Welcome to the DSAG´s UI5 Best Practice Guide

If you want to contribue, head over to the GitHub Repository: DSAG's UI5 best practice guide

Because the DSAG UI5 best practice guide is a living document 👨‍💻 - thriving on and with its' community 🥳

Table of Contents

XML Views

```xml
<mvc:View
    xmlns:mvc="sap.ui.core.mvc"
    xmlns:layout="sap.ui.layout"
    xmlns:form="sap.ui.layout.form"
    xmlns="sap.m">
</mvc:View>
```

One of the required namespaces can be defined as the default namespace (xmlns="..."). The control tags for this namespace do not need a prefix.

- Keep this un-prefixed control tag always to the very same library **sap.m** in all view XMLs of your project!
- Keep each named prefix constant throughout your project.

Deployment of UI5 apps to on premise systems

For a more streamlined development experience it is necessary not to switch out of your development environment. Also for developers who are not familiar with SAP technology you may want to prepare a simpler way of deploying their UI5 application to an SAP on premise system.

To accomplish this, you need to install the node.js module `ui5-nwabap-deployer-cli` with npm (either locally or globally).
For global installation execute

```
npm install ui5-nwabap-deployer-cli -g
```

Please refer to the documentation of the node.js module (see references for the link) for detailed installation instructions and further configuration options.

Create a file called `ui5deployrc` in your project root.

```
{
    "server": "<your on premise system url>",
    "client": "<your on premise mandate>",
    "package": "<refer to a valid package here. if you specify a local package (beginning with $) it won't get deployed>",
    "bspContainer": "<container name to store the files to>",
    "bspContainerText": "<description for the container>",
    "createTransport": true,
    "transportText": "<description of the transport request>",
    "transportUseLocked": true
}
```

These are the minimal values to let the node module create a new transport (if not exists) in your system. With this configuration it will reuse the transport if it is locked.

To authenticate with your on premise system it is discouraged to put your credentials within the `ui5deployrc` file, because you will include it in your VCS. Instead you may inject your username and password from environment variables of your system, while calling the node module from the commandline. This procedure enables you to even process your deployment from any CI/CD system.

For this simple bash script to work, you have to export `SAP_ONPREMISE_USER` and `SAP_ONPREMISE_PASSWORD` before executing. It is best to add your variable definitions to your `.bashrc` to have them set in every future bash session.

```
#!/bin/bash

rm -rf dist
npm run "build"

ui5-deployer deploy --user "${SAP_ONPREMISE_USER}" --pwd "${SAP_ONPREMISE_PASSWORD}"
```

Change this script accordingly if you use a different shell.

References

- ui5-task-nwabap-deployer (https://www.npmjs.com/package/ui5-task-nwabap-deployer)

Model Naming

The standard view model is very often unnamed in all the Walkthrough tutorials:

```
var oData = {
  recipient: {
    firstName: "John",
    lastName: "Doe",
  },
};
var oModel = new JSONModel(oData);
this.getView().setModel(oModel); // <-- No naming here!
```

Then, inside the corresponding view XML, model data may be referenced like so:

```
<Title text="Hello {/recipient/firstName} {/recipient/lastName}" />
```

In large projects with diverse models inside many views and controllers, its often very helpful to quickly lookup the usage of a certain model. In such cases it would really save a lot time if the model had a proper naming:

```
this.getView().setModel(oModel, "DisplayModel"); // <-- Example of a proper name
```

Then, inside the view XML, it would be:

```
<Title text="Hello {DisplayModel>/recipient/firstName}
{DisplayModel>/recipient/lastName}" />
```

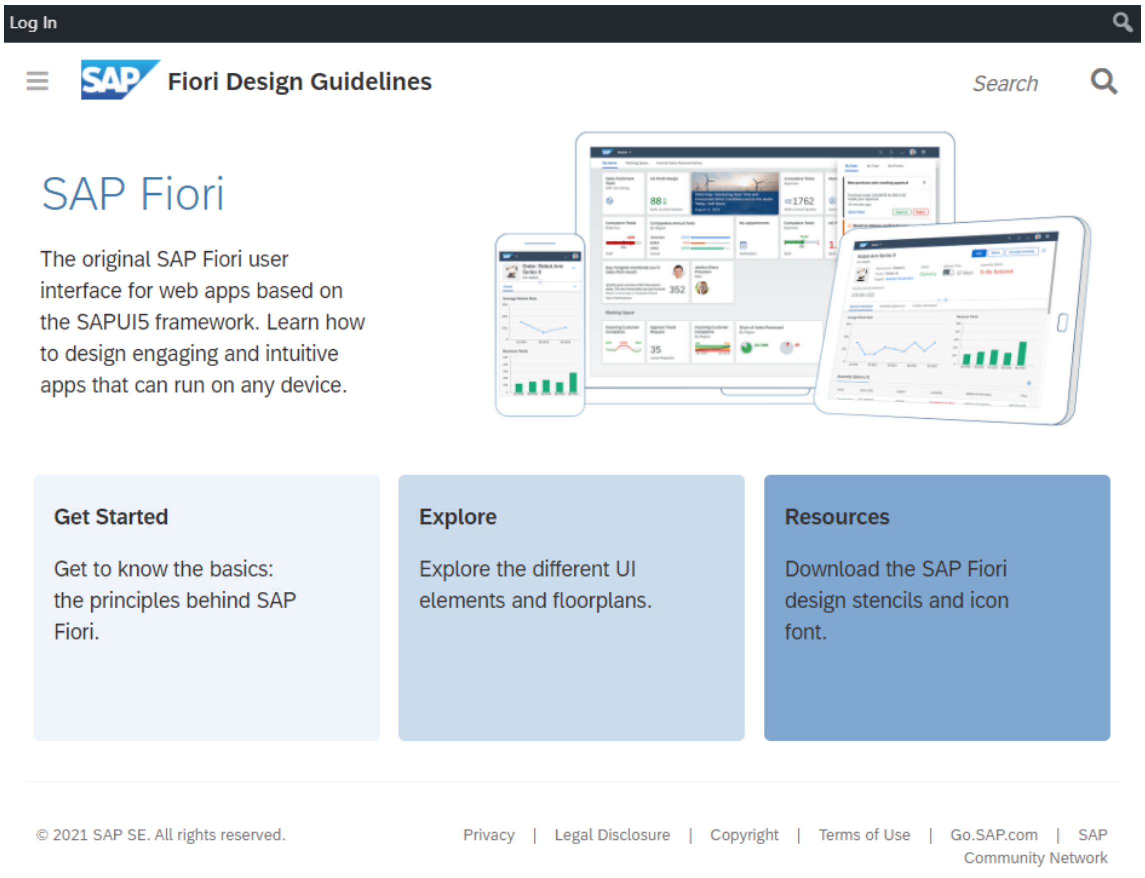
Overview Fiori Design Guidelines

Most classical SAP GUI business transactions aren't considered to be very user friendly. In contrast to that, users are getting more and more accustomed to intuitive and simple user interfaces in their daily life. This trend was especially driven by touch devices in the last years and leads to the expectation to get similar experiences in business environments.

Therefore, design and usability play major roles when developing Fiori Apps. Every UI5 developer should be interested in creating these beautiful, nicely to use Apps. Following the Fiori Design Guidelines can help to achieve this goal. And for that reason it is recommended to get familiar with these guidelines.

Furthermore, following the guidelines ensures coherence between UI5 Apps delivered by SAP and custom Apps, which is one of the five Fiori design principles (see below for more on that).

But what exactly is Fiori and what is the difference between UI5 and Fiori Apps?



SAP Fiori

The original SAP Fiori user interface for web apps based on the SAPUI5 framework. Learn how to design engaging and intuitive apps that can run on any device.

Get Started

Get to know the basics: the principles behind SAP Fiori.

Explore

Explore the different UI elements and floorplans.

Resources

Download the SAP Fiori design stencils and icon font.

© 2021 SAP SE. All rights reserved.

Privacy | Legal Disclosure | Copyright | Terms of Use | Go.SAP.com | SAP Community Network

Fiori, Fiori Apps and UI5 Apps

Fiori is a design system that describes how Apps should be designed.

A Fiori App is an App that is designed according to this Fiori design paradigm.

UI5 is a software framework, which is used to develop Apps. This means that not every UI5 App is an Fiori App at the same time. It is possible to build a UI5 App that doesn't strictly follow the Fiori Design Principles, but still uses UI5 "controls" (UI building blocks) that are styled in a Fiori-like design. UI5 is usually needed to create a Fiori App, because it includes many elements, which are the technical building blocks of the Fiori design principles. These building blocks can also be used in other frameworks with the UI5 Web Components.

Besides dedicated development in UI5, SAP offers another framework to develop Apps according to the Fiori Design Guidelines which is called Fiori Elements. It makes heavy use of annotations and metadata to generate the UI instead of the more development-focused approach of UI5.

This chapter describes some of the basics concepts, when it comes to designing Fiori Apps. For details it is strongly recommended to refer to the official Fiori Design Guidelines published by SAP.

Design Principles

Fiori is based on the following five design principles which should be considered in any design decision:

- ROLE-BASED
- DELIGHTFUL
- COHERENT
- SIMPLE
- ADAPTIVE

For a detailed definition and description we refer once again to the Fiori Design Guidelines.

Nevertheless, we would like to highlight the importance of a 'coherent' user experience. In the context of Fiori coherence means that the design, behavior and interaction of different Apps should be consistent.

It is important to ensure that users know what the consequences of their actions are. For example the same icon should be used in all Apps to delete items from a list. Inevitably this means that third-party developers should follow the design of standard SAP applications, because most self-developed UI5 Apps are deployed together with SAP standard applications on the same system.

Floorplans

An important concept in designing Fiori Apps are Floorplans, which are basically page layouts. Taking a closer look it can be noticed that many SAP Fiori Apps are built according to these Floorplans. The Design Guidelines consider them to be good layout choices for given use cases.

At the moment there are seven Floorplans:

- Analytical List Page
- List Report
- Worklist
- Overview Page
- Object Page
- Initial Page
- Wizard

For information on their detailed structure and use cases, please check the Fiori Design Guidelines. Keep in mind, that each of them has its own layout and should be used for specific use cases. In order to help making a decision on the most fitting layout, the guidelines provide some useful rules in the "When to Use" section of each Floorplan.

