SAP Fiori - Development golden rules

|  |  |  |  |
| --- | --- | --- | --- |
| **Document Identification** | | | |
| **Author** | Amine LOUATI | **Document Location** |  |
| **Version** | **Status** | **Date (DD/MM/YYYY)** | **Classification** |
| 1.0 | Work in Progress | 22/11/2016 | Confidential |

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision History** | | | |
| **Version** | **Date** | **Sections revised** | **Description** |
| 1.0 | 22/11/2016 |  | Initial creation |
|  |  |  |  |
|  |  |  |  |

|  |  |  |
| --- | --- | --- |
| **Reference Documentation** | | |
| **Version** | **Document** | **Description** |
| 1.0 | Development main principles |  |
| 1.0 | Development environment |  |
| 1.0 | Development naming conventions |  |
| 1.0 | Development check |  |

|  |  |  |
| --- | --- | --- |
| **Responsible** | | |
| **Interlocutor** | **Main** | **Backup** |
| **Business** |  |  |
| **Development Shared Services** |  |  |
| **Support** |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Review and Approval** | | | |
| **Name Decathlon Approver** | **Approved (Yes/ No)** | **Comment** | **Date (DD/MM/YY)** |
|  |  |  |  |

**TABLE OF CONTENTS**

[**Purpose of this document**](#_30j0zll)

[**1.1 Basic Files**](#_y02u635i9fxh)

[**1.1.1 Descriptor File - manifest.json**](#_8cy7biiutx9s)

[**1.1.2 Root View - App.view.xml**](#_s29rpz426jzl)

[**1.1.3 Component (Component.js)**](#_9piz8o62npfj)

[**1.1.4 Component Container**](#_egyeauf3a1i8)

[**1.1 Folder Structure**](#_bmpu0bgqlsk3)

[**1.1.1 The root Folder**](#_pzn6vdq4jb9e)

[**1.1.2 The webapp Folder**](#_tgde6tjw1rw8)

[**1.1.2.1 The view folder**](#_xmwelzeklp87)

[**1.1.2.2 The controller folder**](#_hhsz6ph8tlmg)

[**1.1.2.3 The model folder**](#_xebkyhb9vayz)

[**1.1.2.4 The localization folder - i18n**](#_oest5wpnvvyz)

[**1.1.2.5 The fragment folder**](#_3faqgax1ikgw)

[**1.1.2.6 The localService folder**](#_ov0b95gocmi5)

[**1.1.3 The test Folder**](#_q53ppqlpmjy)

[**1.1 JavaScript Coding Guidelines**](#_9xx6u3rjvzt3)

[**1.1.1 Do/Don’t**](#_cxt2lunuas56)

[**1.1.2 Code Formatting**](#_scprmnxqsq22)

[**1.1.3 Documentation - JSDoc**](#_i2fxks8jxch)

[**1.2 CSS Coding Guidelines**](#_1xf9g0oxceul)

[**1.3 Compatibility Rules**](#_wqlw2nod7eux)

[**1.3.1 API Evolution**](#_y16cnmk4gan8)

[**1.3.2 Compatible/Incompatible Changes**](#_15tdntd9p21m)

[**1.3.3 Inheritance**](#_w0i14ih8dyap)

[**1.3.4 Deprecation**](#_dxldai9a76v7)

# Purpose of this document

This document provides the golden rules to apply in all SAP Fiori / UI5 development.

**Important: All Developments have to be aligned with the SAPUI5 libraries version of the SAP Fiori xUx system.**

## 1.1 Basic Files

It is strongly recommended to create the following files in the structure of your application:

- the descriptor (manifest.json)

- the component (Component.js)

- the main view of the app (App.view.xml).

### 1.1.1 Descriptor File - manifest.json

The manifest.json file allows the identification and the description of the configuration and the settings. **It is strongly recommended to add it into SAPUI5 application.**

It is a descriptor where we put in all important information needed to run the app:

- **Attributes for information purposes (such as the minimum SAPUI5 version - minUI5version)**

- Attributes to help external components to integrate the application correctly (for example the SAP Fiori Launchpad)

- Attributes to configure specific aspects of the app that are needed frequently.

