Week 4 Unit 2

Update and Manipulate Data

Please perform the exercises below to your app project as shown in the video.

Table of Contents

[1 Override Language 2](#_Toc453748302)

[2 Open The Add View 3](#_Toc453748303)

[3 Enter and Save Data 8](#_Toc453748304)

[4 Network Trace 12](#_Toc453748305)

[5 Beyond this Session 13](#_Toc453748306)

## Preview

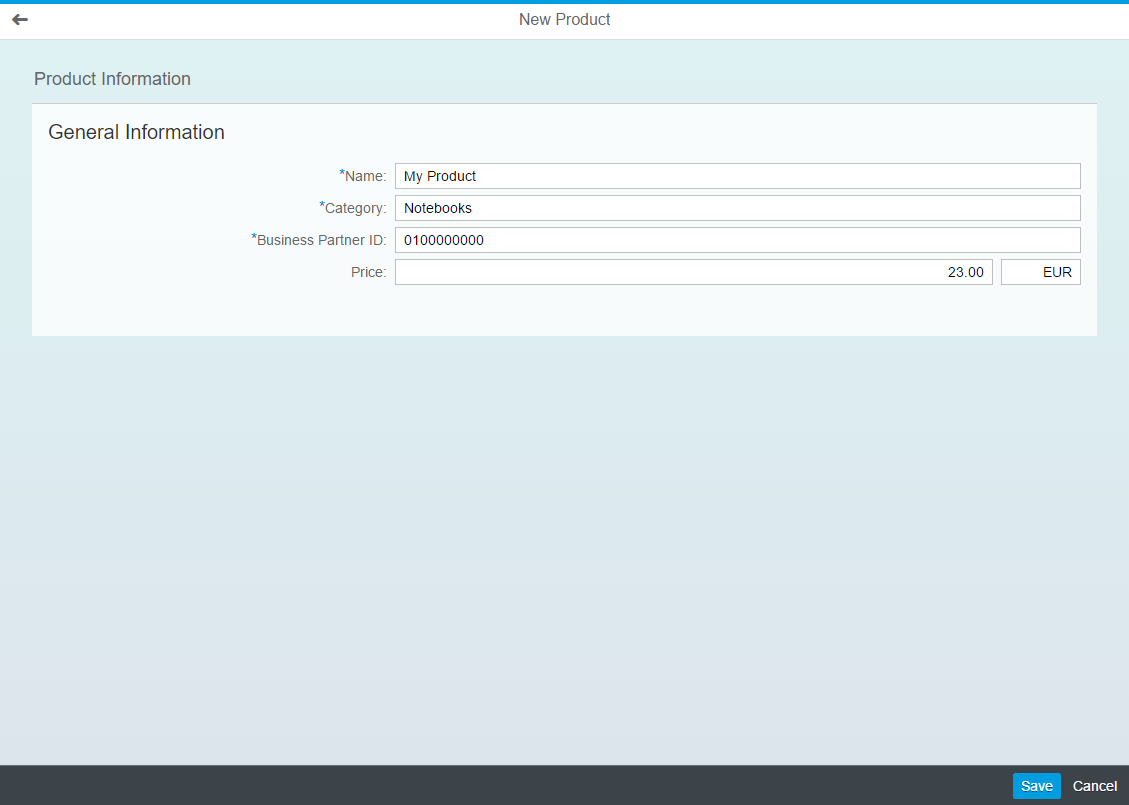


Figure 1 - Preview of the app after doing this unit’s exercises

# Override Language

The OData service used in this exercise has some internal logic that does work in all logon languages. As a workaround we set the logon language to English.

| Explanation | **Screenshot** |
| --- | --- |
| 1. Open the **Run Configurations** |  |
| 1. Select the run configuration you are using (e.g. Run App) 2. Switch to tab **URL Components** 3. Enter **sap-language** as name 4. Enter **EN** as value 5. Save the change by pressing **OK** |  |

# Open The Add View

**webapp/i18n/i18n.properties**

…

...

