
           "text": "01/01/2012"
        }
      ]
    },
    {
       "key": "Measures",
       "text": "Measures",
       "axis": "COLUMNS",
       "axis_index": 1,
       "containsMeasures": true,
       "members": [
        {
           "key": "QUANTITYSOLD",
           "text": "QUANTITYSOLD",
           "formatString": "#,##0;'-'#,##0"
        }
      ]
    }
    ,{
       "key": "CITY",
       "text": "CITY",
       "axis": "ROWS",
       "axis_index": 0,
       "members": [
        {
           "key": "Berlin",
           "text": "Berlin"
        },
        {
           "key": "Tokyo",
           "text": "Tokyo"
        }
      ]
```

```
    }
  ],
   "locale": "en_US"
}
```

The `dimensions` JSON property contains dimension and member information for each dimension relevant for the selection.

The `locale` JSON property contains the browser's locale string.

5.3.7 Result Set Selection

To get the data values of the entire result set, use a data-bound property of type `ResultSet`.

For example, you have the following result set:

	2010-01-01		2012-01-01	
	SALESREVENUE	QUANTITYSOLD	SALESREVENUE	QUANTITYSOLD
Berlin	190,958.00	1,479	393,902.00	2,721
Rio de Janeiro	139,410.00	1,104	259,345.00	1,752
Tokyo	194,392.00	1,471	412,279.00	2,700
Overall Result	524,760.00	4,054	1,065,526.00	7,173

To select the entire result set, use the following selection string:

```
{}
```

or an empty string.

The SDK framework returns the following Data Runtime JSON:

Data Runtime JSON

```
{
  "selection": [-1, -1, -1],
   "data": [
    190958,
    1479,
    393902,
    2721,
    139410,
    1104,
    ...
    7173
  ],
   "tuples": [
    [0, 0, 0],
    [0, 1, 0],
    [1, 2, 0],
    [1, 3, 0],
```

```
    [0, 0, 1],
    [0, 1, 1],
    ...
    [1, 3, 3]
  ],
   "dimensions": [
    {
       "key": "DATE",
       "text": "DATE",
       "axis": "COLUMNS",
       "axis_index": 0,
       "members": [
        {
           "key": "2010-01-01",
           "text": "01/01/2010"
        },
        {
           "key": "2012-01-01",
           "text": "01/01/2012"
        }
      ]
    },
    {
       "key": "Measures",
       "text": "Measures",
       "axis": "COLUMNS",
       "axis_index": 1,
       "containsMeasures": true,
       "members": [
        {
           "key": "SALESREVENUE",
           "text": "SALESREVENUE",
           "formatString": "#.##0,00;'-'#.##0,00"
        },
        {
           "key": "QUANTITYSOLD",
           "text": "QUANTITYSOLD",
           "formatString": "#.##0;'-'#.##0"
        },
        {
           "key": "SALESREVENUE",
           "text": "SALESREVENUE",
           "formatString": "#.##0,00;'-'#.##0,00"
        },
        {
           "key": "QUANTITYSOLD",
           "text": "QUANTITYSOLD",
           "formatString": "#.##0;'-'#.##0"
        }
      ]
    },
    {
       "key": "CITY",
       "text": "CITY",
       "axis": "ROWS",
       "axis_index": 0,
       "members": [
        {
           "key": "Berlin",
           "text": "Berlin"
        },
        {
           "key": "Rio de Janeiro",
           "text": "Rio de Janeiro"
        },
        {
           "key": "Tokyo",
           "text": "Tokyo"
```

```
        },
        {
           "key": "Result",
           "text": "Overall Result",
           "type": "RESULT"
        }
      ]
    }
  ],
   "axis_columns": [
    [0, 0, -1],
    [0, 1, -1],
    [1, 2, -1],
    [1, 3, -1]
  ],
   "axis_rows": [
    [-1, -1, 0],
    [-1, -1, 1],
    [-1, -1, 2],
    [-1, -1, 3]
  ],
  "locale": "en_US"
  "columnCount": 4,
  "rowCount": 4
}
```


The `selection` JSON property reflects the selection. It contains an array of three indexes, corresponding to the three dimensions of the result set, in the order **DATE**, **Measures**, and **CITY**. An index value of -1 indicates that the respective dimension member is unspecified.

The `data` JSON property contains an array with the data values of all result set cells in the following order: left-to-right cell, first-to-last row.

For each data value, the `tuples` JSON property contains a tuple of indexes of the selected dimensions members. For example, the first tuple `[0, 0, 0]` points to the following dimension members:

- `0 = 2010-01-01` for dimension **DATE**
- `0 = SALESREVENUE` for dimension **Measures**
- `0 = Berlin` for dimension **CITY**

The second tuple `[0, 1, 0]` points to the following dimension members:

- `0 = 2010-01-01` for dimension **DATE**
- `1 = QUANTITYSOLD` for dimension **Measures**
- `0 = Berlin` for dimension **CITY**

The `axis_columns` JSON property specifies the column header cells. For each column axis position, this JSON property contains a tuple of indexes of the appropriate dimension members. The indexes are in the order **DATE**, **Measures**, and **CITY**. An index value of `-1` indicates that the respective dimension is not on the column axis. For example, the last tuple `[1, 3, -1]` (representing the last column axis tuple) points to the following dimension members:

- `1 = 2012-01-01` for dimension **DATE**
- `3 = QUANTITYSOLD` for dimension **Measures**
- `-1` = Dimension **CITY** is not on the column axis

The `axis_rows` JSON property specifies the row header cells. For each row axis position, this JSON property contains a tuple of indexes of the appropriate dimension members. The indexes are in the order **DATE**, **Measures**, and **CITY**. An index value of `-1` indicates that the respective dimension is not on the row axis. For

example, the last tuple `[-1, -1, 3]` (representing the last column row tuple) points to the following dimension members:

- `-1` = Dimension **DATE** is not on the axis
- `-1` = Dimension **Measures** is not on the axis
- `3 = Overall Result` for dimension **CITY**

Metadata Runtime JSON

> **i** Note
>
> For a data-bound property of type `ResultSet`, the Metadata Runtime JSON content is part of the Data Runtime JSON. There is no separate Metadata Runtime JSON.

5.3.8 Master Data

Data-bound properties also support data from master data data sources (see option `selectionShape` in "Element <option>" under Elements of the Contribution XML File [page 21]). A master data data source contains dimension member values on one axis (either row or column axis) and no key figures.

For example, you have the following result set representing data from a master data data source. It has two row dimensions `DATE` and `CITY` and no key figures. The dimension `DATE` has the members `2010-01-01`, `2012-01-01`, and `Overall Result`. The dimension `CITY` has the members `Berlin`, `Rio de Janeiro`, `Tokyo`, and `Result`.

Date	City
2010-01-01	Berlin
2010-01-01	Rio de Janeiro
2010-01-01	Tokyo
2010-01-01	Result
2012-01-01	Berlin
2012-01-01	Rio de Janeiro
2012-01-01	Tokyo
2012-01-01	Result
Overall Result	Result

To select the result set, use the following selection string:

```
{}
```

or an empty string ("").

The Component SDK framework returns the following Data Runtime JSON:

Data Runtime JSON

Sample Code

```
{
  "selection": [-1, -1],
  "data": [
    null,
    null,
    null,
    null,
    null,
    null,
    null,
    null,
    null
  ],
  "tuples": [
    [0, 0],
    [0, 1],
    [0, 2],
    [0, 3],
    [1, 0],
    [1, 1],
    [1, 2],
    [1, 3],
    [2, 3]
  ],
  "dimensions": [
   {
      "key": "DATE",
      "text": "DATE",
      "axis":"COLUMNS",
      "axis_index": 0,
      "members": [
        {
          "key": "2010-01-01",
          "text": "01/01/2010"
        },
        {
          "key": "2012-01-01",
          "text": "01/01/2012"
        },
        {
          "key": "SUMME",
          "text": "Overall Result",
          "type": "RESULT"
        }
      ]
    },
    {
      "key": "CITY",
      "text": "CITY",
      "axis": "ROWS",
      "axis_index": 1,
      "members": [
        {
          "key": "Berlin",
```

```
            "text": "Berlin"
          },
          {
            "key": "Rio de Janeiro",
            "text": "Rio de Janeiro"
          },
          {
            "key": "Tokyo",
            "text": "Tokyo"
          },
          {
            "key": "SUMME",
            "text": "Result",
            "type": "RESULT"
          }
        ]
      }
    ],
    "axis_columns": [],
    "axis_rows": [
      [0, 0],
      [0, 1],
      [0, 2],
      [0, 3],
      [1, 0],
      [1, 1],
      [1, 2],
      [1, 3],
      [2, 3]
    ],
    "locale": "en_US",
    "columnCount": 0,
    "rowCount": 9
}
```


The `selection` JSON property reflects the selection. It contains an array of two indexes corresponding to the two dimensions of the result set in the order `DATE` and `CITY`. An index value of `-1` indicates that the respective dimension member is unspecified.

The `data` JSON property contains `null` values.

For each `data` value, the `tuples` JSON property contains a tuple of indexes of the selected dimensions members. For example, the first tuple `[0, 0]` points to the following dimension members:

- `0 = 2010-01-01` for dimension `DATE`
- `0 = Berlin` for dimension `CITY`

The second tuple `[0, 1]` points to the following dimension members:

- `0 = 2010-01-01` for dimension `DATE`
- `1 = Rio de Janeiro` for dimension `CITY`

The `axis_rows` JSON property contains the same tuples as the `tuples` JSON property.

The `axis_columns` JSON property contains an empty array.

The `rowCount JSON` property contains the number of dimension members on the row axis.

The `columnCount` JSON property contains the value `0`.

In general, the following rules apply:

- If the dimension members are located on the **row axis** then the `axis_rows` JSON property and the `tuples` JSON property contain the same tuples. The `axis_columns` JSON property is empty. The `rowCount` JSON property contains the number of dimension members on the row axis. The `columnCount` JSON property contains the value `0`.
- If the dimension members are located on the **column axis** then the `axis_columns` JSON property and the `tuples` JSON property contain the same tuples. The `axis_rows` JSON property is empty. The `columnCount` JSON property contains the number of dimension members on the columns axis. The `rowCount` JSON property contains the value `0`.

Metadata Runtime JSON

> **i** Note
>
> The Metadata Runtime JSON content is part of the Data Runtime JSON. There is no separate Metadata Runtime JSON.

5.4 Sample Implementation

To learn about data binding with a real, simple data-bound extension component, you can import the *Simple Table* SDK extension. It is contained in the sample project `com.sap.sample.simpletable` in the *SDK Templates and Samples* folder. This is located, for example, under `C:\ds_sdk` (see Importing a Sample SDK Extension [page 10]).

The *Simple Table* displays up to three columns of data from columns (or rows) of a result set. The top cell of each column displays a column header text. An additional (first) column displays row header texts for each row.

Example: Simple Table

	01/01/2010 SALESREVENUE	01/01/2010 QUANTITYSOLD	01/01/2012 SALESREVENUE
Berlin	190,958.00	1,479	393,902.00
Rio de Janeiro	139,410.00	1,104	259,345.00
Tokyo	194,392.00	1,471	412,279.00
Result	524,760.00	4,054	1,065,526.00

5.4.1 Configuring the Simple Table

Procedure

1. To fill a *Simple Table* with data, assign a data source to the table.

> **Note**
>
> You can assign a data source by dragging and dropping a data source from the *Outline* view onto the data-bound extension component on the canvas.

2. In the *Properties* view of the *Simple Table*, you will see the properties *Column 1*, *Column 2*, and *Column 3*.
3. Click the button ... to the right of *Column 1*.

   The *Select Data* dialog box appears.
4. Select a column and close the dialog box.
5. Repeat steps 3 to 5 for *Column 2* and *Column 3*.

Results

The system displays the selection strings in the *Properties* view.

5.4.2 Data Binding in the Simple Table

In the *Simple Table* implementation, there are two locations relevant for data binding:

- Contribution XML
- Component JavaScript

5.4.2.1 Contribution XML

In the `contribution.xml` file, there are two locations relevant for data binding:

- Attribute `databound`
- Data-bound properties

Attribute `databound`

The `databound` attribute in the `<component>` element enables data binding for the *Simple Table*:

```
<component ... databound="true">
```

> **Note**
>
> - This automatically adds the `Data Source` property to the *Simple Table*. It is displayed in the *Properties* view of the extension component in SAP Lumira Designer.
> - This automatically adds the `metadata` property to the *Simple Table*.

Data-Bound Properties

The *Simple Table* uses three data-bound properties to provide data cell values of three result set columns at runtime:

```
<property id="column1"type="ResultCellList"title="Column 1"group="DataBinding">
  <option name="includeFormattedData"value="true"/>
  <option name="includeData"value="false"/>
</property>
<property id="column2"type="ResultCellList"title="Column 2"group="DataBinding">
  <option name="includeFormattedData"value="true"/>
  <option name="includeData"value="false"/>
</property>
<property id="column3"type="ResultCellList"title="Column 3"group="DataBinding">
  <option name="includeFormattedData"value="true"/>
  <option name="includeData"value="false"/>
</property>
```

The three properties `column1`, `column2`, and `column3` are displayed as *Column 1*, *Column 2*, and *Column 3* under *Data Binding* in the *Properties* view of the *Simple Table*. As they are properties of type `ResultCellList`, each property receives the data values of a single column (or row) from the result set at runtime. The selected column (or row) is specified at design time by the selection string (see Design Time Property Values [page 72]).

The properties `column1`, `column2`, and `column3` use options to include the `formattedData` JSON property and to remove the `data` JSON property in the Data Runtime JSON object. Therefore the properties only provide formatted data and not the float number data.

5.4.2.2 Component JavaScript

The Component JavaScript creates the visual appearance of the *Simple Table* component. This involves creating an HTML table and filling it with appropriate result set data.

Function `init`

The `init()` function of the *Simple Table* component adds a CSS style class and a vertical scrollbar to the `<div>` element provided by the SDK framework. Then it creates a `<table>` element (which holds the HTML table) and adds it to the `<div>` element.