The most important configuration settings are:

- **Models**. Examples of models are the configuration of the OData service (default model) and language files (i18n model).

All models described in the manifest.json file are automatically instantiated when the app is started.

- **Libraries** and **components** that are used in the app and have to be loaded during app initialization.

- The **root view** of your application.

- **Routing** configuration that defines the navigation between views.

You can find an example of manifest.json in annexe (Example of manifest.json).

### 1.1.2 Root View - App.view.xml

It defines a global UI structure of the the application (managing position of views, containers, etc…) and allows a management of this .

Type of views in SAPUI5 can be declared in XML, HTML, JavaScript and JSON. The XML views are recommended by SAP because they allows a separation between the definition of the views (UI components) and the view controller (management of rules regarding UI components) which is declared in a JavaScript file associated to the view.

SAP recommends creating a separate file for each view declared in your application.

### 1.1.3 Component (Component.js)

The Component.js file holds the app setup. The init function of the component is automatically started by SAPUI5 when the component is instantiated.

All components extends UIComponent class, so if you decide to override the init function in the component, make sure that the init function of UIComponent is called and the initialization of the router is afterwards.

In the metadata section of the component, you define a reference to the descriptor file. When the component is instantiated, the descriptor is loaded and parsed automatically.

### 1.1.4 Component Container

The component container is mandatory and loads the component when a SAPUI5 application is launched.

Also you need to create an index.html file within a component container is created which refers to the component.

## 1.1 Folder Structure

The following folder structure described can be a best practice for structuring an application containing one component, one OData service and less than 20 views.

For more components, OData services and views, it is recommended to introduce more folder levels.

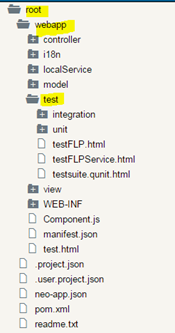
The 3 main folders in an application have to be:

- the root folder

- the webapp folder

- the test folder.

Regarding their structure, it has to be as described in the following screenshot from the SAP Web IDE.



### 1.1.1 The root Folder

The root folder has to contain files that are not part of your application coding.

Examples:

- build configuration files

- pom.xml for maven

- Gruntfile.js for node/grunt

- documentation files like readme.md or txt.

- …

These files can be grouped in folders. For example, all documentation files can be included to a doc folder.

### 1.1.2 The webapp Folder

The webapp folder has to contain all files related to the code of the application. It also contains:

- the JavaScript files for the logic

- view files written in xml, html, json or js format

- files for localization, such as i18n.properties files.

- files relevant for testing

The webapp folder has also to contain:

- 3 folders related to the MVC (model, view, controller) pattern used in SAPUI5

o controller

o model

o view

- a localization folder (i18)

- a local-services folder used for emulating OData services (localService - optional)

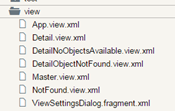
- a test folder (refer to chapter The test Folder - optional)



#### 1.1.2.1 The view folder

This folder has to contain all SAPUI5 views and fragments and no application logic has to be included (no JavaScript files unless you are using JavaScript views – not recommended due to a possible mix between UI and control logic)

The following screenshot shows an example of the view folder containing views and fragments.



We recommend creating subfolders if the view folder becomes too big. These subfolders have to respect a consistency regarding views logic.

#### 1.1.2.2 The controller folder

This folder has to contain all the controllers used by views and additional logic files that are used by controllers.

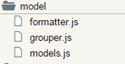
The structure of the controller folder has to be a counterpart of the view folder. If a view is in a subfolder, the controller of the view must also be in the corresponding subfolder.



#### 1.1.2.3 The model folder

This folder has to contain all files needed for creating models and logic relating to model data. It includes grouping, filtering and formatting data.

In the following example, the file models.js is a factory for creating models, formatter.js a formatter of data and grouper.js the JavaScript file related to the grouping.



#### 1.1.2.4 The localization folder - i18n

This folder is dedicated to localization files for translation purposes. Each language has to own its properties file.

**Note** the localization folder is not part of the model folder because the .properties files have a different semantic since they are used for translation.

#### 1.1.2.5 The fragment folder

This folder serves to group all the fragments.