#~~~ Add View ~~~~~~~~~~~~~~~~~~~~~~~~~~

#XTIT: Add view title

addPageTitle=New Product

**#XTIT: The titel of the form**

**formTitle=Product Information**

**#XTIT: The titel of the form group**

**formGroupLabel=General Information**

**#YMSG: The not found text is displayed when there was an error loading the resource (404 error)**

**newObjectCreated=The product {0} has been added**

…

First of all you need a bunch of new i18n properties. Add them after into the **Add View** section.

### webapp/manifest.json

…

"sap.ui5": {

…

"models": {

…

"": {

"dataSource": "mainService",

"settings": {

**"defaultBindingMode": "TwoWay",**

"metadataUrlParams": {

"sap-documentation": "heading"

}

}

}

},

…

By default the OData model is configured to use OneWay binding. This means data can be displayed but not changed. To enable the view to change model data automatically you set the defaultBindingMode to TwoWay.

### webapp/controller/Add.controller.js

…

/\* =========================================================== \*/

/\* event handlers \*/

/\* =========================================================== \*/

\_onRouteMatched: function() {

**// register for metadata loaded events**

**var oModel = this.getModel();**

**oModel.metadataLoaded().then(this.\_onMetadataLoaded.bind(this));**

**},**

**\_onMetadataLoaded: function () {**

**// create default properties**

**var oProperties = {**

**ProductID: "" + parseInt(Math.random() \* 1000000000, 10),**

**TypeCode: "PR",**

**TaxTarifCode: 1,**

**CurrencyCode: "EUR",**

**MeasureUnit: "EA"**

**};**

**// create new entry in the model**

**this.\_oContext = this.getModel().createEntry("/ProductSet", {**

**properties: oProperties**

**});**

**// bind the view to the new entry**

**this.getView().setBindingContext(this.\_oContext);**

**},**

**…**

You replace the comment from the last unit in the \_onRouteMatched function with a usage of the metadataLoaded promise of the OData model. The then function will be executed whenever the metadata has been loaded (This might happen synchronous or asynchronously. Learn more about “promises” [in the internet](https://www.google.de/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=javascript%20promise)).

In the new \_onMetadataLoaded function, we create the oProperties collection - a first set of hard-coded properties for the new object. The product service has 8 mandatory fields and we want to default some of them without prompting for user input. This includes generating a random ProductID.

**Note: Generation of IDs on client or server**This example generates IDs on the client to keep the code simple. Still, when generating IDs on the client, submitting them to the server might fail because IDs are already used. In productive apps, you will usually rather let the server generate IDs, so that you do not have to deal with ID collisions in the client-side code.

After calling createEntry, the OData model contains a new object that (a) contains all properties specified in the metadata and (b) defaults the properties with your oProperties. The createEntry function only puts the data to the model in the client, but does not yet request creation on the server.

When creating the new entry in the model, we assume it will be successfully submitted to the server later on in most cases. Still, there are two cases when this does not happen: When the user navigates back to the work list, or when he clicks the “Cancel” button. In these cases, we have to make sure the new entry does not remain in the model, but is discarded. Otherwise, when the user really submitted a new entry after such a case, the OData request would also include such “outdated” entries. To be able to discard the entry, we keep a reference to the binding context returned by createEntry.

Last but no least, we bind the view to the **binding context** pointing to the new object in the model.

### webapp/view/Add.view.xml

<mvc:View

controllerName="opensap.manageproducts.controller.Add"

xmlns:mvc="sap.ui.core.mvc"

xmlns:semantic="sap.m.semantic"