Other Content of the Design Guidelines and their Limitations

A few must-read articles can be found in the section 'General Concepts'. For instance this includes the article 'Action Placement' on where to place Buttons and other clickable elements. Another interesting example is the chapter 'Message Handling' describing how messages should be displayed.

In the section 'UI Elements' single UI5 elements are presented. Recommendations are given when and when not to use them. Furthermore, their behavior and possible interactions are described. This can be very helpful to decide which UI elements are appropriate in a given context.

Of course lots of other interesting articles can be found in the guidelines, which aren't mentioned here because the goal is to provide an overview, not to discuss each and every topic covered by the guidelines.

Concluding, the Fiori Design Guidelines have some gaps and will never be able to answer all design questions. To achieve a company wide uniform App design it is recommend to enhance the official design guidelines with company specific rules. This is especially important, if the guidelines don't provide an answers for a very specific design question.

While creating company specific rules the five design principles should always be taken into account.

Error Handling Overview

Error handling is probably the most important issue that is regularly neglected. It should be considered as a cornerstone of every app: It is the communication basis between the app and the user.

Two different kinds of errors

Errors to be dealt with can be classified into two categories:

1. Errors that affect the interface logic.
2. Errors affecting application logic or data processing in the backend.

The crucial difference between these two categories: The errors from category 1 are caught and output in the controller, the message text for them is stored in the i18n file. The errors from category 2, on the other hand, come from the backend. The message text is transmitted via the gateway service.

As far as category 1 is concerned, the corresponding logic must be mapped in the controller and messages must be output at the appropriate place.

For category 2 errors, the SAPUI5 framework provides the message manager. This parses messages from the service responses and triggers an event when new messages are received. The message manager can be used for messages from OData V2 services as well as for messages from OData V4 services.

Sample ErrorHandler.js file

It is recommended to outsource the error handling to a separate ErrorHandler.js file or library. In the section Sample Error Handler, a sample error handler is provided which outputs all messages parsed via the message handler and provides corresponding methods for displaying error-, warning-, information- and success-messages from the controller.

Error Handling Best Practices

The following points should be observed with regard to error handling:

Use a separate ErrorHandler.js file for error output

A separate ErrorHandler.js file should at least display all error messages of the integrated OData service. The event "error", what is thrown by manual constructed oData queries, should only be used for a further processing of the error, not for displaying the error.

Example: Don't do this:

```
this.getModel().read("/Entity('key'", {
    error: function () {
        sap.m.MessageBox.error("Error!", {
            styleClass: this.getOwnerComponent().getContentDensityClass()
        });
        this.getView().setBusy(false);
    }
});
```

Instead do that in Component.js file:

```
this._oErrorHandler = new ErrorHandler(this);
```

And only implement further processing in error function:

```
this.getModel().read("/Entity('key')", {
    error: function () {
        this.getView().setBusy(false);
    }
});
```

A sample for a separate error handler file can be found in the section Sample Error Handler.

Consider the SAP Fiori Guidelines

The SAP Fiori Guidelines deal with error handling in several articles. The article Error, Warning and Info Messages serves as the basis.

An important first step is the selection of the right control: Error messages, warnings and information to be confirmed are displayed in a Message Box. Success messages should be displayed in an automatically disappearing Message Toast. Don't display an error message in a message toast!

If you need to display multiple messages, use a Message View.

Make sure that message boxes and messages views use the correct content density class. When calling a message box, the style class must always be transferred. If this does not happen, the controls in the view and in the message box may be in different style classes, which does not look very professional.

Send application-related messages from the frontend and technical or business messages from the backend

It should be noted that verifications should be made at the right place and thus error messages should be thrown at the appropriate place.

Error messages from the SAPUI5 application should only concern the application logic and not contain any business logic. Example for a message from the frontend: Before a form is submitted, it is checked whether all mandatory fields are filled and the postcode entered is in the correct format (five digits). If this is not the case, an error message is sent from the controller and no request is sent against the backend.

Subject-specific verifications, on the other hand, should take place in the backend and subject-specific error messages should be issued accordingly via the OData service. Example for a message from the backend: When data is entered via a form, a check is made to see whether the postcode entered exists. If the postcode does not exist, an error message will be thrown by the service.

Also display several error messages at once

An OData service can send a response with several error messages as a result of a request. It is also possible that several requests are made in parallel and that error messages from two different requests are sent to the frontend app. Likewise, it is also possible that an error message is displayed originating from the controller and at that moment an error message from the backend reaches the frontend app. To summarise briefly: It is possible that several relevant messages exist at the same time.

In the literature or in instructions on the internet, similar variants like the following for the output of service errors can be found (don't use that):

```
_showServiceError: function(sDetails) {
    if (this._bMessageOpen){
        return;
    }
    this._bMessageOpen = true;
    MessageBox.error("An ErrorOccurred", {
        details: sDetails,
        actions: [MessageBox.Action.CLOSE],
        onClose: function(){
            this._bMessageOpen = false;
        }.bind(this)
    });
}
```

If you need to display multiple messages, use a Message View.

Message boxes and message views can be a stumbling block when it comes to setting the correct content density class. When calling a message box, the content density class must always be transferred. If this does not happen, the controls in the view and in the message box may be in different style classes, which does not look very professional.

Send application related messages from the frontend and technical or business messages from the backend

It should be noted that verifications should be made at the right place and thus error messages should be thrown at the appropriate place.

Error messages from the SAPUI5 application should only concern the application logic and not contain any business logic. Example for a message from the frontend: Before a form is submitted, it is checked whether all mandatory fields are filled and the postcode entered is in the correct format (five digits). If this is not the case, an error message is sent from the controller and no request is sent against the backend.

Subject-specific verifications, on the other hand, should take place in the backend and subject-specific error messages should be issued accordingly via the OData service. Example for a message from the backend: When data is entered via a form, a check is made to see whether the postcode entered exists. If the postcode does not exist, an error message will be thrown by the service.

display several error messages at once

An OData service can send a response with several error messages as a result of a request. It is also possible that several requests are made in parallel and that error messages from two different requests are sent to the frontend app. Last but not least, it is also possible that an error message is displayed from the controller and at that moment an error message from the backend reaches the frontend app. To summarise briefly: It is possible that several relevant messages exist at the same time.

In the literature or in instructions on the internet, the following variant for the output of service errors is often found (don't use that):

```
_showServiceError: function(sDetails) {
    if (this._bMessageOpen){
        return;
    }
    this._bMessageOpen = true;
    MessageBox.error("An ErrorOccurred", {
        details: sDetails,
        actions: [MessageBox.Action.CLOSE],
        onClose: function(){
            this._bMessageOpen = false;
        }.bind(this)
    });
}
```

This form of error output ensures that only one error is displayed and all other errors are ignored. Instead, in the case of multiple errors, you should use a message view instead of the message box.

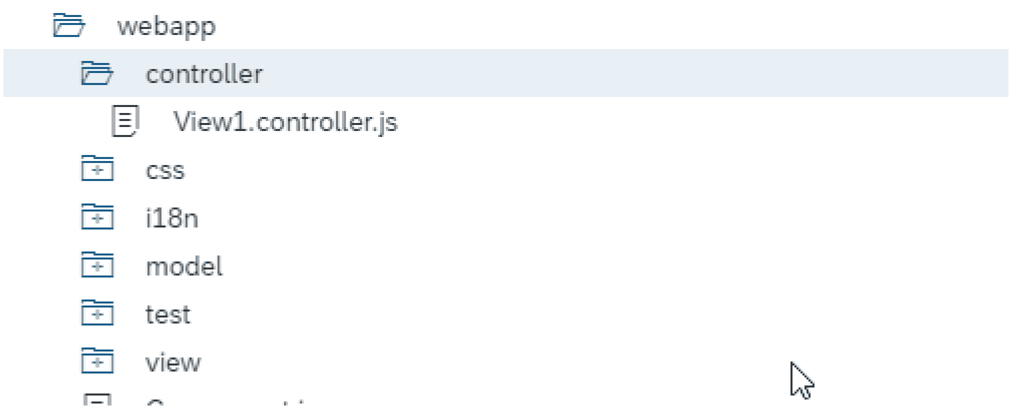
An example for a smarter implementation can be found in the [Sample Error Handler](#).

Sample Error Handler

At this point, a sample error handler is provided that takes into account the best practices described in this guide. If the error handler is correctly integrated, messages from an OData V2 or Odata V4 service are automatically output in the designated control: Error messages in a Message Box, success messages in a Message Toast and a collection of messages in a Message View. For application-related messages from the controller, the methods displayError, displayWarning, displayInformation and displaySuccess are provided.

Setup

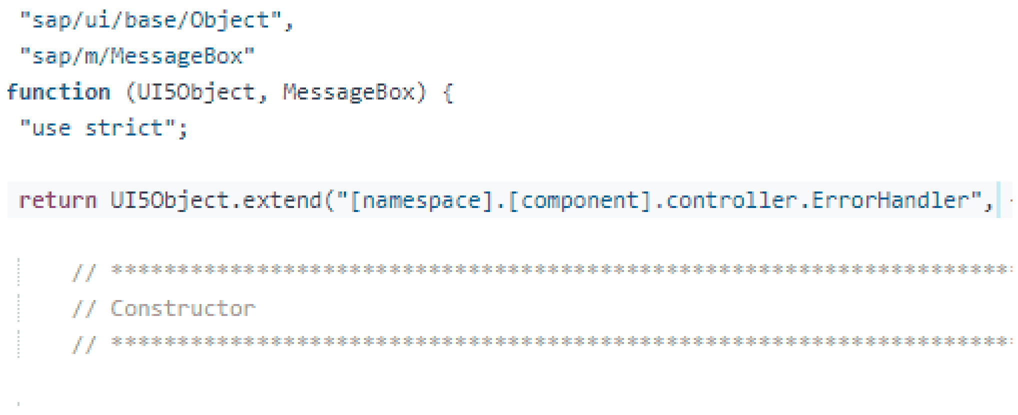
1. Add a new file called ErrorHandler.js in folder webapp/controller.