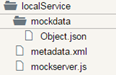
#### 1.1.2.6 The localService folder

**Note** This folder is optional but can be useful during the development.

This folder has to contain all files required to emulate a preview mode or OData services.

The preview mode allows a consumption of data hosted locally (declared in the mockdata subfolder) instead of connecting to a backend

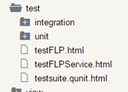
The localService folder also contains one or more metadata.xml files which describe the backend connection (one per OData service).



### 1.1.3 The test Folder

**Note** This folder has to be aligned to the test strategy so it can be optional in certain cases.

The test folder included in the webapp folder has to contain all files needed for running automated tests.



3 sets of files are located in it:

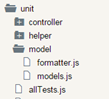
- files related to unit tests

- files related to integration tests

- html files for either launching the tests or for testing the application manually

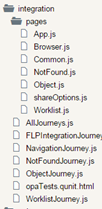
Inside the unit subfolder, the structure of the webapp folder is replicated for the files that are being tested.

For the following example, the files being tested are webapp/model/formatter.js and webapp/model/models.js.



**Note** You may be using JavaScript files or html files have to be used to run tests, depending on the runners executed. SAP recommends using .js files for writing test cases, so that it can be run them with tools such as Karma (<https://karma-runner.github.io/>) or Jasmine (<http://jasmine.github.io/>) for example.

The integration subfolder has to contain the integration tests and OPA tests. SAP recommends separating unit and integration tests in 2 distinct folders because of the data returns and the difference of performances between the 2 kinds of test cases.



## 1.1 JavaScript Coding Guidelines

### 1.1.1 Do/Don’t

- **Don't use private and protected methods or properties of SAPUI5**

Don't use or override "private" and "protected" functions. Private functions are typically (but not always) prefixed with "\_". Protected functions are indicated by a yellow diamond in front of the function name within the API Reference documentation.

Always double check in the API Reference documentation. If SAPUI5 changes the implementation in a future release, your code will break if you fail to follow this guideline.

|  |  |
| --- | --- |
| **Bad** | **Good** |
| var sText = oControl.mProperties["text"]; | var sText = oControl.getText(); |
| oSelectDialog.\_oList.setGrowing(false); |  |
| var sPart = oEvent.oSource.oBindingContexts.description.sPath.split('/')[3]; |  |

- **Don't use deprecated APIs**

Entities marked as “deprecated” in the API Reference documentation (this includes properties, methods, events, and their parameters as well as entire controls and other APIs) are no longer intended to be used. They will not get feature updates in the future. Alternatives, if available, are described in the API Reference documentation.

One prominent example is the old jQuery.sap.device API that has been replaced with sap.ui.Device.

- **Don't override or add control methods**

If you override methods like onBeforeRendering, onAfterRendering, or getters and setters, the original methods will no longer be called. You have to make sure that you call them in your method explicitly. Even if they are not implemented right now, they could be added in the future. This applies to control inheritance in particular.

Instead, you should consider using delegates.

|  |  |
| --- | --- |
| **Bad** | **Good** |
| oControl.onAfterRendering = function() {  // do something  }; | oControl.addEventDelegate({  onAfterRendering:function() {  // do something  }  }); |
| oControl.prototype.setText = function(){ ... }; |  |

- **Don't manipulate the DOM structure within controls**

Manipulating the DOM structure of controls rendered by SAPUI5 can result in undesired behavior and only has a temporary effect. Changes will be overridden after the next rerendering or the DOM might change in a future version of SAPUI5, which can break your code. In addition, your DOM changes could break the code of the SAPUI5 control if it relies on a certain structure.

If you need to manipulate the DOM of an SAPUI5 control, attach a delegate to the afterRendering hook of the control, safeguard your code against DOM changes, but still be prepared to have to rework your code at any time when the DOM structure (which is in no way guaranteed to remain stable!) changes. The adaptation should be covered by your automated tests.