```
this.init = function() {
  this.$().addClass(CSS_CLASS_DIV);
  this.$().css("overflow-y", "scroll");
  this.jqTable = $("<table class=\"" + CSS_CLASS_TABLE + "\"/>");
  this.$().append(this.jqTable);
};
```

Property Setter and Getter Functions

Three property setter/getter functions store and return values of the Simple Table's data-bound properties `column1`, `column2`, and `column3`. The setter clauses store the property values provided by the SDK framework in local variables `column1_data`, `column2_data`, and `column3_data` and the getter clauses return these values. The JavaScript code, which is executed after the property values have been set by the SDK framework using the setter clauses, can access these values. This especially applies to JavaScript code in function `afterUpdate()`.

> **i Note**
>
> A fourth setter/getter function `this.metadata()` stores and returns the value of the metadata property in the local variable `meta_data`. The property is implicitly added when declaring a property of type `ResultCell`, `ResultCellList`, or `ResultCellSet` in the Contribution XML.

```
var column1_data = null;
var column2_data = null;
var column3_data = null;
var meta_data = null;
this.column1 = function(value) {
  if (value === undefined) {
    return column1_data;
  } else {
    column1_data = value;
    return this;
  }
};
this.column2 = function(value) {
  if (value === undefined) {
    return column2_data;
  } else {
    column2_data = value;
    return this;
  }
};
this.column3 = function(value) {
  if (value === undefined) {
    return column3_data;
  } else {
    column3_data = value;
    return this;
  }
};
this.metadata = function(value) {
  if (value === undefined) {
    return meta_data;
  } else {
    meta_data = value;
    return this;
  }
};
```

Function `afterUpdate`

> **i Note**
>
> Most extension components place their visualization code in this function, because it is executed after all extension component property values have been updated by the SDK framework.

Function `afterUpdate()` fills the HTML table.

```
this.afterUpdate = function() {
  this.jqTable.empty();
  var column_data = getAnySetColumn_Data();
  if (column_data) {
    var jqHeader = $("<thead/>").appendTo(this.jqTable);
    var jqHeaderRow = $("<tr class=\"" + CSS_CLASS_TR_HEADER +
        "\"/>").appendTo(jqHeader);
    jqHeaderRow.append($("<td class=\"" + CSS_CLASS_TD_HEADER + "\"/>"));
    appendColumnHeaderCell(jqHeaderRow, column1_data);
    appendColumnHeaderCell(jqHeaderRow, column2_data);
    appendColumnHeaderCell(jqHeaderRow, column3_data);
    for (var i = 0; i < column_data.formattedData.length; i++) {
      var jqRow = $("<tr/>");
      this.jqTable.append(jqRow);
      appendRowHeaderCell(jqRow, i);
      appendCell(jqRow, column1_data, i);
      appendCell(jqRow, column2_data, i);
      appendCell(jqRow, column3_data, i);
    }
  }
};
```

First, nested elements are removed from the `<table>` element.

Then, the code checks if any of the column variables contain a value with the following helper function:

```
function getAnySetColumn_Data() {
  if (column1_data && column1_data.formattedData) {
    return column1_data;
  } else if (column2_data && column2_data.formattedData) {
    return column2_data;
  } else if (column3_data && column3_data.formattedData) {
    return column3_data;
  }
  return null;
}
```

If a column variable does contain a value, the table header is composed. First, a `<thead>` (table header) element is created and added to the table. Then a `<tr>` (table row) element is added to the table header. Next, a `<td>` (table cell) element, which is an empty header cell, is added to the table row. The helper function `appendColumnHeaderCell()` is called three times to add the remaining three column header table cells.

A loop adds a `<tr>` (table row) element for each row in the result set to the table. Helper functions `appendRowHeaderCell()` and `appendCell()` add four table cells (one row header table cell and three data table cells) to each table row.

Note how the number of rows is determined in the loop: Variable `column_data` contains the Data Runtime JSON of one the properties `column1`, `column2` or `column3`. The JavaScript expression `column_data.formattedData` returns the value of the `formattedData` JSON property of the Data Runtime

JSON, which is an array of string values. Adding `.length` to this expression returns the number of data values in this array, which is the number of rows of the result set.

Function `appendCell`

Helper function `appendCell()` adds a cell to a table row. The cell contains the appropriate value from the result set. The passed arguments are a `<tr>` (table row) element, the Data Runtime JSON of a property of type `ResultCellList` and a row index.

After various safety checks (Is data available, in other words, does a Data Runtime JSON of the property actually exist? Is the row index in the correct range?), the JavaScript expression `column_data.formattedData[i]` picks the appropriate formatted data value from the Data Runtime JSON using the row index. The cell text is placed into a `<td>` (table cell) element, which is added to the table row.

```
function appendCell(jqRow, column_data, i) {
  if (column_data && column_data.formattedData && (i <
column_data.formattedData.length)) {
    var cellText = column_data.formattedData[i];
    jqRow.append($("<td class=\"" + CSS_CLASS_TD_DEFAULT + "\">" + cellText + "</
td>"));
  }
}
```

Function `appendRowHeaderCell`

Helper function `appendRowHeaderCell()` adds a row header cell to each table row. A row header cell contains a text concatenation of all row dimension member values in that row. The passed arguments are a `<tr>` (table row) element and a row index.

```
function appendRowHeaderCell(jqRow, i) {
  var column_data = getAnySetColumn_Data();
  if (meta_data && column_data && column_data.formattedData && (i <
column_data.tuples.length)) {
    var tuple = column_data.tuples[i];
    var headerText = "";
    for (var j = 0; j < tuple.length; j++) {
      if (column_data.selection[j] == -1) {
        headerText += " " + meta_data.dimensions[j].members[tuple[j]].text;
      }
    }
    headerText = headerText.replace("|", " ");  // Delimiter used for multiple
presentations
    jqRow.append($("<td class=\"" + CSS_CLASS_TD_HEADER + "\">" + headerText +
"</td>"));
  }
}
```

After various safety checks (Is metadata available, in other words, does a Metadata Runtime JSON actually exist? Is data available, in other words, does a Data Runtime JSON of one of the properties `column1`, `column2` or `column3` actually exist? Is the row index in the correct range?), the JavaScript expression `column_data.tuples[i]` picks the appropriate tuple from the Data Runtime JSON using the row index. The tuple contains dimension member indexes for each dimension.

An empty row header text is defined.

A loop over the number of dimensions (equal to the number of tuple elements `tuples.length`) combines the row header text. Only row dimension members are combined to make the row header text. This is achieved by checking if the appropriate dimension member is flagged as unspecified `(= -1)` in the `selection` JSON property of the Data Runtime JSON. Since we made a **column** selection, this means column dimension member indexes in the `selection` JSON property are unequal to `-1` and row dimension member indexes are equal to `-1`. If a row dimension member is found, its dimension information is picked from the Metadata Runtime JSON with expression `meta_data.dimension[j]` and the corresponding dimension member is retrieved with `.members[tuple[j]]`. This member returns the dimension member text with `.text`. Finally, the combined row header text is placed into a `<td>` (table cell) element, which is added to the table row.

Function `appendColumnHeaderCell`

Helper function `appendColumnHeaderCell()` adds a column header cell to the table header row. A column header cell contains a text concatenation of all column dimension member values in that column. The passed arguments are a `<tr>` (table header row) element and the Data Runtime JSON of the property representing that column.

```
function appendColumnHeaderCell(jqHeaderRow, column_data) {
  if (column_data && column_data.formattedData) {
    var headerText = "";
    for (var i = 0; i < column_data.selection.length; i++) {
      var selectionIndex = column_data.selection[i];
      if (selectionIndex != -1) {
        headerText += " " + meta_data.dimensions[i].members[selectionIndex].text;
      }
    }
    $("<td class=\"" + CSS_CLASS_TD_HEADER + "\">" + headerText + "</
td>").appendTo(jqHeaderRow);
  }
}
```

After a safety check (Is data available, in other words, does a Data Runtime JSON of the property representing that column actually exist?), an empty column header text is defined.

A loop over the number of dimensions (equal to the number of elements of the Data Runtime JSON `selection` JSON property) combines the column header text.

Only column dimension members are combined to make the column header text. This is achieved by checking if the appropriate dimension member is not flagged as unspecified `(!= -1)` in the `selection` JSON property of the Data Runtime JSON. Since we made a **column** selection, this means column dimension member indexes in the `selection` JSON property are unequal to `-1` and row dimension member indexes are equal to `-1`. If a column dimension member is found, its dimension information is picked from the Metadata Runtime JSON with expression `meta_data.dimensions[i]` and the corresponding dimension member is retrieved with `.members[selectionIndex]`. This member returns the dimension member text with `.text`. Finally, the combined column header text is placed into a `<td>` (table cell) element, which is added to the table header row.

5.5 Select Data Dialog Box

In the *Properties* view of Lumira Designer, the following properties are displayed with an input field (into which you can type a selection string) and a value help button:

- *ResultCell*
- *ResultCellList*
- *ResultCellSet*
- *ResultSet*

Using the value help button makes creating a selection string easier. The *Select Data* dialog box appears. In this dialog box, you can create your selection based on the result set data. Your selection is automatically restricted by the property type.

Example

You can only select a single cell for a property of type *ResultCell*, whereas you can select a single row or column of cells for a property of type *ResultCellList*.

When you close the dialog box, the relevant selection string is displayed in the *Properties* view.

> **!** Restriction
>
> - The *Select Data* dialog box does not support all types of queries.
> - The *Select Data* dialog box only supports selections of multiple rows or columns for properties of type *ResultCellSet*.

6 SDK Extensions Using SAPUI5 Controls

The SDK also allows you to create SDK extension components based on SAPUI5 controls. The SDK uses the SAPUI5 extension mechanism of SAPUI5 to first extend an SAPUI5 control and then modify it. An SDK component inherits all the properties of the extended SAP UI5 component.

For more information, see the *SAPUI5 Developer Guide* at https://sapui5.hana.ondemand.com/sdk/#docs/guide/OnTheFlyControlDefinition.html.

The following sections describe the modifications that can be made to SDK extensions and extension components when creating SDK extension components based on SAPUI5 controls. The necessary modifications are described using the *RatingIndicator* component of the UI5 SDK sample extension as an example.

SAPUI5 comes in two flavors: SAPUI5 and SAPUI5 m. Both flavors provide a set of SAPUI5 controls, many of them with the same or similar functionality. When you develop an SDK component based on an SAPUI5 control, you must decide which SAPUI5 flavor your SDK component will be based on. Note, however, that with a litte extra effort, you can base your SDK compont on both flavors, as the *RatingIndicator* SDK component example shows.

6.1 Contribution XML

Example: File `contribution.xml` of the SAPUI5 SDK extension