2. Insert the source code of the error handler. You can download the source code or copy it from the collapsed text.
[Error Handler (English comments)](#)
[Error Handler (German comments)](#)

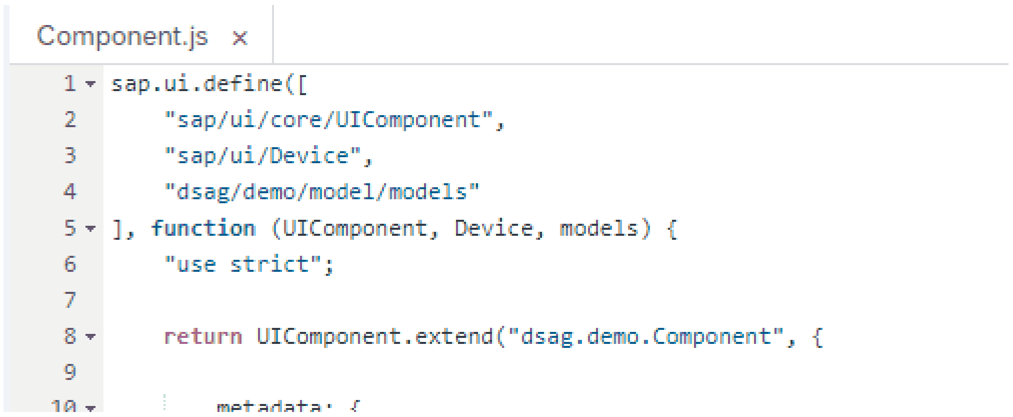
▶ Open Error Handler Source Code

3. In UI5Object.extend line (line 32), adjust the namespace and the component name of your SAPUI5 app.



```
 "sap/ui/base/Object",
 "sap/m/MessageBox"
function (UI5Object, MessageBox) {
 "use strict";

return UI5Object.extend("[namespace].[component].controller.ErrorHandler",

    // ********************************************************************
    // Constructor
    // ********************************************************************
```

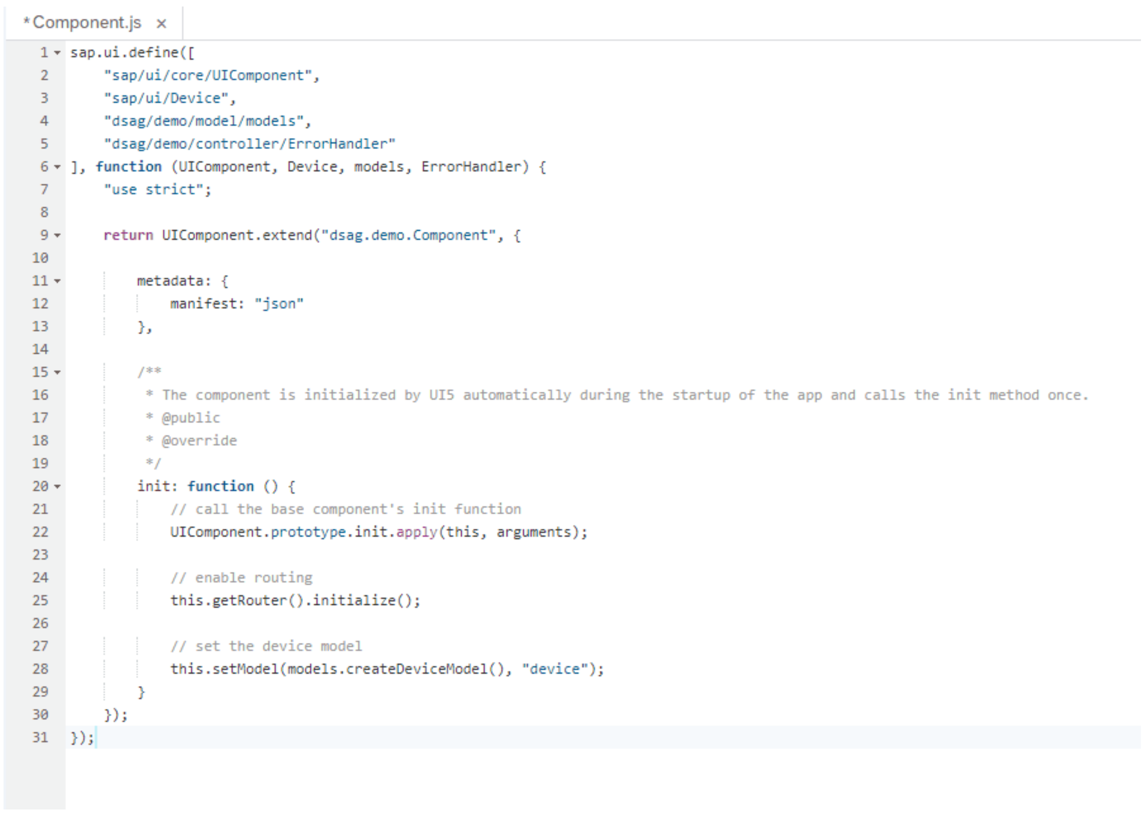
4. Include the error handler file in Component.js by declaring it as a required resource in the document header.



```
sap.ui.define([
	"sap/ui/core/UIComponent",
	"sap/ui/Device",
	"dsag/demo/model/models"
], function (UIComponent, Device, models) {
	"use strict";

	return UIComponent.extend("dsag.demo.Component", {

		metadata: {
```

5. Initialise the error handler in the onInit method of Component.js by inserting the following code line:

~~~
```js
this._oErrorHandler = new ErrorHandler(this);
```
~~~



```
sap.ui.define([
	"sap/ui/core/UIComponent",
	"sap/ui/Device",
	"dsag/demo/model/models",
	"dsag/demo/controller/ErrorHandler"
], function (UIComponent, Device, models, ErrorHandler) {
	"use strict";

	return UIComponent.extend("dsag.demo.Component", {

		metadata: {
			manifest: "json"
		},

		/**
		 * The component is initialized by UI5 automatically during the startup of the app and calls the init method once.
		 * @public
		 * @override
		 */
		init: function () {
			// call the base component's init function
			UIComponent.prototype.init.apply(this, arguments);

			// enable routing
			this.getRouter().initialize();

			// set the device model
			this.setModel(models.createDeviceModel(), "device");
		}
	});
});
```

6. Make sure that the Component.js contains the method getContentDensityClass for the style class setting.

▶ If missing, copy and paste it from here.

Further links

This overview offers further links on the topic of error handling.

SAP Fiori Guidelines: Message Handling

The SAP Fiori Guidelines article on message handling describes the things to consider in terms of UX.

Official SAPUI5 Documentation: Error, Warning and Info Messages

The official SAPUI5 documentation article describes how to use the SAPUI5 Message Manager.

Message Handling

Recommended guidelines for message handling from official SAPUI5 documentation.

Fiori Elements vs. Freestyle

There are two ways to develop your own Fiori applications: Either you start with a complete self-development ("freestyle") or you use the SAP Fiori Elements Framework.

The Fiori Elements Framework offers the possibility for various floorplans to create an app according to this pattern on the basis of an OData service without any own source code. In this way, an app can be created with very little effort, but can only be extended with own functionalities to a limited extent.

SAP offers a detailed guide, including a decision tree, for deciding when to develop a Fiori Elements app and when to develop a Freestyle app:
Usage Guide: When to use Fiori Elements

Overview i18n Overview

Many companies use SAP systems and applications and many different countries. To ensure easy translation, the "i18n" (Internationalization) process is used. The app is not always recreated, but only the texts are translated and replaced.

Language Tags

To identify the individual languages, so-called "Language Tags" are used which are defined by BCP-47. These language tags are stored in the SAP system in table T002 for all stored languages. These correspond to the "ISO 639 alpha-2 language code". The UI5 application then converts the determined language key into a BCP-47 tag.

As a best practice, it should be defined that the BCP-47 language code should be kept as short as possible. When creating it, you should avoid regions, script or subtags as far as possible - unless you add important and helpful information.

Good example

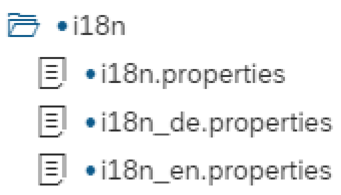
- `i18n.properties`
- `i18n_DE.properties`

Bad example

- `i18n.properties`
- `i18n_DE.properties`
- `i18n_DE_DE.properties`
- `i18n_DE_CH.properties`
- `i18n_DE_AT.properties`

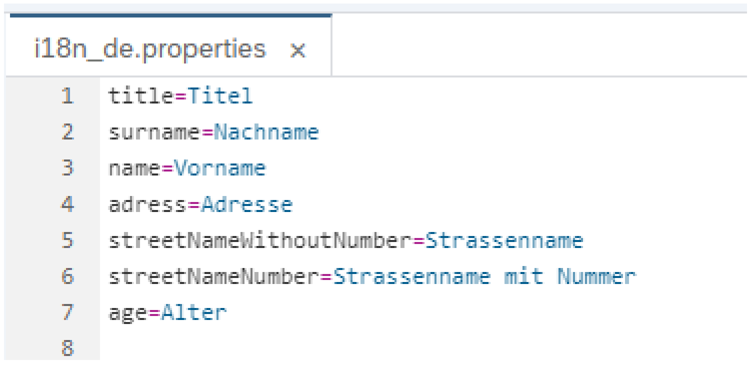
Files in the UI5 App

The individual translations are stored in the i18n folder with the prefix i18n_ and the corresponding language tag. In addition, a default file is defined which is automatically called if no language is specified.