Even onAfterRendering may not be called when a control handles certain property changes without complete rerendering.

|  |  |
| --- | --- |
| **Bad** | **Good** |
| oControl.$().find(".sapMLabel")[0].innerHTML = "reallybad"; | oControl.addEventDelegate({  "onAfterRendering": function() {  var $label = oControl.$().find(".sapMLabel");  if (/\* sanity check whether the change still makes sense \*/) {  // TODO: re-test after UI5 updates, create automated test  $label.text("Better");  }  }  }); |
| oControl.$().find(".sapMLabel").remove(); |  |

- **Don't attach DOM event handlers**

Use attachBrowserEvent() if you need to listen to any DOM event on SAPUI5 controls. An even better approach is to use addEventDelegate() for the most important event types instead, as it avoids additional event registrations and listens to the regular SAPUI5 event dispatching.

If you are creating event handlers in custom controls, you can use listen to DOM events directly, but make sure that the listeners are properly deregistered in onBeforeRendering() and in exit(), and registered in onAfterRendering().

Good example for arbitrary events:

oControl.attachBrowserEvent("mousemove", function() {

// do something

});

Good example for wide but limited selection of browser events:

oControl.addEventDelegate({

onmouseover:function() {

// do something

}

});

- **Don't create global IDs (when running with other views or apps)**

When you create JSViews or applications that will be running together with views or applications from other sources (that are not owned by you), or JSViews that will be instantiated several times in parallel, you must not create static IDs for your controls, fragments, or views in SAPUI5. Doing so might result in duplicate ID errors that will break your application. Especially when running together with other applications, there could be name clashes or other errors.

Use the createId() function of a view or controller instead. This is done automatically in XMLViews and JSONViews. The createId() function adds the View ID as a prefix, thus recursively ensuring uniqueness of the ID (for example: \_\_page0--\_\_dialog0).

|  |  |
| --- | --- |
| **Bad** | **Good** |
| createContent: function(oController) {  var btn = new sap.m.Button("myBtn", {text: "Hello"});  return btn;  } | createContent: function(oController) {  var btn = new sap.m.Button(this.createId("myBtn"), {text: "Hello"});  return btn;  } |

- **Don't forget about control lifecycle management**

SAPUI5 controls are kept alive until they are destroyed, so lifecycle management of controls is important since multiple apps can be opened and closed in the same user session. Controls that are not destroyed cause memory leaks and may slow down the browser after prolonged use.

Also clean up internal structures in controllers, views and your custom controls.

- **Don't hard code or concatenate strings that need to be translatable**

Hard coding UI strings will exclude them from translation. In addition, concatenating translatable strings in applications might lead to errors in internationalization: the texts in question might have a different translation order in other languages and will then be syntactically wrong.

|  |  |
| --- | --- |
| **Bad** | **Good** |
| Using separate texts like " you selected " and " items " in the translation file to construct sentences like: " you selected " + 10 + "items ". This would lead to a wrong word order in languages where the verb needs to be at the end of the sentence, for example. | Using a complete sentence including a placeholder in the translation file: " you selected {0} items ". This allows translators to change the word order and the position of the inserted placeholder value. |

- **Don't forget about proper "this" handling**

For developers new to JavaScript, it is often confusing to understand how the "this" keyword behaves. In event handlers in particular, but also for other callback functions, the "this"-pointer must be used correctly, so make sure you check what it actually refers to. Without proper usage of the execution context, unexpected results can occur (this-pointer might be the global window object or a different control).

- **Don't use console.log()**

There is an API available for logging errors and warnings in the developer console of your browser, but some browsers might even crash while you are using it (because "console" is only defined while the console is actually open). Use jQuery.sap.log.\* instead, which offers different severities as well as additional filter strings. jQuery.sap.log.setLevel() then defines the minimum severity to be logged.

|  |  |
| --- | --- |
| **Bad** | **Good** |
| console.log("some message"); | jQuery.sap.log.info("some message"); |

**Note** Errors and warnings in the developer console thrown by the SAPUI5 framework are potential bugs in your application and must be analyzed thoroughly!