```
<?xml version="1.0" encoding="UTF-8"?>
<sdkExtension ...
  id="com.sap.sample.ui5">
  ...
  <component ...
    id="RatingIndicator"
    handlerType="sapui5"
    modes="commons m"
    group="sapui5">
    <requireJs modes="commons">res/js/components</requireJs>
    <requireJs modes="m">res/js/components_m</requireJs>
    ...
    <property ...
      id="value"
      type="float"/>
    <property ...
      id="onChange"
      type="ScriptText"/>
  </component>
</sdkExtension>
```

The component ID `RatingIndicator` (ID does not have to match the name of the extended SAPUI5 control) is combined with the extension ID `com.sap.sample.ui5`, to create the unique extension component ID `com.sap.sample.ui5.RatingIndicator` for the *RatingIndicator* SDK extension component.

The component handler type must be `sapui5`.

To provide access to the property `value` of the SAPUI5 *RatingIndicator* control, define a property with the same name and type for the SDK component.

The modes attribute of the element `<component>` indicates which SAPUI5 library this SDK component supports (see Element <component> [page 23]). The value `"commons m"` indicates that the *RatingIndicator* SDK component supports both the SAPUI5 and the SAPUI5 m libraries and can therefore be used with analysis applications created from a template based on the SAPUI5 or SAPUI5 m library.

Each `<requireJs>` element references a Component JavaScript file for the *RatingIndicator* SDK component (see Element <requireJs> [page 29]). The first Component JavaScript file is used for the *RatingIndicator* SDK component in analysis applications based on the SAPUI5 library. The second Component JavaScript file is used for the *RatingIndicator* SDK component in analysis applications based on the SAPUI5 m library.

To provide access to a property of the SAPUI5 *RatingIndicator* control, define a property of the same name and type for the *RatingIndicator* SDK component. For example, see the definition of the property `value` in the example above.

6.2 Component JavaScript

The component JavaScript of an SAPUI5-based SDK extension component uses a different syntax than a normal SDK extension component, because it has to follow SAPUI5 rules.

The SDK framework lets you specify the order in which resource files of your extension component, like JavaScript and CSS files, are loaded before the Component JavaScript is executed. This also applies for SAPUI5-based SDK extension components, as is the case with normal SDK extension components. Just nest the content of the Component Javascript in the `define` function as described in Loading Resources in a Specific Order [page 45].

The following example shows the Contribution JavaScript of the *RatingIndicator* SDK component that is supported by the SAPUI5 library. Note that the *RatingIndicator* SDK component with ID `com.sap.sample.ui5.RatingIndicator` extends the SAPUI5 control `sap.ui.commons.RatingIndicator`.

Example: File contribution.js of the SAPUI5 Component SDK extension for SDK components based on the SAPUI5 library

Sample Code

```
define([], function() {
  //...
  sap.ui.commons.RatingIndicator.extend("com.sap.sample.ui5.RatingIndicator",
{
    initDesignStudio: function() {
      this.attachChange(function() {
        this.fireDesignStudioPropertiesChanged(["value"]);
        this.fireDesignStudioEvent("onChange");
      });
    },
    renderer: {}
  });
  //...
});
```

The `define` function is provided by the RequireJS library. It lets you specify the order in which resource files are loaded by passing elements to the array and by passing arguments to the anonymous function (see Loading Resources in a Specific Order [page 45]). The *RatingIndicator* SDK component has no additional resource files, so the array is empty and the function has no arguments.

Incidentally, the define function is used in the same way by regular SDK components (with `handlerType=div`). In that case, however, both a first array element and a first function argument must be present (see Loading Resources in a Specific Order [page 45]).

The following example shows the Contribution JavaScript of the *RatingIndicator* SDK component that is supported by the SAPUI5 m library. Note that the *RatingIndicator* SDK component with ID `com.sap.sample.ui5.RatingIndicator` extends the SAPUI5 m control `sap.m.RatingIndicator`.

Example: File contribution_m.js of the SAPUI5 Component SDK extension for SDK components based on the SAPUI5 m library

Sample Code

```
define([], function() {
  sap.m.RatingIndicator.extend("com.sap.sample.ui5.RatingIndicator", {
    initDesignStudio: function() {
      this.attachChange(function() {
        this.fireDesignStudioPropertiesChanged(["value"]);
        this.fireDesignStudioEvent("onChange");
      });
    },
    renderer: {}
  });
});
```

Extension Component Lifecycle

SAPUI5-based SDK extension components also use the concept of a rendering lifecycle with functions that you can override, similar to normal SDK extension components. The following functions are called in the specified sequence during the rendering lifecycle:

- `initDesignStudio()`
- `beforeDesignStudioUpdate()`
- Property Setter and Getter Functions
- `afterDesignStudioUpdate()`
- `renderer()`

Related Information

JavaScript Function Calls [page 106]
Events [page 109]

6.2.1 JavaScript Function Calls

Function `initDesignStudio`

Syntax: `initDesignStudio()`

Implement this function to execute JavaScript code when the SAPUI5-based SDK extension component is rendered for the first time, after the SAPUI5-based SDK extension component has been created. Usually you attach event listeners in this function.

Function `beforeDesignStudioUpdate`

Syntax: `beforeDesignStudioUpdate()`

Implement this function to execute JavaScript code before the properties of the SAPUI5-based SDK extension component are updated.

Property Setter and Getter Functions

For SDK extension component properties that map to SAPUI5 control properties, no explicit getter/setter functions are necessary. The mapping is configured automatically by the SDK framework.

For SDK extension component properties that do not map to SAPUI5 control properties, you can implement a getter and a setter function. Their names follow this convention (note the uppercase and lowercase letters): For the property `fooProp`, the getter function is named `getFooProp`, and the setter function is named `setFooProp`.

Example: The SDK component in the following example has a property `copyrightText` of type `String`. It is defined in the file `contribution.xml` of the SDK component as follows:

```
...
<property
  id="copyrightText"
  title="Copyright Text"
  type="String"/>
...
```

Since this property is not a property of the SAPUI5 component, which the SDK component is based on, you need to define explicit getter and setter functions in the component's JavaScript file (unlike the properties of the SAPUI5 component, which are available automatically). Note that the first letter after `get` and `set` is in uppercase:

```
...
getCopyrightText: function() {
  returnthis.copyrightText;
},
setCopyrightText: function(copyrightText) {
  this.copyrightText = copyrightText;
}
```

```
...
```

Function `afterDesignStudioUpdate`

Syntax: `afterDesignStudioUpdate()`

Implement this function to execute JavaScript code after the properties of the SAPUI5-based SDK extension component have been updated.

Function `renderer`

Syntax: `renderer()`

If this SAPUI5 function is not empty, it contains renderer code. Usually it is empty. This is because you want to leverage the rendering of the SAPUI5 control, which this SAPUI5-based SDK extension component is based on. For more information, see the *SAPUI5 Developer Guide* at https://sapui5.hana.ondemand.com/sdk/#docs/guide/OnTheFlyControlDefinition.html.

Function `callZTLFunction`

Syntax: `callZTLFunction(sMethodname, function, arg1, arg2, ...)`

Call this function to execute a method of the Lumira Designer Script contribution file `contribution.ztl`.

The argument `sMethodname` is the name of the method.

The argument `function` is a JavaScript function that is executed after the method call and the result of the method call is passed.

The arguments `arg1, arg2, ...` are arguments of the method. Arguments should be strings, JSONs, or arrays.

> **i Note**
>
> You can also call private Lumira Designer Script contribution methods.

> **⚠ Caution**
>
> Do not modify data sources during a call of `callZTLFunction`, for example by calling `setFilter`. This adds an error message to the Lumira Designer error log.

In the example below, the private Lumira Designer Script contribution method `getDimension` is called (without arguments). The result is passed to a component setter.

Example:

(`contribution.ztl`)

Sample Code

```
@Visibility(private)
String getDimensions() {*
  //...
  return ...;
*}
```

(`contribution.js`)

Sample Code

```
//...
that.callZTLFunction("getDimensions", function(result) {
  that.setItems(result);
});
```

Function `callZTLFunctionNoUndo`

Syntax: `callZTLFunctionNoUndo(sMethodname, function, arg1, arg2, ...)`

This is similar to the `callZTLFunction` function but it doesn't record the resulting state changes made by the application's undo stack.

6.2.2 JavaScript Tips

How can I add my own function to the JavaScript of an SAPUI5-based SDK component?

Example: To implement the function `myFunction(arg1, arg2)`, add the following:

```
...
myFunction: function(arg1, arg2) {
  // method body
},
...
```

How can I share variables in my functions in the JavaScript of an SAPUI5-based SDK component?

Use `this.` with the variable name.

> **i Note**
>
> Choose a variable name that does not conflict with the variable names of the SAPUI5 component, which your SDK component is based on.

Example: To get and set the value of variable `myVariable` in function `myFunction`, add the following:

```
...
myFunction1: function() {
  // ...
  this.myVariable = x;  // set the shared variable value
  // ...
},
myFunction2: function() {
  // ...
  var x = this.myVariable;  // get the shared variable value
  // ...
},
...
```

6.3 Events

SAPUI5-based SDK extension components also have event methods, similar to normal SDK extension components:

Function `fireDesignStudioPropertiesChanged`

Syntax: `fireDesignStudioPropertiesChanged([sPropertyname1, sPropertyname2, ...])`

Call this function to inform the SDK framework that one or more properties of this SAPUI5-based SDK extension component have changed in the browser.

> **i Note**
>
> Calling `fireDesignStudioPropertiesChanged` triggers a server roundtrip. Therefore, frequent use of this function may decrease the performance of your analysis application. We recommend that this function should only be called upon user interaction. We do not recommend calling this function to implement implicit changes to properties (so-called event cascading), as this may lead to a large number of (or even infinite) server roundtrips. Lumira Designer's standard components only trigger server roundtrips upon user interaction. This ensures efficient use of server roundtrips, which leads to better performance and avoids the threat of indeterministic (or even infinite) server roundtrips through event cascading.

Function `fireDesignStudioEvent`

Syntax: `fireDesignStudioEvent(sPropertyname)`

Call this function to execute the Lumira Designer script that is stored in a property of type `ScriptText` of this SAPUI5-based SDK extension component.

> **i Note**
>
> Calling `fireDesignStudioEvent` triggers a server roundtrip. Therefore, frequent use of this function may decrease the performance of your analysis application. We recommend that this function should only be called upon user interaction. We do not recommend calling this function to implement implicit changes to properties (so-called event cascading), as this may lead to a large number of (or even infinite) server roundtrips. Lumira Designer's standard components only trigger server roundtrips upon user interaction. This ensures efficient use of server roundtrips, which leads to better performance and avoids the threat of indeterministic (or even infinite) server roundtrips through event cascading.

Example

File `components.js` of the SAPUI5 SDK extension

```
...
sap.ui.commons.RatingIndicator.extend("com.sap.sample.ui5.RatingIndicator", {
  initDesignStudio: function() {
    this.attachChange(function() {
      this.fireDesignStudioPropertiesChanged(["value"]);
      this.fireDesignStudioEvent("onChange");
    });
  },
  renderer: {}
});
...
```

In the first line, the *RatingIndicator* SDK extension component is extended from the SAPUI5 *RatingIndicator* control using the unique extension component ID `com.sap.sample.ui5.RatingIndicator`.

Function `initDesignStudio()` implements initialization tasks. Event listeners are usually attached to the SAPUI5 control here in order to map SAPUI5 event listening to SDK event listening. Note the SAPUI5 naming convention: The name of the function starts with `attach`, followed by the SAPUI5 event name with the first letter in uppercase. For more information, see the *SAPUI5 Developer Guide* at https://sapui5.hana.ondemand.com/sdk/#docs/guide/OnTheFlyControlDefinition.html.

In the event listener code you can trigger:

- the update of SDK component properties with the `fireDesignStudioPropertiesChanged()` method (`value` is an SDK extension component property of type `float` defined in `contribution.xml`)
- the execution of Lumira Designer scripts with the `fireDesignStudioEvent()` methods (`onChange` is an SDK extension component property of type `ScriptText` defined in `contribution.xml`)

Function `renderer()` implements the actual rendering of the component. If this is left empty, then the renderer of the SAPUI5 parent class `sap.ui.commons.RatingIndicator` renders the SDK component.

Function `fireDesignStudioPropertiesChangedAndEvent`

Syntax: `fireDesignStudioPropertiesChangedAndEvent([sPropertyname1, sPropertyname2, ...], sPropertyname);`

This function is equivalent to