•i18n
•i18n.properties
•i18n_de.properties
•i18n_en.properties

UI5 i18n Files and Folder

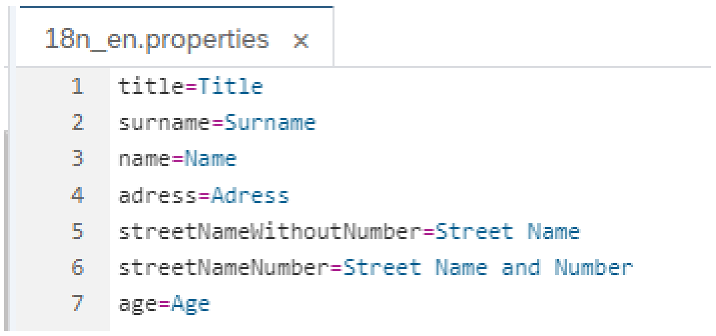
Here are two simple examples, for a German and an English translation. The "keys" are always identical and will be retrieved later by App based on them. The keys are written in (lower) camelCase.



i18n_de.properties ×

```
title=Titel
surname=Nachname
name=Vorname
adress=Adresse
streetNameWithoutNumber=Strassenname
streetNameNumber=Strassenname mit Nummer
age=Alter

```

i18n Simple Example DE File

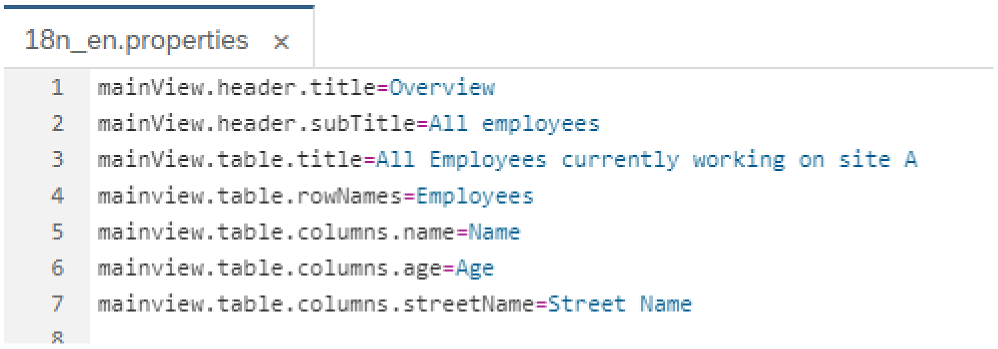


18n_en.properties ×

```
title=Title
surname=Surname
name=Name
adress=Adress
streetNameWithoutNumber=Street Name
streetNameNumber=Street Name and Number
age=Age
```

i18n Simple Example EN File

It is also possible to connect the i18n keys with a point to get a better overview. You can still use here camelCase.



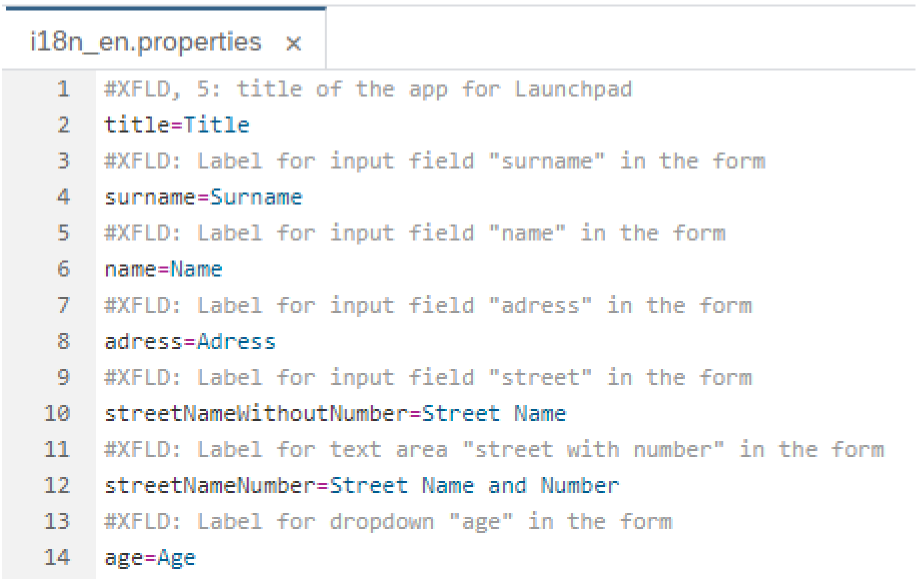
18n_en.properties ×

```
mainView.header.title=Overview
mainView.header.subTitle=All employees
mainView.table.title=All Employees currently working on site A
mainview.table.rowNames=Employees
mainview.table.columns.name=Name
mainview.table.columns.age=Age
mainview.table.columns.streetName=Street Name

```

i18n example with dot notation

Annotations in i18n

Annotations in the i18n files are helpful to understand how the i18n texts are used in the app. Annotations are inserted as comments and include a text type classification, an optional length restriction and a freetext explanation.



```
#XFLD, 5: title of the app for Launchpad
title=Title
#XFLD: Label for input field "surname" in the form
surname=Surname
#XFLD: Label for input field "name" in the form
name=Name
#XFLD: Label for input field "adress" in the form
adress=Adress
#XFLD: Label for input field "street" in the form
streetNameWithoutNumber=Street Name
#XFLD: Label for text area "street with number" in the form
streetNameNumber=Street Name and Number
#XFLD: Label for dropdown "age" in the form
age=Age
```

i18n example with annotations

A list of the possible text classification can be found here: [Annotations in Translation Files]()

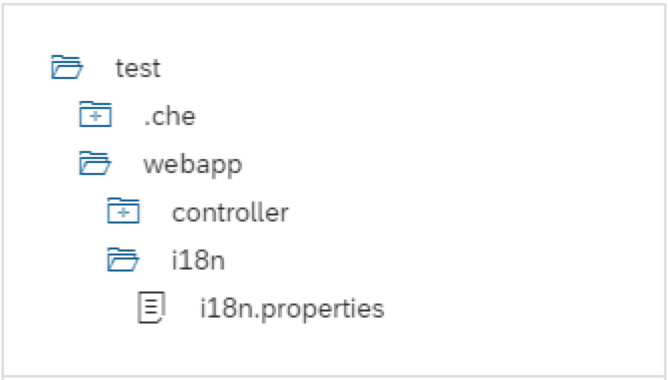
Usage of i18n in UI5 Apps

In general, the handling of translated texts is very well documented in the SAP documentation: Documentation - Walktrough - Step 8: Translatable Texts

Setup

1. create a new folder with the name "i18n
2. create at least one file with the name "i18n.properties

   The folder should be created in the "webapp" folder



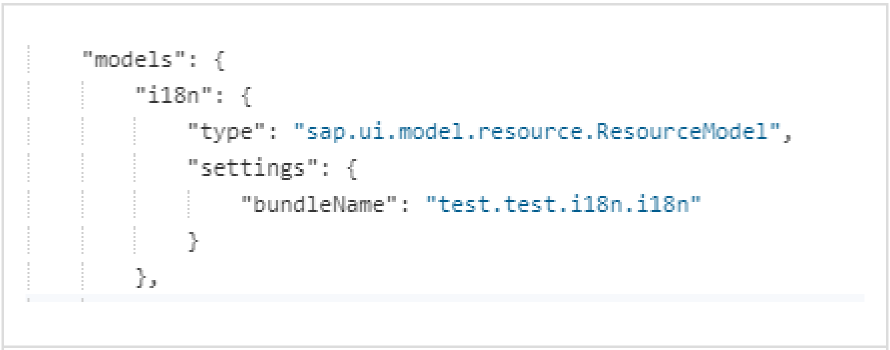
```
test
  .che
  webapp
    controller
    i18n
      i18n.properties
```

Folder Structure and Init File for i18n

3. configure the manifest.json

   In our manifest.json we have to configure the above created i18n file under models area within UI5 section as shown below. This code is placed in "sap.ui5" --> "models"

```
"i18n": {
        "type": "sap.ui.model.resource.ResourceModel",
        "settings": {
            "bundleName": "your.namespace.i18n.i18n"
        }
```



```
"models": {
    "i18n": {
        "type": "sap.ui.model.resource.ResourceModel",
        "settings": {
            "bundleName": "test.test.i18n.i18n"
        }
    },
```

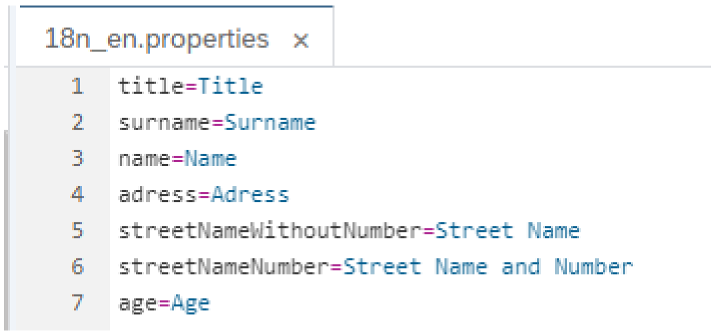
Folder Structure and Init File for i18n

Usage in XML Views

Since we have defined a model in manifest.json, we access the individual keys in the XML view as with any other model.

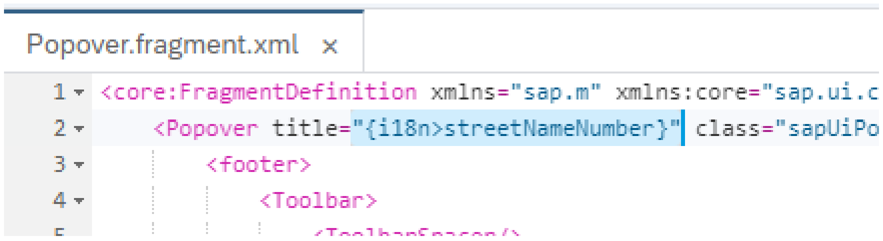
The i18n model is referenced first in curly brackets and then the key: `{i18n>streetNameNumber}`

We take as an example the already previously defined translations



```
title=Title
surname=Surname
name=Name
adress=Adress
streetNameWithoutNumber=Street Name
streetNameNumber=Street Name and Number
age=Age
```

i18n Simple Example EN File



```
<core:FragmentDefinition xmlns="sap.m" xmlns:core="sap.ui.c
    <Popover title="{i18n>streetNameNumber}" class="sapUiPo
        <footer>
            <Toolbar>
```

i18n example XML View

Usage in Controllers

It is good practice to create a BaseController that has several methods already implemented. One of them is to access the i18n model directly in the controllers:

```
/**
    * Getter for the resource bundle.
    * @public
    * @returns {sap.ui.model.resource.ResourceModel} the resourceModel of the
component
*/
getResourceBundle : function () {
    return this.getOwnerComponent().getModel("i18n").getResourceBundle();
}
```

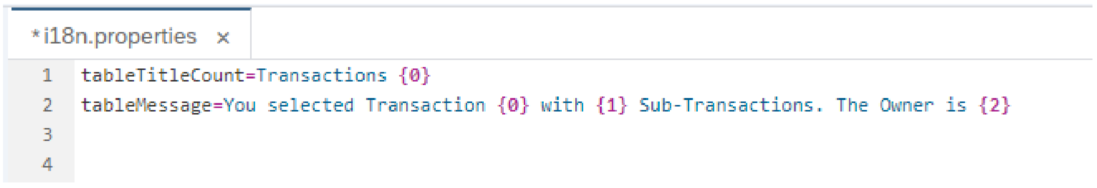
Now it is possible to access the texts in the inherited controllers as follows:

```
var sTitleText = this.getResourceBundle().getText("title");
```

Usage of placeholders

It is good practice to mark variables in texts with placeholders and not concatenate them directly in JavaScript.