- **Don't use timeouts**

Executing logic with timeouts is often a workaround for faulty behavior and does not fix the root cause. The timing that works for you may not work under different circumstances (other geographical locations with greater network latency, or other devices that have slower processors) or when the code is changed. Use callbacks or events instead, if available.

|  |  |
| --- | --- |
| **Bad** | **Good** |
| jQuery.ajax("someData.json");  setTimeout(fnProcessResults, 300); | jQuery.ajax("someData.json").done(fnProcessResults); |

- **Don't build apps without reasonable automated tests**

This should not come as surprise, but it is very difficult to refactor or modify apps that do not have any (or have bad) automated test cases. There are substantial risks when QUnit and OPA tests are missing in applications.

- **Never make OData calls in loops.**

Design the OData service properly instead.

- **Use Odata models instead of normal Ajax calls.**

OData models are tightly integrated to the UI5 controls also. When the data changes, all the controls will be updated automatically.

See SAPNOTE ODATA setup

- **Remove all the commented codes**

Otherwise, all the time it needs to load them from server to client and it is a waste of bandwidth usage.

- **Destroy complex controls after the usage.**

**- Stick on the MVC architecture**

Never go for OData calls from the view.

- **Use relative paths for web resources**

Otherwise you will struggle when you transport the application to quality or productive system.

- **Select only required columns in the OData calls.**

Lazy loading is by default implemented in some of the controls (like UI5 table). Make sure you implement the same in OData level also.

- **Implement all the possible features (like paging and filtering) in OData implementation**

In order to reduce the OData consumption from SAPUI5 side, always limit the number of records.

- **Use json format in OData calls**

In order to reduce file size transferred from the backend

The following rules are dedicated to the creation of a new own SAPUI5 content (e.g. new SAPUI5 control)

- **Do not use global JavaScript variables;**

Organize all global objects in an "sap.\*" namespace structure, or extend the jQuery.sap object. The method jQuery.sap.declare(sModuleName) assists in doing so.

Do not use undeclared variables. When using global variables introduced by other libraries, declare the use in a special global comment: /\*global JSZip, OpenAjax \*/.

- Do not access internal (private) members of other objects.

- **Do not use console.log()**

- **Use jQuery.sap.byId("<someId>") instead of jQuery("#<someId>") when <someId> is not a known string**

Certain characters in IDs need to be escaped for jQuery to work correctly.

- **Keep modifications of jQuery and other embedded Open Source to a minimum and document them clearly with the term *SAP modification*.**

Such modifications may not alter the standard behavior of the used library in a way that breaks other libraries

- **Bind component to Odata models.**

The Odata model manage all features of OData queries and in case of update (two-way binding), detect data changes.

- **Use Odata model V2 insteed of Odata model.**

V2 version has more options (use of batch request, json format by default…) and better performance.

- **Create Odata models on application initialization**

To ensure that the metadata of the service is loaded before first data request.

This can be done on the init function of component.js, or declare it on the manifest (available from version 1.32)

- **Do not change object properties directly**

Use model assigned to view to change the view objects, or model assigned to application to change objects of other views.

|  |  |
| --- | --- |
| **Bad** | **Good** |
| <Button id="delSub" press="onDelete" "/>    This.getView().byId("delSub").setVisible(true); | <Button id="delSub" press="onDelete" visible="{worklistView>/deleteVisible}"/>    this.getModel("worklistView").setProperty("/deleteVisible", true); |

- **Create page fragment for reusable object (Dialog, Search Help…)**

### 1.1.2 Code Formatting

In order to run successfully, ESLint check (used by SAP to check the well formatting of SAPUI5 source code) needs the following rules regarding code formatting:

- Add a semicolon after each statement, even if optional

- No spaces before and after round braces (function calls, function parameters)

- Use spaces after if/else/for/while/do/switch/try/catch/finally, around curly braces, around operators and after commas

- Opening curly brace (functions, for, if-else, switch) is on the same line

- Use "===" and "!==" instead of "==" and "!="; see the ESLint documentation for special cases where "==" is allowed

- The code should therefore look like this:

function outer(c, d) {

var e = c \* d;

if (e === 0) {

e++;

}

for (var i = 0; i < e; i++) {

// do nothing

}

function inner(a, b) {

return (e \* a) + b;

}

return inner(0, 1);

}

- You can use the Eclipse default settings for the JavaScript editor, but make sure tabs are used for indentation.