```
fireDesignStudioPropertiesChanged([sPropertyname1, sPropertyname2, ...]);
fireDesignStudioEvent(sPropertyname);
```

Function `fireDesignStudioPropertiesChangedAndEvent` is a faster implementation of this frequent combination of function calls requiring only one server round-trip.

> **i** Note
>
> Calling `fireDesignStudioPropertiesChangedAndEvent` triggers a server roundtrip. Therefore, frequent use of this function may decrease the performance of your analysis application.
>
> We recommend that this function should only be called upon user interaction. We do not recommend calling this function to implement implicit changes to properties (event cascading), as this may lead to a large number of (or even infinite) server roundtrips. Lumira Designer's standard components only trigger server roundtrips upon user interaction. This ensures efficient use of server roundtrips, which leads to better performance and avoids the threat of indeterministic (or even infinite) server roundtrips through event cascading.

7 SDK Extensions as Data Sources (Data Source SDK)

In addition to creating SDK components that simply visualize data from a data source, you can also create SDK components that act as data sources for SDK components (SDK data sources). In other words, not only can you create SDK components that **consume** data but also SDK components that **produce** data.

This enables SDK components to use SDK data sources, in order to access a broad range of data sources, for example, a local file, a Web service or a new type of back end system. When you implement an SDK data source, you implement the actual access to the data and supply the data to SDK components using the APIs of the Data Source SDK, which is a part of the Component SDK.

Restrictions

SDK data sources can be consumed by SDK components and standard components, with the exception of the standard *Crosstab* component and standard filter components such as *Dimension Filter* and *Filter Panel*.

7.1 Using SDK Data Sources in SAP Lumira Designer

SDK data sources are added to and removed from a SAP Lumira Designer installation like any other SDK component.

SDK data sources do not appear in Lumira Designer's *Components* view. In order to add an installed SDK data source to your application, follow these steps:

1. In Lumira Designer, right-click the *Data Sources* folder in the *Outline* view.
2. Choose *Add Custom Data Source...*. A submenu appears with a list of installed SDK data sources.
3. Choose one of the listed SDK data sources.

Restrictions

- In general, an SDK data source operates on the provided data. It has no built-in concept of background dimensions, which can be used for filtering data, like normal data sources. However, you can implement SDK data sources that provide this background dimension-like behavior.
- The *Select Data* dialog box in the *Properties* view does not currently support SDK data sources.

7.2 Implementing an SDK Data Source

SDK data sources have the same project structure as any other SDK component. The following sections list and explain the differences.

Prerequisites

You have understood sections SDK Extensions [page 20] and SDK Extensions and Data Binding [page 71].

Contribution XML

The handler type of an SDK data source component is

```
handlerType="datasource"
```

Component JavaScript

There are two ways of implementing the Component JavaScript part of an SDK data source:

- You can extend your SDK data source from the `DataSource` JavaScript class, which is provided by the SDK framework. This is the most basic way to implement an SDK data source. It offers you the most control over your SDK data source implementation but requires you to create the potentially intricate Metadata Runtime JSON and Data Runtime JSON objects.
- You can extend your SDK data source from the `DataBuffer` JavaScript class, which is provided by the SDK framework. This class sits on top of, or in other words, extends the basic `DataSource` JavaScript class. It offers you a more convenient way of implementing an SDK data source.

The two implementation options are transparent to application designers working with your SDK data source. They will not be able to recognize which option you used to implement your SDK data source.

7.3 Option 1: Extending the DataSource JavaScript class

The most basic way to implement an SDK data source is to extend it from the `DataSource` JavaScript class, which is provided by the SDK framework. The API that you need to implement consists of only two methods. In a nutshell, both methods return the Metadata Runtime JSON and the Data Runtime JSON objects, as specified in the sections on the “Metadata Runtime JSON” and “Data Runtime JSON” under Runtime Property Values [page 73]. If you find it challenging to create JSON objects that conform to these specifications you might want to try Option 2.

The example below shows an extract of the Component JavaScript of the SDK data source component *Constant Data Source* that extends the `DataSource` JavaScript class:

Example

(File `component.js`)

```
sap.designstudio.sdk.DataSource.subclass("com.sap.sample.constantdatasource.Const
antDataSource", function() {
  var oMetadataRuntimeJson = ...;
  var oFullDataRuntimeJson = ...;
  this.fetchData = function(oSelection, oOptions) {
    return oFullDataRuntimeJson;
  };
  this.metadata = function(value) {
    if(value === undefined) {
      return JSON.stringify(oMetadataRuntimeJson);
    } else{
      return this;
    }
  }
});
```

7.3.1 JavaScript Function Calls

SDK data sources that extend from the `DataSource` JavaScript class share the same Component JavaScript API as other SDK extension components, implementing or calling JavaScript functions like `init`, `beforeUpdate`, `afterUpdate`, `firePropertiesChanged`, or `fireEvent`. However, there are a few additional JavaScript functions that are specific to SDK data sources that extend from the `DataSource` JavaScript class. They are listed in the following sections.

Function `fetchData`

Syntax: `fetchData(oSelection, oOptions)`

Implement this function to return the Data Runtime JSON object as specified in “Data Runtime JSON” under Runtime Property Values [page 73]. The argument `oSelection` is the Design Time JSON object ("selection string") (see Design Time Property Values [page 72]). The argument `oOptions` is a JSON object that contains property options for data-bound properties (see “Element <Option>” under Elements of the Contribution XML File [page 21]).

With your implementation of this function, you may want to evaluate the Design Time JSON object and the property options before constructing and returning the appropriate Data Runtime JSON object.

The example below shows the implementation of this function in the SDK data source component *Constant Data Source*. It ignores the passed selection string and the options, and always returns a constant Data Runtime JSON object (hence the component's name):

Example: (File `component.js`)

```
this.fetchData = function(oSelection, oOptions) {
  return oFullDataRuntimeJson;
};
```

Note that in this SDK data source component, the Metadata Runtime JSON part is always included in the Data Runtime JSON object for simplicity reasons. In a more elaborate implementation, you would decide (based on the Metadata Runtime and Data Runtime JSON specification in combination with the passed options) what properties to include in the Metadata Runtime JSON and Data Runtime JSON objects.

Getter and setter function for property `metadata`

Implement this function as a combined getter and setter function for the property `metadata`. When called as a getter function, it must return the Metadata Runtime JSON object as a string. When called as a setter function, it must return `this` to allow function calls to be chained, thus creating a fluent interface.

The example below shows the implementation of this function in the SDK data source component *Constant Data Source*. This function always returns a constant Metadata Runtime JSON object (hence the component's name). If the `metadata` property is set, it is ignored:

Example: (File `component.js`)

```
this.metadata = function(value) {
  if(value === undefined) {
    returnJSON.stringify(oMetadataRuntimeJson);
  } else{
    return this;
  }
}
```

Function `fireUpdate`

Syntax: `fireUpdate(bWillUpdateServer)`

Call this function to notify the SDK framework that your SDK data source contains updated data. If the optional argument `bWillUpdateServer` is `true`, then the SDK framework also notifies the server on the back end of the change. This may lead to back end roundtrips.

7.3.2 Script Contributions

SDK data sources, like other SDK components, can contribute Lumira Designer script methods, which are defined in their `contribution.ztl` file. Even if your SDK data source does not contribute any Lumira

Designer script methods, you may find it useful to add an empty Script Contribution file, which extends from the `DataSource` JavaScript class but which does not contain any Lumira Designer script methods.

The example below shows the empty Script Contribution file of the SDK data source *Constant Data Source*:

Example (File `contribution.ztl`)

```
class com.sap.sample.constantdatasource.ConstantDataSource extends SdkDataSource
{
  // needed to inherit parent class methods
}
```

This will let your SDK data source automatically inherit the following Lumira Designer script methods (similar to a normal data source):

```
String          getDataAsString(Measure measure, MultiDimFilter selection)
DataCell        getData(Measure measure, MultiDimFilter selection)
DimensionArray  getDimensions(optional AxisEnum axis)
String          getDimensionText(Dimension dimension)
Dimension       getMeasuresDimension()
MemberArray     getMembers(Dimension dimension, int maxNumber)
void            setFilter(Dimension dimension, FilterArray value)
void            clearAllFilters()
void            clearFilter(Dimension dimension)
String          getFilterText(Dimension dimension)
```

Calling these methods in a Lumira Designer script may lead to calls of the `fetchData` method, which has to evaluate the passed arguments and return the appropriate Metadata Runtime and Data Runtime JSON objects.

7.4 Option 2: Extending the DataBuffer JavaScript Class

A more convenient way to implement an SDK data source is to extend it from the `DataBuffer` JavaScript class, which is provided by the SDK framework. The `DataBuffer` JavaScript class extends the more basic `DataSource` JavaScript class and takes care of the potentially intricate details of creating the appropriate Metadata Runtime JSON and Data Runtime JSON objects.

The example below shows an extract from the Component JavaScript class of the SDK data source component CSV Data Source that extends the `DataBuffer` JavaScript class:

Example

(File `component.js`)

```
sap.designstudio.sdk.DataBuffer.subclass("com.sap.sample.csvdatasource.CsvDataSou
rce", function() {
  ...
});
```

7.4.1 JavaScript Function Calls

SDK data sources that extend from the `DataBuffer` JavaScript class share the same Component JavaScript API as other SDK extension components, which implement or call JavaScript functions like `init`, `beforeUpdate`, `afterUpdate`, `firePropertiesChanged`, `fireEvent`. However, there are a few additional JavaScript functions that are specific to SDK data sources, which extend from the `DataBuffer` JavaScript class. These additional functions are listed in the following sections.

7.4.1.1 Function `defineDimensions`

Syntax: `defineDimensions(aoDimensions, oExternalMeasuresDimension)`

You must call this function to set the dimensions of your SDK data source.

The argument `aoDimensions` contains an array of JSON objects, each JSON object defining a dimension. This argument is the value of the `dimensions` JSON property of the Metadata Runtime JSON object (see "Metadata Runtime JSON" under Runtime Property Values [page 73]).

The optional argument `oExternalMeasuresDimension` contains a JSON object, defining an external dimension. This argument is equivalent to the single element of the array `externalDimensions`, a JSON property of the Metadata Runtime JSON object (see link above).

The example below shows an extract of the Component JavaScript class of the SDK data source component *CSV Data Source* that extends the `DataBuffer` JavaScript class. It defines a column dimension `cols`, a row dimension `rows`, and an external dimension `measures`.

The example specifies dimension members directly with the the `members` JSON property (for the external dimension `measures`). The example leaves other dimension members unspecified (for column dimension `cols` and row dimension `rows`); those dimensions' members are created automatically when data is added with `setData` later.

Example

(File `component.js`)

```
this.defineDimensions([{
  "key": "cols",
  "text": "Columns",
  "axis": "COLUMNS",
  "axis_index": 0
}, {
   "key": "rows",
   "text": "Rows",
   "axis": "ROWS",
   "axis_index": 0
}], {
   "key": "measures",
   "text": "Measures",
   "containsMeasures": true,
   "members": [{
```

```
    "key": "Measure",
    "text": "Measure",
    }]
});
```

Typically, you call this function in the `init` function of the Component JavaScript of your SDK data source.

The example below shows an extract of the Component JavaScript class of the SDK data source component *CSV Data Source* that extends the `DataBuffer` JavaScript class:

Example: (File `component.js`)