Variables are set in the strings in the i18n.properties files. Numbers, starting with 0, are placed in curly brackets. In case of multiple variables the numbers are counted up.


*i18n.properties ×

```
tableTitleCount=Transactions {0}
tableMessage=You selected Transaction {0} with {1} Sub-Transactions. The Owner is {2}
```

Usage of Placeholder in i18n

Good example

```
    var sTranslatedText = this.getResourceBundle().getText("worklistTitle",
[iCounterVariable]);
```

Bad example

```
    var sTranslatedText = this.getResourceBundle().getText("worklistTitlePart1")
                                + ": "
                                +  iCounterVariable
                                +
this.getResourceBundle().getText("worklistTitlePart2");
```

The use of placeholders directly in XML views is described in Advanved Features in i18n

Advanced Features in i18n

Placeholder in XML Views

It is also possible to use the parameters directly in the XML view. There are several things to be aware of here. When `getText("title", [parameter])` is called in the controller to the RessourceModel, the method `sap/base/strings/formatMessage` is used to implement the parameter. Since formatter can be used in certain UI5 controls, we can use this method in XML view as well. However, in order to call the method in the control, it must be declared beforehand.

Since version `1.69` this is possible directly in the XML view, before that the declaration must still be made in the controller. The Documentation for this is found here: Require Modules in XML View and Fragment

Usage in 1.69 and above

Instead of a custom formatter, the standard `formatMessage` is used here.

The `xmlns="sap.m"` and `xmlns:core="sap.ui.core"` can also be decleared in the mcv:View directly.

```
<Title xmlns="sap.m" xmlns:core="sap.ui.core"
  core:require="{ formatMessage: 'sap/base/strings/formatMessage' }"
  text="{
    parts: [
      'i18n>testString',
      'i18n>testString2'
    ],
    formatter: 'formatMessage'
  }"
/>
```

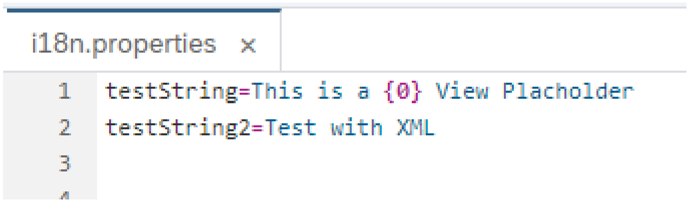
Usage in 1.68 and below

Here you have to point the formatter to the Controller where you point to the required `formatMessage` module.

```
<Title text="{
  parts: [
    'i18n>testString',
    'i18n>testString2'
  ],
  formatter: '.formatMessage'
}"/>
```

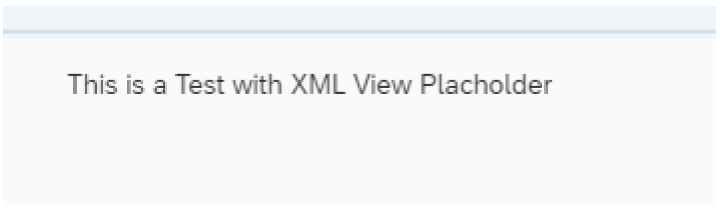
```
sap.ui.define([
  "sap/ui/core/mvc/Controller",
  "sap/base/strings/formatMessage",
  // ...
], function(Controller, formatMessage/*, ...*/) {
  "use strict";

  return Controller.extend("...", {
    formatMessage: formatMessage,
    // ...
  });
})
```

Result



```
testString=This is a {0} View Placholder
testString2=Test with XML
```

Usage of Placeholder in i18n in XML Views



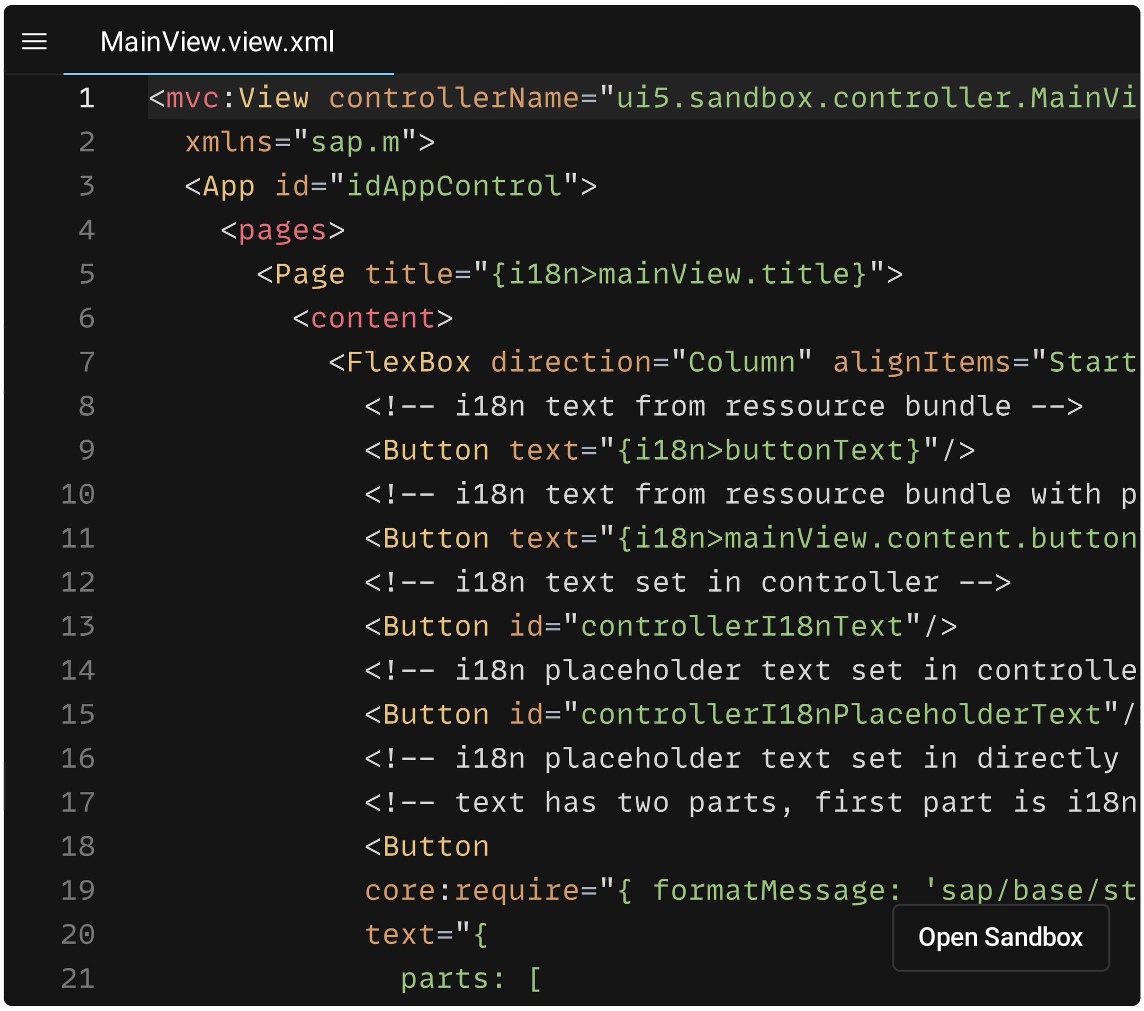
This is a Test with XML View Placholder

Screenshot of Placeholder in i18n in XML Views

Ressoures:

- [Stackoverflow Comment]
- [Openui5 Issue with Change to use declaration in XML View]

CodeSandbox Sample



```
<mvc:View controllerName="ui5.sandbox.controller.MainVi
  xmlns="sap.m">
  <App id="idAppControl">
    <pages>
      <Page title="{i18n>mainView.title}">
        <content>
          <FlexBox direction="Column" alignItems="Start
            <!-- i18n text from ressource bundle -->
            <Button text="{i18n>buttonText}"/>
            <!-- i18n text from ressource bundle with p
            <Button text="{i18n>mainView.content.button
            <!-- i18n text set in controller -->
            <Button id="controllerI18nText"/>
            <!-- i18n placeholder text set in controlle
            <Button id="controllerI18nPlaceholderText"/
            <!-- i18n placeholder text set in directly
            <!-- text has two parts, first part is i18n
            <Button
            core:require="{ formatMessage: 'sap/base/st
            text="{
              parts: [
```

`ui5 tooling` tasks

At the time of this writing, there are a couple of open source `ui5-tooling` tasks available that help with translation/internationlization efforts:

- `ui5-task-18ncheck` : checking for missing translations in `i18n*` -files
- `ui5-task-translationhub` : automated upload, translation and download of `i18n*` -files

Model View Controller

Some general thoughts about the MVC concept:

- SAPUI5 MVC concept
- MVC or MVVC ?