### 1.1.3 Documentation - JSDoc

SAPUI5 uses the JSDoc3 toolkit, as JavaDoc, to retrieve comments and tags to generate the documentation:

- Document the constructor with @class, @author, @since, …

- For subclasses, document the inheritance by using an @extends tag in their constructor doclet.

- Document at least public and protected methods with JSDoc, mark them as @public/@protected.

To document private methods with JSDoc, the tags @private has to be put. This is currently the default in SAPUI5, but not in JSDoc, so it is safer to explicitly specify it.

"Protected" is not clearly defined in a JavaScript environment, in SAPUI5 it means: Not for use by applications, but might be used even outside the same class or subclasses, but only in closely related classes.

- Document method parameters with type (in curly braces) and parameter name (in square brackets if optional).

- Use @namespace for static helper classes that only provide static methods.

## 1.2 CSS Coding Guidelines

The modification of style is allowed by SAPUI5 which uses CSS. Applications can adapt their styling by using their own CSS.

It is recommended be careful with CSS to be aligned with the future version of SAPUI5 and avoid any future problem of styling.

Pease respect Airbus CLAF, and use CSS provided by ICT if it exists

- Don't override control class styling directly

SAPUI5 does not guarantee the stability of style class names across versions. If the naming of style classes is changed in future versions, styling rules will no longer be applied to targeted elements. In addition, overriding control class styles directly might lead to style clashes when applications are run in shared runtime environments (like SAP Fiori launchpad).

Add your own namespaced classes instead.

|  |  |
| --- | --- |
| **Bad** | **Good** |
| .sapMInputBaseError {  font-weight: bold;  } | *Add a custom CSS class to the control in those situations where you want additional styling:*  oButton.addStyleClass("poaAppError");  *Then provide the style for this class:*  .poaAppError {  font-weight: bold;  } |

- Don't style DOM element names directly

Styling DOM elements directly will lead to unpredictable results, as SAPUI5 does not guarantee the stability of the inner-control DOM-tree over time. In addition, this might lead to styling clashes when applications run in shared runtime environments (like SAP Fiori launchpad) or when custom HTML is added. It is better to limit styling changes to specifically used CSS classes.

|  |  |
| --- | --- |
| **Bad** | **Good** |
| div {  width: 120px;  } | .myStyleClass {  width: 120px;  } |

- Don't use generated IDs in CSS selectors

SAPUI5 applications can create dynamic IDs for elements. Do not use these IDs as selectors in custom CSS as they can change over time. It is better to add and use CSS classes instead.

|  |  |
| --- | --- |
| **Bad** | **Good** |
| #\_\_view1\_\_button0 {  font-weight: bold;  } | *Add a style class as described above and then define the following:*  .myEmphasizedButton {  font-weight: bold;  } |

- Don't create selectors that are not namespaced

Custom selectors and CSS classes that are not namespaced might lead to style clashes in shared runtime environments like SAP Fiori launchpad, or when other JavaScript libraries are included that might use the same CSS class names.

|  |  |
| --- | --- |
| **Bad** | **Good** |
| .title {  font-weight: bold;  } | .poaAppTitle {  font-weight: bold;  } |

- Don't use hard-coded colors, font sizes and images if the application should be themable

Themability of applications relies on LESS calculations within the SAPUI5 theme CSS. Hard-coding colors, fonts and images in applications and custom controls means that these colors are not modified by theming, which leads to design issues or accessibility issues (for example, in the High Contrast Black (HCB) theme). You can use special CSS classes instead that are supplied by these LESS calculations.

|  |  |
| --- | --- |
| **Bad** | **Good** |
| .myCustomHTML {  color: #FFF;  background-color: blue;  } | Add the CSS classes sapThemeTextInverted and sapThemeHighlight-asBackgroundColor to your custom HTML element. |

- Don't use theming parameters for attributes they were not intended for

SAPUI5 applications come with a built-in set of parameters which are used for theme-dependent styling, mainly for colors. They are accessible using the sap.ui.core.theming.Parameters.get()API. These theme parameters have descriptive names, meaning that by looking at a parameter name, you can see the usage it has been defined for.