```
this.init = function() {
  this.defineDimensions(...);
};
```

7.4.1.2 Function `setDataCell`

Syntax: `setDataCell(aCoordinates, value)`

Call this function to set the value of a single data cell of your SDK data source.

The argument `aCoordinates` contains either an array of dimension names (provided that dimension members were specified in the previous call of `defineDimensions`) or dimension member indexes. Either way, the array specifies the coordinates of the data cell.

The argument `value` contains the new value of the data cell. It is a float number, a string, or `null`.

If the value is a float number then it is added to the `data` JSON property (an array) of the Data Runtime JSON. Then, the value is converted to a string and added to the `formattedData` JSON property (an array) of the Data Runtime JSON.

If the value is a string then it is added to the `formattedData` JSON property (an array) of the Data Runtime JSON. Then, the SDK framework attempts to convert the value to a float number, which is added to the `data` JSON property (an array) of the Data Runtime JSON.

If the value is `null` then it is added to both the `data` JSON property (an array) of the Data Runtime JSON and the `formattedData` JSON property (an array) of the Data Runtime JSON.

> ⚠ Caution
>
> When populating your data source with data cells, you must strictly follow this sequence: Set the data cells left-to-right first, then top-to-bottom. Adding data cells randomly (with respect to their coordinates) may lead to an unexpected arrangement of data cells.

The example below shows the correct sequence for setting the data cells of an SDK data source, in order to provide the data of the following result set: The result set has 3 dimensions, with the first dimension (`products`) in the rows and the remaining two dimensions (`year` and `City`) in the columns, with 2 x 2 x 3 member values:

	2013			2014		
	Berlin	**Sydney**	**Tokyo**	**Berlin**	**Sydney**	**Tokyo**
Product 1	1	2	3	4	5	6
Product 2	7	8	9	10	11	12

Define the dimensions (in function `init`) with:

```
this.defineDimensions({
  [
    {
      "key": "year",
      "text": "Year",
      "axis": "COLUMNS",
      "axis_index": 0,
      "members": [
        {
          "key": "2013",
          "text": "2013"
        }, {
          "key": "2014",
          "text": "2014"
        }
      ]
    }, {
      "key": "City",
      "text": "city",
      "axis": "COLUMNS",
      "axis_index": 1,
      "containsMeasures": true,
      "members": [
        {
          "key": "berlin",
          "text": "Berlin"
        }, {
          "key": "sydney",
          "text": "Sydney"
        }, {
          "key": "tokyo",
          "text": "Tokyo"
        }
      ]
    }, {
      "key": "products",
      "text": "Products",
      "axis": "ROWS",
      "axis_index": 0,
      "members": [
        {
          "key": "product1",
          "text": "Product 1"
        }, {
          "key": "product2",
          "text": "Product 2"
        }
      ]
    }
  ],
  "locale": "en"
});
```

Add the data (in method `afterUpdate`) using dimension members names. Note the particular sequence in which data are added:

```
this.setDataCell(["2013", "berlin", "product1"], 1);
this.setDataCell(["2013", "sydney", "product1"], 2);
this.setDataCell(["2013", "tokyo", "product1"], 3);
this.setDataCell(["2014", "berlin", "product1"], 4);
this.setDataCell(["2014", "sydney", "product1"], 5);
this.setDataCell(["2014", "tokyo", "product1"], 6);
this.setDataCell(["2013", "berlin", "product2"], 7);
this.setDataCell(["2013", "sydney", "product2"], 8);
this.setDataCell(["2013", "tokyo", "product2"], 9);
this.setDataCell(["2014", "berlin", "product2"], 10);
this.setDataCell(["2014", "sydney", "product2"], 11);
this.setDataCell(["2014", "tokyo", "product2"], 12);
```

An alternative way to add the data is using dimension member indexes instead of dimension member names:

```
this.setDataCell([0, 0, 0], 1);
this.setDataCell([0, 1, 0], 2);
this.setDataCell([0, 2, 0], 3);
this.setDataCell([1, 0, 0], 4);
this.setDataCell([1, 1, 0], 5);
this.setDataCell([1, 2, 0], 6);
this.setDataCell([0, 0, 1], 7);
this.setDataCell([0, 1, 1], 8);
this.setDataCell([0, 2, 1], 9);
this.setDataCell([1, 0, 1], 10);
this.setDataCell([1, 1, 1], 11);
this.setDataCell([1, 2, 1], 12);
```

ℹ Note

In case, you do not specify dimension members, you can proceed like this:

You defined the dimensions without members, for example, with

```
this.defineDimensions({
  [
    {
      "key": "year",
      "text": "Year",
      "axis": "COLUMNS",
      "axis_index": 0
    }, {
      "key": "City",
      "text": "city",
      "axis": "COLUMNS",
      "axis_index": 1,
      "containsMeasures": true
    }, {
      "key": "products",
      "text": "Products",
      "axis": "ROWS",
      "axis_index": 0
    }
  ],
  "locale": "en"
});
```

You can add data using dimension member indexes only (as no dimension member names are available):

```
this.setDataCell([0, 0,0], 1);
this.setDataCell([0, 1,0], 2);
```

```
this.setDataCell([0, 2,0], 3);
this.setDataCell([1, 0,0], 4);
this.setDataCell([1, 1,0], 5);
this.setDataCell([1, 2,0], 6);
this.setDataCell([0, 0,1], 7);
this.setDataCell([0, 1,1], 8);
this.setDataCell([0, 2,1], 9);
this.setDataCell([1, 0, 1], 10);
this.setDataCell([1, 1,1], 11);
this.setDataCell([1, 2,1], 12);
```

This is the resulting result set (note the dimension member indexes, which were created automatically):

	0			1		
	0	1	2	0	1	2
0	1	2	3	4	5	6
1	7	8	9	10	11	12

7.4.1.3 Function `fillWithArray`

Syntax: `fillWithArray(aData, bHasHeaderRow, bHasHeaderColumn)`

If your SDK data source contains 2-dimensional data (arranged like a spreadsheet), and you defined a single row and column dimension with `defineDimensions` you can use this function to initialize the data cells in one go from an array of data. The necessary dimension members are created automatically.

The argument `aData` contains the data arranged as a nested 2-dimensional array. For example, an array of 3 columns x 2 rows containing data is expressed as `[[1, 2, 3], [4, 5,6]]`.

The argument `bHasHeaderRow` indicates whether the data also contains the column header titles. If set to `true` then the first array element of the data contains the column header titles. They are used to name the column dimension members, which are created automatically.

> **i** Note
>
> To work properly, all column header titles must differ from each other, as they serve as dimension member names of the column dimension.

If set to `false` the column dimension members, which are created automatically, are named using letters A, B, C, and so on.

The argument `bHasHeaderColumn` indicates whether the data also contain the row header titles. If set to `true` then the first element of each array element of the data contains a row header title. They are used to name the row dimension members, which are created automatically.

> **i** Note
>
> To work properly, all row header titles must differ from each other, as they serve as dimension member names of the row dimension.

If set to `false` the row dimension members, which are created automatically, are named using numbers 0, 1, 2, and so on.

Example

The example below shows the initialization of the data cells of an SDK data source from an array of 3 columns x 3 rows, with the first row containing the column header titles:

```
fillWithArray([[1, 2, 3], [4, 5, 6], [7, 8, 9]], false, false);
```

This is the resulting result set (note the titles of the column and row dimension members, which were generated automatically):

	A	**B**	**C**
1	1	2	3
2	4	5	6
3	7	8	9

Example

The example below shows the initialization of the data cells of an SDK data source from an array of 3 columns x 3 rows with the first row containing the column header titles:

```
fillWithArray([["Column1", "Column2", "Column3"], [1, 2, 3], [4, 5, 6]], true,
false);
```

This is the resulting result set (note the titles of the row dimension members, which were generated automatically):

	Column1	**Column2**	**Column3**
3	4	5	6
3			
2	1	2	3

Example

The example below shows the initialization of the data cells of an SDK data source from an array of 3 columns x 3 rows with the first column containing the row header titles:

```
fillWithArray([["Row1", 1, 2], ["Row2", 3, 4], ["Row3", 5, 6]], false, true);
```

This is the resulting result set (note the titles of the row dimension members, which were generated automatically):

	B	C
Row1	1	2
Row2	3	4
Row3	5	6

Example

The example below shows the initialization of the data cells of an SDK data source from an array of 3 columns x 3 rows with the first row containing the column header titles and the first column containing the row header titles:

```
fillWithArray([["Column 0", "Column1", "Column2"], ["Row 1", 1, 2], ["Row 2", 3,
4]], true, true);
```

This is the resulting result set (note that the first element of the first element of the array is ignored):

	Column1	Column2
Row 1	1	2
Row 2	3	4

7.4.1.4 Function `clear`

Syntax: `clear(bClearMembers)`

Call this function to reset the SDK data source to its initial state. In particular, this function clears all previously set data information. If the optional argument `bClearMembers` is `true` then the member information is also cleared . This is useful if function `setDataCell` is used with dimension member names (and not dimension member indexes) that automatically create dimension members. Note that the JSON property `externalDimensions` is never cleared.

7.4.1.5 Function `fireUpdate`

Syntax: `fireUpdate(bWillUpdateServer)`

Call this function to notify the SDK framework that your SDK data source contains updated data. If the optional argument `bWillUpdateServer` is `true`, then the SDK framework also notifies the server on the back end of the change. This may lead to back end roundtrips.

> **i** Note
>
> This method is inherited from the `DataSource` JavaScript class, as SDK data sources extending from the `DataBuffer` JavaScript class (which in turn extends from `DataSource` JavaScript class) also inherit the methods of `DataSource`.

7.4.2 Script Contributions

SDK data sources, like other SDK components, may contribute Lumira Designer script methods, which are defined in their `contribution.ztl` file. Even if your SDK data source does not contribute any Lumira Designer script methods, you may find it useful to add an empty Script Contribution file extending from the `DataBuffer` JavaScript class but containing no Design Script methods.

The example below shows the empty Script Contribution file of the SDK data source **CSV Data Source**:

Example

(File `contribution.ztl`)

```
classcom.sap.sample.csvdatasource.CsvDataSource extendsSdkDataBuffer {
  // needed to inherit parent class methods
}
```

This will let your SDK data source automatically inherit the following Lumira Designer script methods (similar to a normal data source):