Use base controllers

Keep commonly used methods such as

- getResourceBundle
- getRouter
- getModel
- setModel
- onNavBack
- *or whatever method or property is being re-used in other controllers in the application*

inside a base controller:

```
|-+ webapp
  |-+ controller
    |-- BaseController.js
```

webapp/controller/BaseController.js:

```
sap.ui.define([
  "sap/ui/core/mvc/Controller",
  "sap/ui/core/routing/History",
  "sap/ui/core/UIComponent"
], function(Controller, History, UIComponent) {

  "use strict";
  return Controller.extend("com.myCompany.myProduct.controller.BaseController", {

    getRouter : function () {
      return UIComponent.getRouterFor(this);
    },
    ...
    onNavBack: function () {
      var oHistory, sPreviousHash;
      oHistory = History.getInstance();
      sPreviousHash = oHistory.getPreviousHash();
      if (sPreviousHash !== undefined) {
        window.history.go(-1);
      } else {
        this.getRouter().navTo("appHome", {}, true /*no history*/);
      }
    }

  });
});
```

All view controllers should use the *BaseController.js* as their parent controller like so:

webapp/controller/App.controller.js:

```
sap.ui.define([
  "com/myCompany/myProduct/controller/BaseController"
], function (Controller) {
  "use strict";
  return Controller.extend("com.myCompany.myProduct.controller.App", {
    onInit: function () {
      ...
    }
  });
});
```

Make use of Data Binding

Instead of manipulating the behavior of your control

```
<Title text="Hello {/recipient/firstName} {/recipient/lastName}" id="myControl123"
/>
```

by its id, like so,

```
var oTitle = this.getView().byId("myControl123");
oTitle.setText("Hello John Doe");
```

better make use of data binding

```
<Title text="Hello {DisplayModel>/recipient/firstName}
{DisplayModel>/recipient/lastName}"/>
```

and work on the data model:

```
var oDisplayModel = this.getView().getModel("DisplayModel");
oDisplayModel.setProperty("/firstName", "John");
oDisplayModel.setProperty("/lastName", "Doe");
```

This pays off especially when working with dynamic arrays being visualized by lists or tables! You feed the model with updated data and data binding will re-render the controls for you automatically.

One-Way-Binding vs Two-Way-Binding

See Step 4 and Step 5 of the *Data Binding Tutorial* to see the practical difference between the two binding modes!

- One-way binding means a binding from the model to the view. Any value changes in the model update all corresponding bindings and the view.
- Two-way binding means a binding from the model to the view and from the view to the model. Any changes in the model or the view fire events that automatically trigger updates of all corresponding bindings and both the view and the model.
- Two-way binding is currently only supported for property bindings.
- When using formatter functions, the binding is automatically switched to "one-way". So you can’t use a formatter function for "two-way" scenarios, but you can use Data Types.

Naming Conventions

Architecture

OData is a protocol for the creation and consumption of RESTful APIs. Thus, as common practices of REST, OData builds on HTTP, AtomPub, and JSON using URIs to address and access data feed resources.

Resource Identification

OData uses URIs to identify resources. For every OData service whose service root is abbreviated as http://host/service/, the following fixed resources can be found:

The Service Document

The service document lists entity sets, functions, and singletons that can be retrieved. Clients can use the service document to navigate the model in a hypermedia-driven fashion.

The service document is available at the "doc root"/service-root: http://host/service/.

The Metadata Document

The metadata document describes the types, sets, functions and actions understood by the OData service. Clients can use the metadata document to understand how to query and interact with entities in the service.

The metadata document is available at http://host/service/$metadata.

Resource operation

OData uses the HTTP verbs to indicate the operations on the resources.

- `GET` : Get the resource (a collection of entities, a single entity, a structural property, a navigation property, a stream, etc.).
- `POST` : Create a new resource.
- `PUT` : Update an existing resource by replacing it with a complete instance.
- `PATCH` : Update an existing resource by replacing part of its properties with a partial instance.
- `DELETE` : Remove the resource.

Major Differences

These are the major differences between OData V2 and OData V4:

Metadata control

In OData V4, the JSON data format now allows to control the amount of metadata that is returned in query responses. There are three levels of metadata supported (see details):

- `full` : The response contains all the metadata needed to describe the response.
- `minimal` : The response metadata references the metadata document. Information in the metadata document is not repeated in the response.
- `none` : The response contains no metadata. The application must understand the response structure.

Search capability

OData V4 adds a new flexible search capability, `$search` . The search feature allows you to query a collection for entities that match a specified search expression. Unlike the existing filter capability, which allows a query to specify that a specific property or properties match certain criteria, the search feature can apply the search expression to any of the properties of an entity.

Enhanced expand

The `$expand` system query option has been enhanced in OData V4. This feature specifies the related resources to be included in line with retrieved resources. In OData 2, if a single value navigation property is expanded, you get all the properties of the entity if it was a single value navigation property. And if a collection navigation property is expanded, you get all of the entities in the collection and all of the properties of those entities. In OData V4, you can now refine the results using the `$select` , `*` , `$filter` and `$top` operations.

Example: Get all teams that have at least one employee who is older than 42

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/Filter",
    "sap/ui/model/FilterOperator"
], function(Controller, Filter, FilterOperator) {
    "use strict";
    return Controller.extend("dsag.filter.Example", {
        // ...
        filterTeams: function() {
            oTeamsBinding.filter(
                new Filter({
                    path : "EMPLOYEES",
                    operator : FilterOperator.Any,
                    variable : "employee",
                    condition : new Filter("employee/AGE", FilterOperator.GT, 42)
                });
            );
        }
    });
});
```

The resulting request would be: `http://host/service/TEAMS?$filter=EMPLOYEES/any(employee:employee/AGE gt 42)`

Counting

$count replaces `$inlinecount` in OData V4. `$count` has been enhanced to be used with `$filter`, `$expand` and `$orderby` options.

Datatypes

Changes in support for data types in OData V4:

- `Edm.DateTime` has been deprecated. The lack of timezone information in OData 2 causes significant problems. Use `Edm.DateTimeOffset` instead.
- `Edm.Time` has been replaced with `Edm.Duration` and `Edm.TimeOfDay` to make it clear whether it is duration of a specific time of day.
- `Edm.Date` has been added as there was no way to express just a date in OData 2.
- `Edm.Float` has been eliminated.

Data access methods

The UI5 OData V4 model does not support the methods `getData`, `getObject`, `getOriginalProperty`, `getProperty`. For data access, use the context API instead of methods on the model.

Batch methods

The UI5 OData V4 model does not support the methods `getChangeBatchGroups`, `getChangeGroups`, `getDeferredGroups`, `setChangeBatchGroups`, `setChangeGroups`, `setDeferredBatchGroups`, `setDeferredGroups`, `setUseBatch` (and corresponding model construction parameters). Batch groups are solely defined via binding parameters with the corresponding parameters on the model as default. Application groups are by default deferred; there is no need to set or get deferred groups. You just need the `submitBatch` method on the model to control execution of the batch. You can use the predefined batch group `$direct` to switch off batch either for the complete model or for a specific binding (only possible for the complete model in V2).

OData operations executed via binding

The UI5 OData V4 model does not support the method `callFunction`. Use an operation binding instead.

Example:

View:

```
<Form id="getNextAvailableItem" binding="{/GetNextAvailableItem(...)}">
    <Label text="Description"/>
    <Text text="{Description}"/>
    <Button text="Call the function" press="onGetNextAvailableItem"/>
</Form>
```

Controller:

```
onGetNextAvailableItem : function (oEvent) {
    this.getView().byId("getNextAvailableItem").getObjectBinding().execute();
}
```

No CRUD methods on model

The UI5 OData V4 model does not support the methods `create` , `read` , `remove` , `update` . `read` , `update` , `create` and `remove` operations are available implicitly via the bindings, so that changes are bound to controls. It is not possible to trigger requests for specific OData URLs.

No metadata access via model

The UI5 OData V4 model does not support methods `getServiceAnnotations` , `getServiceMetadata` , `refreshMetadata` as well as methods corresponding to the events `metadataFailed` , `metadataLoaded` . Metadata is only accessed via `ODataMetaModel` . Metadata is only loaded when needed (e.g. for type detection or to compute URLs for write requests); the corresponding methods on the `v4.ODataMetaModel` use promises instead of events.

AnnotationHelper

`sap.ui.model.odata.AnnotationHelper` is not supported for OData V4.

NOTE

Due to the limited feature scope of this version of the SAPUI5 OData V4 model, check that all required features are in place before developing applications. Double check the detailed documentation of the features, as certain parts of a feature may be missing although you might expect these parts as given. Some controls might not work due to small incompatibilities compared to `sap.ui.model.odata.(v2.)ODataModel` , or due to missing features in the model (like tree binding).

Naming Conventions

OData Term	Rule	Good Example	Bad Example
General	Use Camel case	SalesDocument	SALESDOCUMENT
General	No Underscores	PurchaseOrder	Purchase_Order
General	Use English Names	Order	Auftrag
General	No SAP technical names	CompanyCode	BUKRS
Entity Names	Only Nouns	CostCenter	CostCenterF4
Entity Names	Only Singulars	PurchaseOrder	PurchaseOrderList
Entity Names	No Operation Names	SalesOrder	CreateSalesOrder
Entity Sets	Use Plural of Entity Name or add "Set"	PurchaseOrders, PurchaseOrderSet	PurchaseOrder
Navigations	Name as Entity name if the target cardinality is 1	`/OrderHead('1')/OrderItem('1')`	`/OrderHead('1')/Header_Item('1')`
Navigations	Same as EntitySet name if the target cardinality is M	`/OrderHead('1')/OrderItems`	`/OrderHead('1')/Header_Item`
Function Imports (V2) / Actions (V4)	Use Clear Names	BlockSalesOrder	Block

Dos and Don'ts

This section contains useful dos and don'ts for developers programming SAPUI5 applications using OData and optimizing its performance.

Dos

* Do think in REST terms

If you find yourself inventing function imports such as CreateX, ReadX, GetX, ChangeX, UpdateX, or DeleteX, you should make the entity type X part of your model. `CRUD` is covered by `GET`, `PUT`, and `DELETE` to entities and `POST` to entity sets. .

Function imports may be used for operations other than `CRUD`, such as Approve, Reject, Reserve, and Cancel. However, you might find that a reservation entity that you can create, update, and delete is better suited.

* Do return completely filled entries in response to GET ~/EntitySet

Think:

```
SELECT * FROM EntitySet WHERE ($filter) ORDER BY ($orderby) UP TO ($top) ROWS
```

if you see the following:

```
GET ~/EntitySet?$filter=...&$orderby=...&$top=...
```

OData does not distinguish GetList from GetDetail. OData clients expect to obtain all information from reading an entity set. Take a simple UI5 master/detail application as an example. The OData model retrieves data via the master list binding. It is common practice to use relative binding for the detail page if the user navigates to one. The client will now use the same binding path as the master page does. If the information is not transparent and comprehensive, OData clients might attempt subsequent updates that could lead to errors in your data. Thinking of the example above, if your detail page modifies or deletes data of its relative binding path, also the data of your master page will be modified. This could lead to errors or incoherence in the data.

* Do use the OData system query options $filter, $top, $skip, $orderby

Clients can, and should, use the query options to tailor responses to your needs. Content creators should support these query options to enable clients to optimize requests.

* Do provide a usable set of filterable properties

Make as many properties as possible filterable. At least all primary and foreign key properties should be filterable.

* Do make your properties concrete

Do not name your properties like "data" or "dynamicData" and so on. You must not include dynamic JSON data within your properties (eg. a property called data containing a dynamic JSON string).

* Do use the right data type for your properties

For more information, see: ABAP Dictionary Type to EDM.Type Mapping.

*** Do represent quantities and monetary amounts as Edm.Decimal**

Use the annotation `sap:unit` for the amount or quantity property to point to the corresponding currency or unit. If you need to distinguish currency units from units of measure, use the corresponding `sap:semantics` annotation.

*** Do use Edm.IntXX or Edm.Decimal for NUMC fields**

This takes care of leading zeros and alpha conversions

*** Do use media resources and media link entries**

Binary data in common formats (for example `PDF` , `GIF` , `JPEG`) is naturally represented in OData as a media resource with a corresponding media link entry, that is, a structured record describing the media resource, and containing a link to it. In this way, the binary can be accessed directly via its own URL, just like a browser accesses a picture in the Internet. In fact, a browser can then access your binary data directly, without needing to know OData.

Another benefit is performance. Binary data as a media resource is streamed directly byte by byte, whereas binary data hidden within an OData Property of type `Edm.Binary` is represented as a string of hex digits, thereby doubling its size.

*** Do provide navigation properties to link related data**

If your model contains Customers and SalesOrders, provide navigation between them as `/Customers(4711)/SalesOrders` and `/SalesOrders(42)/Customer` . This will ensure that the client knows how to construct a query on SalesOrders using a CustomerID, or the URI of the customer resource from the CustomerID.

If one of your entity types has an xxxID property, make sure you also provide a corresponding xxx entity type and navigation to it. The same applies for xxxCode; provide an entity set that lists all possible code values and meaningful descriptions for them. The service should be self-describing.

It is possible to create a referential constraints for an association. A referential constraint is similar to a relational database table foreign key, defining relationships between and within tables.

*** Do follow links provided by the server**

URI construction rules can change, but the basic convention surrounding links will not. Following links is therefore future-proof.

Note Strive to develop high-quality OData services. For more information, see Creating High-Quality OData Services.

Don'ts

*** Don't think in SOAP terms**

For more information, see: Do think in REST terms.

*** Don't construct URIs in your client application**

For more information, see: Do follow links provided by the server.

*** Don't invent your custom query options**

Clients cannot discover custom query options for themselves. You have to define how your clients are to use every single custom query option.

Use function imports with `HttpMethod="GET"` returning collection of entity type if you need to expose special queries that do not fit `$filter`. Function imports are described in the `$metadata` document and can be discovered by clients.

*** Don't force the client to construct links**

For more information, see: Do provide navigation properties to link related data.

*** Don't use Placeholder001 or ForFutureUse properties**

For more information, see: Do make your properties concrete.

*** Don't use binary properties**

For more information, see: Do use media resources and media link entries.

*** Don't use generic name-value entity types**

It is inappropriate to express entity-relationship models with just two entities, one named Entity and one named Relationship.

*** Don't just tunnel homegrown XML**

Performance

This chapter focuses on asynchronous backend OData calls. It shows how you can make your OData calls "Promise ready".

Normal ECMAScript 5 Projects

Find out how to wrap your OData calls with Promises and make use of the Promise Chain feature.

Modern ECMAScript 6 Projects

> *Requires modern Browsers*

If your project uses modern JavaScript, you can consume Promises using the Async / Await pattern and handle OData calls as if they where synchronous.

UI5 Documentation

For a general performance improvement for the UI5 framework, there is a checklist in the official UI5 documentation with which various things can be checked. Among other things, the focus here is on asynchronous methods instead of synchronous ones.

Performance Checklist

Promise

With the help of Promises, you can easily

- handle asynchronous methods
- control the execution of async methods
  - in parallel
  - sequential
- control the flow of async methods in success / error cases
- enhance the UX with loading indicators
- ...

If you are new to Promises, please make yourself familiar before continuing:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

All OData calls used in this article refer to OData v2.

Basic OData call

This is a very basic OData call where the key and the callbacks are hardcoded.

```
var model = that.getView().getModel();
model.read("/MyEntitySet('12345')", {
  success: function (data, response) {
    // success handler
  },
  error: function (error) {
    // error handler
  }
});
```

Promisified OData call

A wrapped version of the basic OData call.

The key is given as parameter and the wrapping method returns a promise. The `resolve` / `reject` methods are used in the `success` / `error` callbacks to resolve or reject the Promise.

As the `resolve` method takes only one parameter, data and response are passed as object.

```
function readMyEntity(id) {
  return new Promise(function (resolve, reject) {
    var model = that.getView().getModel();
    model.read("/MyEntitySet('" + id + "')", {
      success: function (data, response) {
        resolve({ data: data, response: response });
      },
      error: function (error) {
        reject(error);
      }
    });
  });
}
```

The success and error handling can now be done after invoking the method. This is done by applying the Promise Chain Pattern.

```
readMyEntity(12345)
  .then(function (result) {
    // success handler
    console.log(result.data, result.response);
  })
  .catch(function (error) {
    // error handler
    console.error(error);
  });
```

Refactor the Promisified OData call

Now, let's make the OData call more robust for everyday usage:

- hand over the whole entity as object and construct the key from its data
- provide an option to add additional data like filters to the call (`mParameters` as described in the documentation)
- make sure its not possible to provide `success` / `error` callbacks with the parameters object as this would break the promise functionality

The UI5 framework detects when the same OData call is done multiple times. It does only one request and returns the same result object for every call. If this result object is being changed, it changes for all calls. To prevent this from happening, the result is being copied with `Object.assign()` .

If you plan to use this with IE11, make sure to provide the Object.assign Polyfill. In case the Polyfill is not option for you, a simple `JSON.parse(JSON.stringify(data))` could do the trick as well.

```
/**
 * Returns the result of the OData call
 * @param {object} payload - The payload which equals to the entity
 * @param {object} parameters - The parameters like Filters added to the OData call
 * @returns {Promise} Promise object represents result of the OData call
 */
function readMyEntity(payload, parameters) {
  var model = this.getView().getModel();
  parameters = parameters || {};
  return model.metadataLoaded()
    .then(function () {
      return new Promise(function (resolve, reject) {
        var key = model.createKey("/MyEntitySet", payload)
        // prevent success / error callbacks to be overwritten
        var params = Object.assign({}, parameters,
          {
            success: function (data, response) {
              // prevent accidently change of response data for subsequent calls
              var dataCopy = Object.assign({}, data);
              resolve({ data: dataCopy, response: response });
            },
            error: function (error) {
              // additional error handling when needed
              reject(error);
            }
          });
        model.read(key, params);
        // update could look like this:
        // model.update(key, payload, params);
        // create could look like this:
        // model.create("/MyEntitySet", payload, params);
        // delete could look like this:
        // model.remove(key, params);
      })
    })
}

readMyEntity({ id: 12345 })
  .then(function (result) {
    console.log(result.data, result.response);
  })
  .catch(function (error) {
    console.error(error);
  });
```

Chaining multiple OData calls together

Breaks after the first Promise which does not resolve.
The `finally` method is not supported by the Promise Polyfill for IE11.

when you call Promises within a then, return their result and continue with it from the next chained then. Beside Promises, you can also return static values.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#chaining

```
myElement.setBusy(true);
readMyEntity({ id: 12345 })
  .then(function (result) {
    console.log(result.data, result.response);
    return readMyEntity({ id: 67890 })
  })
  .then(function (result) {
    console.log(result.data, result.response);
    return readMyEntity({ id: 34567 })
  })
  .then(function (result) {
    console.log(result.data, result.response);
  })
  .catch(function (error) {
    console.error(error);
  })
  .finally(function () {
    myElement.setBusy(false);
  });
```

Run multiple OData calls in parallel

Promise.all

Breaks with the first Promise that is rejected. The results are ordered the same way your promises where.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all

```
Promise.all([
  readMyEntity({ id: 12345 }),
  readMyEntity({ id: 67890 })
])
  .then(function (results) {
    console.log(results);
    // success handler
  })
  .catch(function (error) {
    console.error(error);
    // error handler
  });
```

Promise.allSettled

Waits until all Promises are fullfilled/rejected and tells the status as well as the result.
For two Promises where the first resolves and the second gets rejected, this could like like this:

```
// example result with two promises
[
  {status: "fulfilled", value: ...},
  {status: "rejected",  reason: ...}
]
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/allSettled

```
Promise.allSettled([
  readMyEntity({ id: 12345 }),
  readMyEntity({ id: 67890 })
])
  .then(function (results) {
    console.log(results);
    // success / error handler
  });
```

async / await

async / await is not compatible with IE 11!

I suggest reading the Promise chapter first.

Get the Promisified OData call ES6 ready

You could use the Promisified OData call, but why not use modern syntax? 😎

Documentation of the used features:

- Destructuring assignment
- Default parameters
- Spread Syntax
- Arrow function expressions
- Shorthand property names

```
/**
 * Returns the result of the OData call
 * @param {object} payload - The payload which equals to the entity
 * @param {object} parameters - The parameters like Filters added to the OData call
 * @returns {Promise} Promise object represents result of the OData call
 */
function readMyEntity({ payload, parameters = {} }) {
  const model = this.getView().getModel()
  return model.metadataLoaded()
    .then(() => {
      return new Promise((resolve, reject) => {
        const key = model.createKey('/MyEntitySet', payload)
        // prevent success / error callbacks to be overwritten
        const params = {
          ...parameters,
          success: (data, response) => {
            // prevent accidently change of response data for subsequent calls
            resolve({ data: { ...data }, response });
          },
          error: error => {
            // additional error handling when needed
            reject(error)
          },
        }
        model.read(key, params)
        // update could look like this:
        // model.update(key, payload, params)
        // create could look like this:
        // model.create('/MyEntitySet', payload, params)
        // delete could look like this:
        // model.remove(key, params)
      })
    })
}
```

async / await syntax sugar

```
async function init() {
  try {
    myElement.setBusy(true)
    const { data, response } = await readMyEntity({ id: 12345 })
    console.log(data, response)
  } catch (error) {
    console.error(error)
  } finally {
    // code placed here will run, even if you throw an error in catch
    myElement.setBusy(false)
  }
}
```

Run multiple Promises in parallel

Promise.all

Breaks with the first Promise that is rejected.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all

```
async function init() {
  try {
    const [result1, result2] = await Promise.all([
      readMyEntity({ id: 12345 }),
      readMyEntity({ id: 67890 }),
    ])
    console.log(result1, result2)
  } catch (error) {
    console.error(error)
  }
}
```

Promise.allSettled

Waits until all Promises are fullfilled and tells the status as well as the result.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/allSettled

```
async function init() {
  try {
    const [result1, result2] = await Promise.allSettled([
      readMyEntity({ id: 12345 }),
      readMyEntity({ id: 67890 }),
    ])
    console.log(result1, result2)
  } catch (error) {
    console.error(error)
  }
}
```

Automated Testing

Automated testing is an essential part of the development workflow to ensure the technical correctness of the requirement. It is a recommended practise to guide the implementation through "Test-Driven Development" (TDD). To test against your acceptance criteria you should combine "TDD" with "Behaviour-Driven Development" (BDD).