To ensure that you do not use combinations of theme colors which may clash after future theme changes, do not use background colors for fonts or vice versa, for example, and do not use border colors for anything else but borders.

|  |  |
| --- | --- |
| **Bad** | **Good** |
| var sColor = sap.ui.core.theming.Parameters.get("sapUiMediumBorder");  $(oSomeDomElement).css("background-color", sColor); | var sColor = sap.ui.core.theming.Parameters.get("sapUiMediumBorder");  $(oSomeDomElement).css("border-color", sColor); |

## 1.3 Compatibility Rules

It is important to take into account the change management during the development of a SAPUI5 application because the next version of SAPUI5 can provide new functionalities, change and remove some.

Reference documentation from SAP regarding compatibility rules and support and maintenance strategy :

ü <https://help.sap.com/saphelp_nw74/helpdata/en/91/f087396f4d1014b6dd926db0e91070/content.htm>

ü <https://help.sap.com/saphelp_uiaddon10/helpdata/en/91/ef99ad6f4d1014b6dd926db0e91070/content.htm>

### 1.3.1 API Evolution

The evolution (including fixes) of the public API can be included in 3 types of changes.

**Major release:** A new major version can introduce new APIs or make incompatible changes to existing APIs.

**Minor release:** A new minor version, can introduce new APIs but must not contain incompatible changes to any APIs.

**Patch release:** A new patch version only contains fixes to the existing implementation, but does not usually contain new features or incompatible API changes.

### 1.3.2 Compatible/Incompatible Changes

|  |  |
| --- | --- |
| **Compatible** | **Incompatible** |
| > Adding new libraries, controls, classes, properties, functions, or namespaces  > Generalizing properties, that is, moving properties up in the inheritance hierarchy  > Adding new values to enumeration types; this means that when dealing with enum properties, always be prepared to accept new values, for example, by implementing a "default" or "otherwise" path when reacting on enum values.  *The following can change in patch or minor release:*  > Manipulating HTML/CSS, for example via jQuery, control.addStyleClass, or directly via CSS  > Using or overriding "private" functions that are not part of the API Reference. Private functions are typically (but not always) prefixed with a preceding "\_".  > Open source libraries  > Log messages | > Removing an API (library, namespace, function, property, control, events, and so on)  > Renaming an API (library, namespace, function, property, control, events, and so on)  > Removing support for parameters  > Removing support for configuration entries  > Reducing the visibility of an API; this does not break JavaScript applications, but changes the contract  > Removing or reordering parameters in an API signature  > Reducing the accepted value range, for example, parameter of a function  > Broadening the value range of a return value (or property). Exception: enumerations  > Moving JavaScript artifacts (namespaces, functions, classes) between modules  > Replacing asserts with precondition checks  > Moving properties (and so on) down in the inheritance hierarchy  > Changing the name of enum values  > Changing defaults (properties, function parameters)  > Renaming or removing files |

### 1.3.3 Inheritance

SAP can modify by adding, removing, changing the source code of parent class in next versions so **it is strongly recommended to use sparingly the inheritance from SAPUI5 objects** (e.g. by calling sap.ui.extend on an existing control to add custom functionality) because it can make appear some strange behavior on your code in next version.

**SAP recommends testing very carefully after any update of SAPUI5 version and not replying the following:**

- Internal structures and methods that are not part of the public API

- Any internal logic and behavior of the object that is not reflected in the public API

- All rendering functionality of a control, including the HTML structure and CSS classes

- Existence of SAPUI5 lifecycle methods like init, exit, onBeforeRendering and onAfterRendering. When overriding an SAPUI5 lifecycle method, always check for existence of the method and call the super class implementation even though it is currently not implemented.

- Naming collisions with SAPUI5 structures and methods. SAPUI5 might introduce new API or internal structures at a later point in time that collide with your implementation. To avoid collisions, a custom prefix may be applied.

### 1.3.4 Deprecation

Some SAPUI5 objects, functions can be replaced by new ones and also declared old ones as deprecated in the SAPUI5 API.

**It is strongly recommended not using deprecated object to avoid any problem with an application. Use a deprecated means to modify the source code in the next upgrade of SAPUI5 libraries version, so to develop.**