```
String         getDataAsString(Measure measure, MultiDimFilterselection)
DataCell       getData(Measure measure, MultiDimFilter selection)
DimensionArray getDimensions(optional AxisEnum axis)
String         getDimensionText(Dimension dimension)
Dimension      getMeasuresDimension()
MemberArray    getMembers(Dimension dimension, intmaxNumber)
void           setFilter(Dimension dimension, FilterArray value)
void           clearAllFilters()
void           clearFilter(Dimension dimension)
String         getFilterText(Dimension dimension)
```

8 Sample Components

In this section, you will find information about the available sample components, in particular the prerequisites, usage, properties, and Lumira Designer script API methods.

> **i** Note
>
> You can download the sample components under *Component SDK Templates and Samples* on SAP Help Portal at http://help.sap.com.

8.1 Colored Box

Sample component that displays a colored rectangle.

This component is an example of a minimal SDK extension component.

How To Proceed

Drag and drop a *Colored Box* into the editor area.

Properties

Name	Type	Description
Color	Color	The color of the *Colored Box*
On Click	Script Text	The Lumira Designer script that is executed when the user clicks the *Colored Box*

Lumira Designer Script API

- `void setColor(String newColor)`
  Sets the color of the *Colored Box*.

Parameters

Name	Type	Description
newColor	String	The new color of the *Colored Box*. All CSS-like color values can be used, for example `"red"` or `"#FF0000"`.

- `String getColor()`
  Returns a string containing the color of the *Colored Box*.

8.2 Simple Table

Sample component that displays up to three columns of key figures from a data source in a table.

This component is an example of a data-bound SDK extension component.

Prerequisites

You need a data source, which contains three key figure columns.

How to Proceed

1. Drag and drop a *Simple Table* into the editor area.
2. Assign a data source to the *Data Source* property.
3. Assign columns of key figures from the data source to the properties *Column1*, *Column2*, and *Column3*. When you assign a column of key figures to this *Simple Table* for the first time, an additional column is displayed on the left of this column. The additional column contains the dimension member values of the rows.

   > **i** Note
   >
   > You can also assign rows of key figures from the data source. However, when you mix columns and rows of key figures, the resulting table may look unexpected.

Properties

Name	Type	Description
DataSource	DataSource	The data source of the *Simple Table*

Name	Type	Description
Column 1	ResultCellList	The first column of key figures displayed in the *Simple Table*
Column 2	ResultCellList	The second column of key figures displayed in the *Simple Table*
Column 3	ResultCellList	The third column of key figures displayed in the *Simple Table*

Lumira Designer Script API

- `void setColumn1Selection(ResultCellListSelection selection)`
  Sets the column of key figures to be displayed in the first column of key figures in the *Simple Table*. When you set columns of this *Simple Table* for the first time, an additional column is displayed on the left of this column. The additional column contains the dimension member values of the rows.

  Parameters

Name	Type	Description
selection	ResultCellListSelection	A selection that specifies a single column (or row) of key figures from a data source

- `void setColumn2Selection(ResultCellListSelection selection)`
  Sets the column of key figures to be displayed in the second column of key figures in the *Simple Table*. When you set columns of this *Simple Table* for the first time, an additional column is displayed on the left of this column. The additional column contains the dimension member values of the rows.

  Parameters

Name	Type	Description
selection	ResultCellListSelection	A selection that specifies a single column (or row) of key figures from a data source

- `void setColumn3Selection(ResultCellListSelection selection)`
  Sets the column of key figures to be displayed in the third column of key figures in the *Simple Table*. When you set columns of this *Simple Table* for the first time, an additional column is displayed on the left of this column. This additional column contains the dimension member values of the rows.

  Parameters

Name	Type	Description
selection	ResultCellListSelection	A selection that specifies a single column (or row) of key figures from a data source

8.3 Simple Crosstab

Sample component that display the data of a data source in a crosstab.

This component is an example of a data-bound SDK extension component.

Prerequisites

You need a data source.

How to Proceed

1. Drag and drop a *Simple Crosstab* into the editor area.
2. Assign a data source to the *Data Source* property.

Properties

Name	Type	Description
DataSource	DataSource	The data source of the *Simple Crosstab*
Data Selection	ResultSet	The displayed result set
On Select	ScriptText	The Lumira Designer script that is executed after the user makes a selection in the *Simple Crosstab*

Lumira Designer Script API

- `void setDataSelection(ResultSetSelection selection)`
  Sets a data selection for to the *Simple Crosstab*. This filters the displayed result set so that only the data selection is displayed.

  Parameters

Name	Type	Description
selection	ResultSetSelection	A data selection from a data source

- `String getVisualSelection()`
  Returns a string; a specification of the data cells in the *Simple Crosstab* currently visually selected by the user.

- `void setVisualSelection(ResultSetSelection selection)`
  Visually selects data cells in the *Simple Crosstab*.

  Parameters

Name	Type	Description
selection	ResultSetSelection	A selection that specifies the visually selected data cells in the *Simple Crosstab*.

- `Member getSelectedMember(Dimension dimension)`
  Returns a member; the visually selected dimension member of the *Simple Crosstab*. Member is `null` if the dimension has no visually selected dimension member.

  Parameters

Name	Type	Description
dimension	Dimension	The dimension of the selected member

8.4 Google Maps

Sample component that displays a Google map.

This component is an example of an SDK extension component, which uses a third party JavaScript API.

Prerequisites

You need a Google API key (learn more about how to obtain a Google API Key on https://developers.google.com/maps/documentation/javascript/tutorial#api_key)

1. In the file `contribution.xml` of this SDK component extension, locate the `<component>` element with an `id` of `GoogleMaps`.
2. Add the Google API key after the keyword `key` in element `<jsInclude>http://maps.googleapis.com/maps/api/js?key=...`
   This enables the SDK extension component to use the Google Maps JavaScript API.

How to Proceed

Drag and drop a *Google Maps* into the editor area.

Properties

Name	Type	Description
Map Type	String	The map type. Possible values:`"hybrid"`, `"roadmap"`, `"satellite"`, `"terrain"` (default setting: `"roadmap"`).
Zoom	int	The zoom factor. Possible values: `0` and greater. The value `0` shows the world map (default setting: `14`).
On Zoom	ScriptText	The Lumira Designer script that is executed when the user zooms the Google map.

Lumira Designer Script API

- `void setZoom(int value)`
  Sets the zoom factor of the Google map.

  Parameters

Name	Type	Description
value	int	The zoom factor. Possible values: integers of `0` and greater. The value `0` shows the world map.

- `int getZoom()`
  Returns an integer; the zoom factor of the Google map. Possible values: integers of `0` and greater. The value `0` shows the world map.

8.5 Google Maps with Data

Sample component that displays a Google map overlaid with vertical bar charts at specific geographical locations. The values and geographical locations of the bar charts are retrieved from a data source.

This component is as an example of an SDK extension component, which uses a third party JavaScript API.

Prerequisites

- You need a data source that contains a dimension with adresses (city names are sufficent) and two key figures in the columns.

- You need a Google API key (learn more about how to obtain a Google API Key on https://developers.google.com/maps/documentation/javascript/tutorial#api_key)

1. In the file `contribution.xml` of this SDK component extension, locate the `<component>` element with an `id` of `GoogleMaps`.
2. Add the Google API key after the keyword `key` in element `<jsInclude>http://maps.googleapis.com/maps/api/js?key=...`
   This enables the SDK extension component to use the Google Maps JavaScript API to map addresses to geographical locations on the Google map.

How to Proceed

1. Drag and drop a *Google Maps with Data* into the editor area.
2. Assign the data source to the *Data Source* property.
3. Assign the dimension containing the addresses to the *Address Dimension* property.
4. Assign a column of key figures to the *Red Markers* property.
5. Optional: assign a column of key figures to the *Blue Markers* property.
6. Optional: assign values to the *Red Scaling Factor* and *Blue Scaling Factor* properties to scale the bar charts.
7. Hide the result row of the data source. Its dimension member - and the corresponding bar charts - cannot be correctly mapped to a geographical location.

Properties

Name	Type	Description
DataSource	DataSource	The data source of the *Google Maps with Data* component
Address Dimension	String	The column dimension of the data source that contains addresses (city names are sufficient)
Red Markers	ResultCellList	A column of key figures from the data source
Red Scaling Factor	int	Red marker key figures are divided by this value before being displayed on the Google map (default setting: `10000`).
Blue Markers	ResultCellList	A column of key figures from the data source
Blue Scaling Factor	int	Blue marker key figures are divided by this value before being displayed on the Google map (default setting: `10000`).

8.6 Timer

Sample component that executes a Lumira Designer script periodically.

This component is an example of an SDK extension component without visualisation.

How To Proceed

1. In Lumira Designer's *Outline* view, right-click *Technical Components* and select *Create* > *Timer*.
2. Assign a Lumira Designer script to the *On Timer* property.
3. Assign a time interval in milliseconds to the *Interval in Milliseconds* property.
4. Start and stop the *Timer* sing the Timer's `start()` and `stop()` Lumira Designer script commands.

Properties

Name	Type	Description
Interval in Milliseconds	int	The time interval of the *Timer* (default setting: `1000`)
On Timer	ScriptText	The Lumira Designer script that is executed periodically, each time the time interval elapses.

Lumira Designer Script API

- `void start()`
  Starts the *Timer*. This executes the Lumira Designer script of the *On Timer* property periodically, each time the time interval elapses.
- `void stop()`
  Stops the *Timer*. This stops the Lumira Designer script of the *On Timer* property.
- `boolean isRunning()`
  Returns `true` if the *Timer* has been started or `false` if the *Timer* has been stopped.

8.7 Clock

Sample component that displays an animated clock.

This component is an example of an animated SDK extension component.

How To Proceed

Drag and drop a *Clock* into the editor area.

Properties

Name	Type	Description
Railway Clock	boolean	If set to `true`, the *Clock* is displayed as a railway clock. If set to `false`, the clock is displayed as a regular clock (default setting: `false`).

8.8 JSONGrabber

Sample component that displays the Metadata Runtime JSON and Data Runtime JSON strings of data-bound property types. This allows you to examine the format and content of these strings.

Prerequisites

You need a data source.

How To Proceed

1. Drag and drop a *JSONGrabber* into the editor area.
2. Assign a data source to the *Data Source* property.
3. Assign an appropriate data selection from the data source to one of the properties *Selection Shape 0 (ResultCell)*, *Selection Shape 1 (ResultCellList)*, *Selection Shape 2 (ResultCellSet and ResultSet)*, or *Selection Shape 3 (ResultCellSet and ResultSet with master data support)*.
4. Select the property whose JSON strings you want to display in the *JSONGrabber* with the *Show Data-Bound Property* property.
5. Optional: set the *PrettyPrint* property to *true* to pretty print the JSON strings.

Properties

Name	Type	Description
Data Source	DataSource	The data source of the *JSONGrabber*
Selection Shape 0 (ResultCell)	ResultCell	The data-bound property that holds a result set of selection shape 0 (ResultCell)
Selection Shape 1 (ResultCell-List)	ResultCellList	The data-bound property that holds a result set of selection shape 1 (ResultCellList)
Selection Shape 2 (ResultCell-Set and ResultSet)	ResultSet	The data-bound property that holds a result set of selection shape 2 (ResultCellSet and ResultSet)
Selection Shape 3 (ResultCell-Set and ResultSet with Master Data support)	ResultSet	The data-bound property that holds a result set of selection shape 3 (ResultCellSet and ResultSet with Master Data support)
Show Data-Bound Property	String	Displays the selected data-bound property. Possible values: *"SelectionShape0"*, *"SelectionShape1"*, *"SelectionShape2"*, and *"SelectionShape3"* (default setting: *"SelectionShape2"*).
Pretty Print	boolean	If set to `true`, the JSON strings are pretty-printed (default setting: `false`).

8.9 KPI Tile

Sample component that displays a single key figure from a data source in a highly customizable tile-like box.

Prerequisites

You need a data source that contains a key figure.

How To Proceed

1. Drag and drop a *KPI Tile* into the editor area.
2. Assign a data source to the *Data Source* property.
3. Assign a key figure from the data source to the *Data Value* property.
4. Optional: configure other properties of the *KPI Tile*.

Properties

Name	Type	Description
Data Source	DataSource	The data source of the *KPI Tile*
Data Value	ResultCell	The result cell that contains the key figure displayed in the *KPI Tile*
Header	String	The header text (default setting: *"Header"*)
Header Visible	boolean	If set to `true`, the header is visible. If set to `false`,the header is hidden (default setting: `true`)
Header CSS Class	String	The header CSS class
Title Text	String	The title text (default setting: *"Title"*)
Title CSS Class	String	The title CSS class
Value Prefix Text	String	The value prefix text
Value Prefix Position	String	The value prefix position. Possible values: *"superscript"*, *"normal"*, *"subscript"* (default setting: *"subscript"*).
Value Prefix CSS Class	String	The value prefix CSS class
Value Text	String	The value text (default setting: *"Value"*)
Value CSS Class	String	The value CSS class
Value Horizontal Alignment	String	The value horizontal alignment. Possible values: *"left"*, *"right"* (default setting: *"left"*).
Value Decimal Places	int	The number of decimal places of the displayed value. Possible values range from `0` to `9` (default setting: `0`).
Value Suffix Text	String	The value suffix text (default setting: *"M$"*)
Value Suffix Position	String	The value suffix position. Possible values: *"superscript"*, *"normal"*, *"subscript"* (default setting: *"subscript"*).
Value Suffix CSS Class	String	The value suffix CSS class
Footer	String	The footer text (default setting: *"Footer"*)
Footer CSS Class	String	The footer CSS class
Footer Horizontal Alignment	String	The footer horizontal alignment. Possible values: *"left"*, *"right"* (default setting: *"left"*).
On Click	ScriptText	The Lumira Designer script that is executed when the user clicks the *KPI Tile*

Lumira Designer Script API

- `void setHeaderText(String text)`
  Sets the header text.

Parameters

Name	Type	Description
text	String	The header text

- `String getHeaderText()`
  Returns a string containing the header text.
- `void setHeaderVisible(boolean isHeaderVisible)`
  Shows or hides the header.

  Parameters

Name	Type	Description
isHeaderVisible	Boolean	If set to `true`, the header is shown. If set to `false`,the header is hidden

- `boolean isHeaderVisible()`
  Returns `true` if the header is shown or `false` if the header is hidden.
- `void setHeaderCssClass(String cssClass)`
  Sets the header CSS class.

  Parameters

Name	Type	Description
cssClass	String	The header CSS class

- `String getHeaderCssClass()`
  Returns a string containing the header CSS class.
- `void setTitleText(String text)`
  Sets the title text.

  Parameters

Name	Type	Description
text	String	The title text

- `String getTitleText()`
  Returns a string containing the title text.
- `void setTitleCssClass(String cssClass)`
  Sets the title CSS class.

  Parameters

Name	Type	Description
cssClass	String	The title CSS class

- `String getTitleCssClass()`
  Returns a string containing the title CSS class.
- `void setValuePrefixText(String text)`
  Sets the value prefix text.

Parameters

Name	Type	Description
text	String	The value prefix text

- `String getValuePrefixText()`
  Returns a string containing the value prefix text.
- `void setValuePrefixCssClass(String cssClass)`
  Sets the value prefix CSS class.

  Parameters

Name	Type	Description
cssClass	String	The value prefix CSS class

- `String getValuePrefixCssClass()`
  Returns a string containing the value prefix CSS class.
- `void setValueText(String text)`
  Sets the value text.

  Parameters

Name	Type	Description
text	String	The value text

- `String getValueText()`
  Returns a string containing the value text.
- `void setValueCssClass(String cssClass)`
  Sets the value CSS class.

  Parameters

Name	Type	Description
cssClass	String	The value CSS class

- `String getValueCssClass()`
  Returns a string containing the value CSS class.
- `void setValueHAlign(String hAlign)`
  Sets the value horizontal alignment.

  Parameters

Name	Type	Description
hAlign	String	The value horizontal alignment. Possible values: *"left"*, *"right"*.

- `String getValueHAlign()`
  Returns a string containing the value horizontal alignment. Possible values: *"left"*, *"right"*.
- `void setValueDecimalPlaces(int decimalPlaces)`
  Sets the number of decimal places of the value.

Parameters

Name	Type	Description
decimalPlaces	int	The number of decimal places of the value. Valid values are between `0` and `9`.

- `int getValueDecimalPlaces()`
  Returns an integer, the number of decimal places of the value. Valid returned values are between `0` and `9`.
- `void setValueSuffixText(String text)`
  Sets the value suffix text.

  Parameters

Name	Type	Description
text	String	The value suffix text

- `String getValueSuffixText()`
  Returns a string containing the value suffix text.
- `void setValueSuffixCssClass(String cssClass)`
  Sets the value suffix CSS class.

  Parameters

Name	Type	Description
cssClass	String	The value suffix CSS class

- `String getValueSuffixCssClass()`
  Returns a string containing the value suffix CSS class.
- `void setFooterText(String text)`
  Sets the footer text.

  Parameters

Name	Type	Description
text	String	The footer text

- `String getFooterText()`
  Returns a string containing the footer text.
- `void setFooterCssClass(String cssClass)`
  Sets the footer CSS class.

  Parameters

Name	Type	Description
cssClass	String	The footer CSS class

- `String getFooterCssClass()`
  Returns a string containing the footer CSS class.
- `void setFooterHAlign(String hAlign)`
  Sets the footer horizontal alignment.

Parameters

Name	Type	Description
hAlign	String	The footer horizontal alignment. Possible values: *"left"*, *"right"*.

- `String getFooterHAlign()`
  Returns a string containing the footer horizontal alignment. Possible values returned:*"left"*, *"right"*.
- `void setDataSelection(ResultCellSelection cellSelection)`
  Sets the result cell whose value is displayed by the *KPI Tile*.

  Parameters

Name	Type	Description
cellSelection	ResultCellSelection	The result cell that contains the value displayed by the *KPI Tile*

8.10 Sparkline

Sample component that displays a series of key figures from a data source in a simple line chart.

Prerequisites

You need a data source that contains a series of key figures.

How To Proceed

1. Drag and drop a *Sparkline* into the editor area.
2. Assign a data source to the *Data Source* property.
3. Assign a row or column of key figures from the data source to the property *Data Series*.
4. Optional: configure the visualization of the line chart by modifying the property *CSS Style*.

Properties

Name	Type	Description
Data Source	DataSource	The data source of the *Sparkline*

Name	Type	Description
Data Series	ResultCellList	The result cell list, which represents the series of key figures displayed by the *Sparkline*
CSS Style	String	The CSS style used to configure the visualization of the line chart (default setting: `"stroke:steelblue;stroke-width:1;fill:none;"`)
On Click	Script Text	The Lumira Designer script, which is executed when the user clicks the *Sparkline*

8.11 Exception Icon

Sample component that displays an icon, whose image changes depending on the value of a key figure cell from a data source. You can use this component to create a traffic-light status icon with three different states: green, yellow, and red.

Prerequisites

You need a data source that contains a key figure.

How To Proceed

1. Drag and drop an *Exception Icon* into the editor area.
2. Assign a data source to the *Data Source* property.
3. Assign a key figure from the data source to the property *Data Value*.
4. Assign an image to each of these properties: *Icon Green*, *Icon Yellow*, and *Icon Red*.
5. Assign decreasing threshold values to each of the properties *Value Icon Green*, *Value Icon Yellow*, and *Value Icon Red*.

Properties

Name	Type	Description
Data Source	DataSource	The data source of the *Exception Icon*