How does "TDD" work?

TDD is defined by the red-green-refactor cycle:

1. Write a new test that describes a part of the requirement and make sure that it fails (red)
2. Write production code to let the test pass (green)
3. Refactor both, test and production code, to clean it up (refactor)

If you are new to TDD it will help to do so-called "baby steps". By this we mean that you write the simplest implementation possible. Then let the tests guide you to the next implementation iteration.

How does "BDD" fit in?

BDD uses the "Given-When-Then" pattern to test against the acceptance criteria. E. g.:

```
Given product is dish washer
 When product number is 123456
 Then the price should show 699€
```

The above text is then mapped to to test implementation, which executes your production code. This or a similar approach is commonly used for integration- and acceptance testing of you application.

How does "TDD" work in the context of SAPUI5?

If you generated you project structure with `easy-ui5` you'll find your test stubs within the `test` subdirectory. By default `opa` is used for integration tests. To run all tests run the test goal in `package.json` with npm

```
npm run test
```

Project test structure

Parent directories and files were omitted to focus on the test directory structure.

```
<project-dir>/uimodule/webapp/test
├── integration
│   ├── AllJourneys.js
│   ├── MainJourney.js
│   ├── /arrangements
│   │   └── Startup.js
│   ├── opaTests.qunit.html
│   ├── opaTests.qunit.js
│   └── /pages
│       └── Main.js
├── testsuite.qunit.html
├── testsuite.qunit.js
└── /unit
    ├── AllTests.js
    ├── /controller
    │   └── MainView.controller.js
    ├── /helper
    ├── /model
    ├── unitTests.qunit.html
    └── unitTests.qunit.js
```

Examples

The following example snippets are based on a new project generated by `easy-ui5` and the `testing-ui5-cart` example project on GitHub (see references for link).

Test suite runner (`test/testsuite.qunit.js`):

```
window.suite = function () {
    "use strict";

    var oSuite = new parent.jsUnitTestSuite(),
        sContextPath = location.pathname.substring(0,
location.pathname.lastIndexOf("/") + 1);

    oSuite.addTestPage(sContextPath + "unit/unitTests.qunit.html");
    oSuite.addTestPage(sContextPath + "integration/opaTests.qunit.html");

    return oSuite;
};
```

Unit Testing

QUnit Test (`test/unit/MainView.controller.js`):

```
sap.ui.define([
 "com/myorg/myUI5App/controller/MainView.controller"
], function(
 MainView
) {
 "use strict";

 QUnit.module("MainViewController", {
  beforeEach: function () {
   this.MainView = new MainView();
   sinon.stub(this.MainView, "returnFromCallMe", function () {
    return "stub";
   });
  },
  afterEach: function () {
      sinon.restore();
  }
 }, function () {
  QUnit.test("Should demonstrate how general stubs can be used", function (assert) {
   assert.strictEqual(this.MainView.callMe(), "stub", "Return value of callMe
function should be from stub, but was from implmentation");
  });

    QUnit.test("Should demonstrate how inline stubs can be used", function (assert)
{
      var stub = sinon.stub(this.MainView, "conditionalWithinAFunction", function ()
{
        return false;
      });
      assert.strictEqual(this.MainView.callMe(), 1);
      assert.strictEqual(stub.callCount, 1, "The conditionalWithinAFunction function
has been successfully called");
    });
 });
});
```

Integration Test

OPA Test (`test/integration/pages/Main.js`):

```
sap.ui.require([
  "sap/ui/test/Opa5",
  "sap/ui/test/actions/Press"
], function (Opa5, Press) {
  "use strict";

  var sViewName = "com.myorg.myUI5App.view.MainView";

  Opa5.createPageObjects({
    onTheMainPage: {
      viewName: sViewName,

      actions: {
        // add action functions here
        iPressTheButton: function () {
          return this.waitFor({
            controlType: "sap.m.Button",
            actions: new Press(),
            errorMessage: "App does not have a button"
          });
        }
      },

      assertions: {
          // add assertion functions here
          iShouldSeeTheTitle: function () {
            return this.waitFor({
              controlType: "sap.m.Title",
              properties: {
                text: "com.myorg.myUI5App"
              },
              success: function() {
                Opa5.assert.ok(true, "The page shows the correct title");
              },
              errorMessage: "App does not show the expected title
com.myorg.myUI5App"
            });
          }
        }

      }
    });

});
```

Implementation of the controller (`<project-dir>/uimodule/webapp/controller/MainView.controller.js`):

```
sap.ui.define(["com/myorg/myUI5App/controller/BaseController"], function
(Controller) {
    "use strict";

    return Controller.extend("com.myorg.myUI5App.controller.MainView", {
      returnFromCallMe: function() {
        return 0;
      },
      conditionalWithinAFunction: function() {
        return true;
      },
      callMe: function() {
        return this.conditionalWithinAFunction() ? this.returnFromCallMe() : 1;
      }
    });
});
```

For further reading regarding this important topic on UI5 we recommend the `Testing SAPUI5 Applications` book (see references for the link).

References

- Testing SAPUI5 Applications (https://www.rheinwerk-verlag.de/testing-sapui5-applications/)
- Testing UI5 Cart Sample Application (https://github.com/ArnaudBuchholz/testing-ui5-cart)
- Sinon Stubs (https://sinonjs.org/releases/latest/stubs/)
- Easy UI5 (https://github.com/SAP/generator-easy-ui5)

Official SAP Documents

Here are the most important SAP Documents related to UI5 and Fiori

Table of Contents

UI5 Documentation

Entry point for the most important documents related to UI5 is:

https://ui5.sap.com

Most important Links in UI5

- Documentation
    - Best Practices for App Developers
- API Reference
- Samples
    - Walkthrough
- Demo Apps
    - Walkthrough
- Tools
    - UI5 Tooling
    - UI5 Inspector
    - Icon Explorer

Fiori Desgin Guidelines

Best Practices in design language for SAP Fiori. A Complete Overview of most controls with best practice how to implement and "Do´s" and "Dont´s"

Fiori Desgin Guidelines

SAP Fiori Apps Reference Library

A Overview of all Standard Fiori Apps, including the Product Features and the Implementation Information is found here:

https://fioriappslibrary.hana.ondemand.com/sap/fix/externalViewer/

The help for this overview is found here: Fiori Apps Reference Library - User Guide

SAP Fiori Launchpad

Entry point for all relevant documentation to SAP Fiori Launchpad is this overview:

https://help.sap.com/viewer/product/SAP_FIORI_LAUNCHPAD/EXTERNAL/en-US

Community

SAP

An Overview of most SAP Community Ressources are found on the SAPUI5 Topic Page or on the Fiori Page :

SAP Community Topic SAPUI5

SAP Community Topic Fiori

SAP Answers

All Questions and Answers related to UI5 are found when using the tag "SAPUI5" or "Fiori":

SAP Answers Tag SAPUI5

SAP Answers Tag Fiori

Overview of all Answer Tags

SAP Blogs

Same as above, you can use the Tags to find relevant Blog Entries:

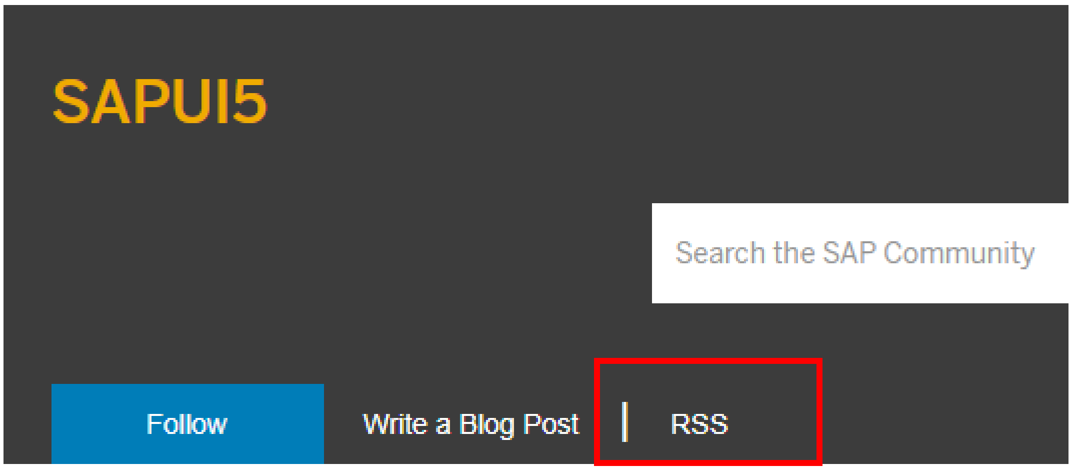
SAP Blogs Tag SAPUI5

SAP Blogs Tag SAPUI5

Overview of all Blog Tags

Extra Tip

You can use the RSS Feed for every Blog Tag to use in your favourite RSS-Reader (you can also do this for Answers, but that´s maybe a bit silly)



GitHub

Official SAP GitHub Repositry

UI5 Community

SAP Mentors & Friends

Official DSAG GitHub Repository

ABAP Open Source Projects

Slack

abapGit Not only discussions about abapGit.

SAP Mentors Everything around SAP Development.

OpenUI5 All topics around the UI5 Framework.