Name	Type	Description
Data Value	ResultCell	The result cell, which contains the key figure for selecting the *Exception Icon*'s image.
Icon Green	Url	The URL of the image (16 x 16 pixels) for the green icon. It is either a fully qualified URL or a local file path. The root of the local file path is the folder of the analysis application.
Icon Yellow	Url	The URL of the image (16 x 16 pixels) for the yellow icon. It is either a fully qualified URL or a local file path. The root of the local file path is the folder of the analysis application.
Icon Red	Url	The URL of the image (16 x 16 pixels) for the red icon. It is either a fully qualified URL or a local file path. The root of the local file path is the folder of the analysis application.
Value Icon Green	float	The lower threshold value for the green icon. It must be the largest of the three threshold values, in order to work correctly.
Value Icon Yellow	float	The lower threshold value for the yellow icon. It must lie between the other two threshold values, in order to work correctly.
Value Icon Red	float	The lower threshold value for the red icon. It must be the smallest of the three threshold values, in order to work correctly.
Exact Match	boolean	If set to *true*, the appropriate icon image is displayed - provided that the corresponding, rounded threshold value matches the key figure value exactly (default setting: *false*).
On Click	ScriptText	The Lumira Designer script, which is executed when the user clicks the *Exception Icon*

8.12 Audio

Sample component that plays an audio file.

How To Proceed

1. Drag and drop an *Audio* into the editor area.
2. Play an audio file using the *Audio*'s `play()` Lumira Designer script command.

Lumira Designer Script API

- `void play(String audioUrl)`
  Plays the audio file located at the URL.

  Parameters

Name	Type	Description
audioUrl	Url	The audio file URL. It is either a fully qualified URL or a local file path. The root of the local file path is the folder of the analysis application. **i Note** *Audio* uses HTML5 to play audio. Not all browsers fully support HTML5. For best results, use Google Chrome.

8.13 Video

Sample component that plays a video file.

How To Proceed

1. Drag and drop a *Video* into the editor area.
2. Play a video file using the `Video's play()` Lumira Designer script command.

Lumira Designer Script API

`void play(String videoUrl)`

Plays the video file located at the URL.

Parameters

Name	Type	Description
videoUrl	Url	The video file URL. It is either a fully qualified URL or a local file path. The root of the local file path is the folder of the analysis application. **i Note** Video uses HTML5 to play the video. Not all browsers fully support HTML5 completely. For best results, use Google Chrome.

8.14 ApplicationHeader

Sample component that provides an SAP UI5 ApplicationHeader control as an SDK component.

For more information, see https://sapui5.hana.ondemand.com/sdk/#test-resources/sap/ui/commons/demokit/ApplicationHeader.html.

i Note

This sample component is only available in applications created from a template based on SAPUI5 (not SAPUI5 m).

Properties

Name	Type	Description
Display Logoff	boolean	If set to `true`, the logoff area is displayed on the right of the application header. If set to `false`, the logoff area is not displayed.
Display Welcome	boolean	If set to `true`, the welcome text is displayed. If set to `false`, the welcome text is not displayed. (default setting: `true`)
User Name	String	The user name that is displayed beside the welcome text
Logo Source	Url	The URI to the logo icon that is displayed in the application header
Logo Text	String	The text that is displayed beside the logo in the application header
On Logoff	ScriptText	The Lumira Designer script that is executed when the user logs off from the application

Lumira Designer Script API

- `void setUserName(String userName)`
  Sets the user name that is displayed beside the welcome text.

  Parameters

Name	Type	Description
userName	String	The user name

- `String getHeaderText()`
  Returns a string containing the user name that is displayed beside the welcome text.
- `void setLogoText(String logoText)`
  Sets the text that is displayed beside the logo in the application header.

  Parameters

Name	Type	Description
logoText	String	The logo text

- `String getLogoText()`
  Returns a string containing the text that is displayed beside the logo in the application header.

8.15 ColorPicker

Sample component that provides an SAP UI5 ColorPicker control as an SDK component.

For more information, see https://sapui5.hana.ondemand.com/sdk/#test-resources/sap/ui/commons/demokit/ColorPicker.html.

> **i** Note
>
> This sample component is only available in applications created from a template based on SAPUI5 (not SAPUI5 m).

Properties

Name	Type	Description
Color	Color	Picked color (default setting: *red*)
On Color Change	ScriptText	The Lumira Designer script that is executed when the user clicks the *ColorPicker*

Lumira Designer Script API

- `void setColor(String colorString)`
  Sets the picked color.

  Parameters

Name	Type	Description
colorString	String	The color string. It can be a hexadecimal string (for example *"#FF0000"*), an RGB string (*"rgb(255,0,0)"*), an HSV string (*"hsv(360,100,100)"*), or a CSS color name (*"red"*).

- `String getColor()`
  Returns a string containing the picked color.

8.16 FormattedTextView

Sample component that provides an SAP UI5 FormattedTextView control as an SDK component.

For more information, see https://sapui5.hana.ondemand.com/sdk/#test-resources/sap/ui/commons/demokit/FormattedTextView.html.

> **i Note**
>
> A similar component is available as a basic component in the design tool. For more information, see the "User Interface Reference" in the *Application Designer Guide: Designing Analysis Applications* under *Help* > *Help Contents* in the design tool.

Properties

Name	Type	Description
HTML Text	String	HTML text

Lumira Designer Script API

- `void setHtmlText(String htmlText)`
  Sets the HTML text.

Parameters

Name	Type	Description
htmlText	String	HTML text

- `String getHtmlText()`
  Returns a string containing the HTML text.

8.17 Paginator

Sample component that provides an SAP UI5 Paginator control as an SDK component.

For more information, see https://sapui5.hana.ondemand.com/sdk/#test-resources/sap/ui/commons/demokit/Paginator.html.

> **i Note**
>
> This sample component is only available in applications created from a template based on SAPUI5 (not SAPUI5 m).

Properties

Name	Type	Description
Current Page	int	The current page number
Number of Pages	int	The total number of pages that are embedded into the parent control (default setting: 3)
On Page Change	ScriptText	The Lumira Designer script that is executed when the user navigates to another page by selecting it directly, or by jumping forward or backward

Lumira Designer Script API

- `void setCurrentPage(int currentPage)`
  Sets the current page number.

  Parameters

Name	Type	Description
currentPage	int	The current page number

- `String getHtmlText()`
  Returns an integer containing the current page number.

- `void setNumberOfPages(int numberOfPages)`
  Sets the total number of pages embedded into the parent control.

  Parameters

Name	Type	Description
numberOfPages	int	Total number of pages

- `int getNumberOfPages()`
  Returns an integer containing the total number of pages that are embedded into the parent control.

8.18 ProgressIndicator

Sample component that provides an SAP UI5 ProgressIndicator control as an SDK component. For more information, see https://sapui5.hana.ondemand.com/sdk/#test-resources/sap/ui/commons/demokit/ProgressIndicator.html

Properties

Name	Type	Description
Bar Color	String	**Note** This property is only available in analysis applications created from a template based on SAPUI5 (not SAPUI5m). The color of the bar; one of the following values: *CRITICAL*, *NEGATIVE*, *NEUTRAL*, *POSITIVE* (default setting: *NEUTRAL*
State	String	**Note** This property is only available in analysis applications created from a template based on SAPUI5 (not SAPUI5m). The state (color) of the bar. Possible values are: `"None"`, `"Success"`, `"Warning"`, `"Error"` (default setting: `"None"`
Display Value	String	The text value displayed in the bar.
Enabled	boolean	If set to *true*, the progress indicator is enabled. If set to *false*, the progress bar is disabled.
Percent Value	int	The numerical value for the displayed length of the progress bar.
Show Value	boolean	If set to *true*, the value is shown inside the bar. If set to *false*, the value is not shown (default setting: *true*).

Lumira Designer Script API

- `void setPercentValue(int percentValue)`
  Sets the percentage value of the progress bar.

  Parameters

Name	Type	Description
percentValue	int	The percent value

- `int getPercentValue()`
  Returns an integer containing the percentage value of the progress bar.
- `void setDisplayValue(String displayValue)`
  Sets the text value displayed in the bar.

  Parameters

Name	Type	Description
displayValue	String	The display value

- `String getDisplayValue()`
  Returns a string containing the display value.

8.19 RatingIndicator

Sample component that provides an SAP UI5 RatingIndicator control as an SDK component.

For more information, seehttps://sapui5.hana.ondemand.com/sdk/#test-resources/sap/ui/commons/demokit/RatingIndicator.html.

Properties

Name	Type	Description
Editable	boolean	**i Note** This property is only available in analysis applications created from a template based on SAPUI5 (not SAPUI5m). If set to `true`, the rating indicator is enabled. If set to `false`, the rating indicator is disabled. The value `true` is required for changes on the rating symbols (default setting:`true`).

Name	Type	Description
Icon Hovered	Url	The URI to the image that is displayed when the mouse hovers over a rating symbol. If this is used, then all custom icons must have the same size. Note that when this attribute is set, the other icon attributes also need to be set.
Icon Selected	Url	The URI to the image which shall be displayed for all selected rating symbols. If this is used, then all custom icons must have the same size. Note that when this attribute is set, the other icon attributes also need to be set.
Icon Unselected	Url	The URI to the image which shall be displayed for all unselected rating symbols. If this is used, then all custom icons must have the same size. Note that when this attribute is set, the other icon attributes also need to be set.
Max Value	int	The number of displayed rating symbols (default setting: `5`)
Value	float	The number of displayed rating symbols.
On Change	ScriptText	The Design Studo script that is executed when the user selects a rating.

Lumira Designer Script API

- `void setValue(float value)`
  Sets a rating value.

  Parameters

Name	Type	Description
value	float	The rating value

- `float getValue()`
  Returns a float containing the rating value.

8.20 Rich Text Editor

Sample component that provides an SAP UI5 m RichTextEditor control as an SDK component. For more information see https://wiki.wdf.sap.corp/wiki/display/zen/Design+Studio+SDK+Documentation#DesignStudioSDKDocumentation-RichTextEditor .

Properties

Name	Type	Description
HTML Text	String	The HTML content. Ensure content is well formed if tags are included.
Wrapped	Boolean	If set to true, the text in the editor will be wrapped.
Editable	Boolean	If set to true, the menus in the editor will be set to enabled.

Lumira Designer Script API

- `void setHtmlText(String htmlText)`
  Set HTML content. Can be a plain string or HTML content with Tags. Please ensure content is well formed and can be rendered by the target component (Example FeedListComponent).

  Parameters

Name	Type	Description
htmlText	String	The editor content

- `String getHtmlText()`
  Returns a string containing HTML content. May not match content in the DOM as the editor may do some transformations.
- `void setEditable(boolean editable)`
  Enable menus in the Rich Text Editor. Calls to ZTL method setHtmlText(String) will be accepted even if the control is not editable.

  Parameters

Name	Type	Description
editable	Boolean	Enable editor menus

- `boolean isEditable()`
  Returns a boolean indicating the editors current state.

> **i Note**
>
> When working with the RichTextEditor sample, you should be aware that it is configured to output an unrestricted set of tags. Some components, such as the *Feed List* component only supports a subset of tags. All of these transformations happen on the client side in JavaScript. It is recommended that the editor output is compatible with the intended input control. If the editor output is not compatible with the input control, there may be some situations where the API method `getHtmlText()` will return content that is not in agreement with the actual content in the editor. This happens because the server state is updated by ZTL script. Once the state is changed, the client receives a delta, and may or may not transform that input. In this scenario, the server will not be in agreement with the client HTML content. A simple DOM inspection in the browser will confirm this. To avoid incompatibility issues, take the following steps:

- Ensure the HTML content set using the API is well formed. Do not allow the Caja HTML sanitizer built into TinyMCE to change your input, by making sure your HTML is well formed.
- Ensure that the HTML content contains tags that can be consumed by the components that accept a restricted set of HTML tags. For the list of supported tags, you can refer to https://openui5.hana.ondemand.com/#/api/sap.m.FormattedText.

8.21 Slider

Sample component that provides an SAP UI5 Slider control as an SDK component.

For more information, see https://sapui5.hana.ondemand.com/sdk/#test-resources/sap/ui/commons/demokit/Slider.html.

Properties

Name	Type	Description
Min	float	The minimum value of the *Slider* (default setting: `0`)
Max	float	The maximum value of the *Slider* (default setting: `100`)
Value	float	The current value of the *Slider* (default setting: `0`)
TotalUnits	int	**i Note** This property is only available in analysis applications created from a template based on SAPUI5 (not SAPUI5m). The number of units that are displayed by ticks (default setting: `10`)
Vertical	boolean	**i Note** This property is only available in analysis applications created from a template based on SAPUI5 (not SAPUI5m). If set to `true`, the *Slider* is oriented vertically. If set to `false`, the *Slider* is oriented horizontally (default setting: `false`).
SmallStepWidth	float	**i Note** This property is only available in analysis applications created from a template based on SAPUI5 (not SAPUI5m). The grip can only be moved in steps of this width (default setting: `10`).

Name	Type	Description
On Change	ScriptText	The Lumira Designer script that is executed when the user has changed the position of the grip.

Lumira Designer Script API

- `void setMin(float min)`
  Sets the minimum value.

  Parameters

Name	Type	Description
min	float	Minimum value

- `float getMin()`
  Returns a float containing the minimum value.
- `void setMax(float max)`
  Sets the maximum value.

  Parameters

Name	Type	Description
max	float	Maximum value

- `float getMax()`
  Returns a float containing the maximum value.
- `void setValue(float value)`
  Sets the value.

  Parameters

Name	Type	Description
totalUnits	int	Total units value

- `float getValue()`
  Returns a float containing the value.

8.22 ConstantDataSource

Sample component that acts as a data source providing constant data.

This component is an example of an SDK data source extension component. It is based on the `DataSource` JavaScript class, which is provided by the SDK framework.

How to Proceed

1. In the *Outline* view, right-click the *Data Sources* icon.
2. Choose *Add Custom Data Source* > *Constant Data Source*.
3. Drag and drop a *Simple Crosstab* into the editor area.
4. Assign the new data source to the *Data Source* property of the *Simple Crosstab*.

Properties

Name	Type	Description
On Result Set Changed	ScriptText	The Lumira Designer script that is executed after the result set has been changed.

Selection Strings

SDK components with data-bound properties can reference data values of a data source with a selection string. A selection string can contain zero, one or multiple dimension-member pairs.

For a *ConstantDataSource* SDK data source, selection strings are not honored; data-bound properties referencing the SDK data source are always assigned the full result set.

8.23 CSVDataSource

Sample component that acts as a data source providing data from a CSV file.

This component is an example of an SDK data source extension component. It is based on the `DataBuffer` JavaScript class, which is provided by the SDK framework.

Prerequisites

- You need a CSV file (a file containing one or more rows with each row containing comma-separated values).

How to Proceed

1. Create a new analysis application.
2. Choose *Application* > *Open Repository Folder*.
3. Place your CSV file into this folder.
4. In the *Outline* view, right-click the *Data Sources* icon.
5. Choose *Add Custom Data Source* > *CSV Data Source*.
6. In the *Properties* view of the added data source, enter the name of your CSV file.
7. Drag and drop a *Simple Crosstab* into the editor area.
8. Assign the new data source to the *Data Source* property of the *Simple Crosstab*.

Properties

Name	Type	Description
CSV File	Url	The URL of the CSV file. It is either a fully qualified URL or a local file path. The root of the local file path is the folder of the analysis application.
Has Header Row	boolean	If set to `true`, the first line of the CSV file contains the column header titles (default setting: `false`).
Has Header Column	boolean	If set to `true`, the first element of each line of the CSV file contains the row header titles (default setting: `false`).
On Result Set Changed	ScriptText	The Lumira Designer script that is executed after the result set has been changed.

Selection Strings

SDK components with data bound properties can reference data values of a data source with a selection string. A selection string can contain zero, one, or multiple dimension-member pairs.

For a *CSVDataSource* SDK data source, the dimension part of the dimension-member pair is `"cols"` to reference a column or `"rows"` to reference a row. The member part of the dimension-member pair, the member name, depends on whether the member is already contained in the CSV data:

- For column references: If the property *Has Header Row* is false, then the column member names are not part of the CSV data. Use member names `"A"`, `"B"`, `"C"`, ... (like in Microsoft Excel) to reference the first, second, third, ... columns. If the property *Has Header Row* is true, then the column member names are part of the CSV data. Use the relevant values in the first line of the CSV data to reference the required columns.
- For row references: If the property *Has Header Column* is false, then the row member names are not part of the CSV data. Use member names `"1"`, `"2"`, `"3"`, ... (like in Microsoft Excel) to reference the first, second, third, ... row. If the property *Has Header Column* is true, then the row member names are part of the CSV data. Use the relevant values in the first column of the CSV data to reference the required rows.

Example

You have assigned a *CSVDataSource* SDK data source to a *Simple Table* SDK component. To display the first column of the CSV data in the *Simple Table* SDK component, set the *Column 1* property of the *Simple Table* SDK component to `{"cols":"A"}` and the *Has Header Row* property of the *CSVDataSource* SDK data source to `false`.

Example

You have assigned a *CSVDataSource* SDK data source to a *Simple Table* SDK component. To display the second row of the CSV data in the *Simple Table* component, set the *Column 1* property of the *Simple Table* component to `{"rows":"2"}` and the *Has Header Row* property of the *CSVDataSource* SDK data source to `false`.

Example

The *Simple Table* SDK component has data-bound properties of type `ResultCellList`, each referencing a single row or column. A data-bound property of type `ResultCellSet` is able to reference, for example, multiple columns or rows. To reference the second and third column of the CSV data for such a data-bound property, for example, you can use the selection string `{"cols":["B", "C"]}`.

8.24 ScalingDataSource

Sample component that acts like a data source, whose data can be scaled at runtime.

This component is an example of an SDK data source extension component. It is based on the `DataBuffer` JavaScript class, which is provided by the SDK framework.

Prerequisites

- You need SAP Lumira Designer 1.5 (or higher)..
- You need a data source to be scaled.

How to Proceed

1. Create a new analysis application.

2. Add a data source as *DS_1* to the application.
3. Drag and drop a *Chart* to the analysis application.
4. In the *Outline* view, right-click the *Data Sources* icon.
5. Choose *Add Custom Data Source* > *Scaling Data Source*.
6. In the *Properties* view of the *Scaling Data Source*, click the Binding icon of the *Data* property.
   This automatically binds the data source *DS_1* to this property.
   The entire result set of data source *DS_1* is used as a source for scaling data cells.
7. In the *Properties* view of the *Scaling Data Source*, click the Binding icon of the *Data Range to Scale* property.
   This automatically binds the data source *DS_1* to this property.
   The entire result set of data source *DS_1* is to be scaled.
8. Assign the *Scaling Data Source* as *DS_2* to the *Chart*.
9. Drag and drop a *Button* to the analysis application.
10. Add the following script to the *On Click* event of the *Button*:

```
DS_2.setScalingFactor(5);
```

11. Save and execute the analysis application.
12. Click the *Button*.
    The values in the chart are multiplied by a factor of 5.

Properties

Name	Type	Description
Data	ResultCellSet	Source result set
Data Range to Scale	ResultCellSet	Selection of data cells of the source result set to be scaled by the scaling factor
Scaling Factor	float	Factor to scale data cells of the source result set

Lumira Designer Script API

```
void setScalingFactor(float factor)
```

Sets the factor to scale data from the source result set.

Parameters

Name	Type	Description
factor	float	The scaling factor

8.25 SAPUI5 List

Sample component that displays a SAPUI5 List component.

> **i** Note
>
> This component is only visible in analysis applications that use the SAPUI5 m library.

It demonstrates how to use the `Array` and `Object` property type.

How to Proceed

1. Drag and drop a *SAPUI5 List* into the editor area.
2. In the *Properties* view of the *SAPUI5 List*, click the *Items* property, and add items.

Properties

Name	Type	Description
Items	Array	An array of items, each consisting of a text, a key, and an image URL

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.

Gender-Related Language

We try not to use gender-specific word forms and formulations. As appropriate for context and readability, SAP may use masculine word forms to refer to all genders.

www.sap.com/contactsap

THE BEST RUN SAP