**xmlns:smartfield="sap.ui.comp.smartfield"**

**xmlns:smartform="sap.ui.comp.smartform"**

xmlns="sap.m">

<semantic:FullscreenPage

id="page"

title="{i18n>addPageTitle}"

showNavButton="true"

navButtonPress="onNavBack">

**<semantic:content>**

**<smartform:SmartForm**

**id="form"**

**editable="true"**

**title="{i18n>formTitle}"**

**class="sapUiResponsiveMargin">**

**<smartform:Group**

**id="formGroup"**

**label="{i18n>formGroupLabel}">**

**<smartform:GroupElement>**

**<smartfield:SmartField**

**id="nameField"**

**value="{Name}"/>**

**</smartform:GroupElement>**

**<smartform:GroupElement>**

**<smartfield:SmartField**

**id="categoryField"**

**value="{Category}"/>**

**</smartform:GroupElement>**

**<smartform:GroupElement>**

**<smartfield:SmartField**

**id="supplierIDField"**

**value="{SupplierID}"/>**

**</smartform:GroupElement>**

**<smartform:GroupElement>**

**<smartfield:SmartField**

**id="priceField"**

**value="{Price}"/>**

**</smartform:GroupElement>**

**</smartform:Group>**

**</smartform:SmartForm>**

**</semantic:content>**

</semantic:FullscreenPage>

</mvc:View>

You add two new XML namespaces for the smart controls.

You extend the view with a SmartForm that allows the user to enter those product properties that have not been defaulted on creating the entry.

The SmartForm contains one group that holds the smart fields. For each SmartField you only have to specify the binding to the model properties. Other information – like the field label – is read automatically by the SmartField from the OData metadata.

### Run the App

Observe that, next to the **price**, an additional input field for the **currency code** is displayed.

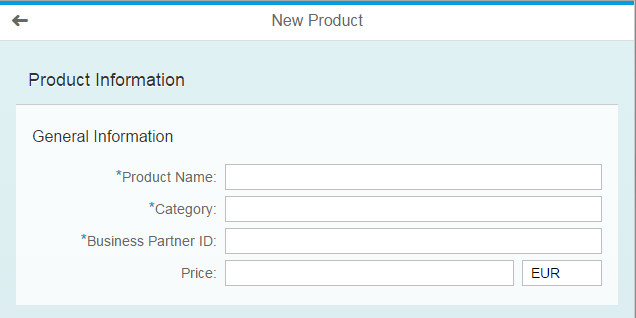


Figure 2 - SmartForm for a new product



Figure 3 - Product entity in the service metadata

The app contains a local copy of the metadata describing our backend service interface in file

webapp/localService/metadata.xml. Open it and search for the Product entity.

Observe that the sap:label annotation is set for the Name, Category, SupplierID and Price. You also see that the sap:unit annotation CurrencyCode is set for the price. The product property CurrencyCode has the sap:semantics annotation also set to CurrencyCode.

# Enter and Save Data

### webapp/view/Add.view.xml

…

<semantic:FullscreenPage

id="page"

title="{i18n>addPageTitle}"

showNavButton="true"

navButtonPress="onNavBack">

<semantic:content>

…

</semantic:content>

**<semantic:saveAction>**

**<semantic:SaveAction id="save" press="onSave"/>**

**</semantic:saveAction>**

**<semantic:cancelAction>**

**<semantic:CancelAction id="cancel" press="onCancel"/>**

**</semantic:cancelAction>**

</semantic:FullscreenPage>

</mvc:View>

You add two additional actions to the semantic page for **Save** and **Cancel**.

### webapp/controller/Add.controller.js

…

\_onMetadataLoaded: function () {

…

},

**/\*\***

**\* Event handler for the cancel action**

**\* @public**

**\*/**

**onCancel: function() {**

**this.onNavBack();**

**},**

**/\*\***

**\* Event handler for the save action**

**\* @public**

**\*/**

**onSave: function() {**

**this.getModel().submitChanges();**

**},**

…

onNavBack: function() {

**// discard new product from model.**

**this.getModel().deleteCreatedEntry(this.\_oContext);**

var oHistory = History.getInstance(),

sPreviousHash = oHistory.getPreviousHash();

…

You implement the event handler for the **Cancel** action by simply calling the existing event handler to navigate back.

You implement the event handler for the **Save** action by calling the submitChanges function of the OData model. The OData model will then send HTTP requests for all changed data to the server.

For the case that the new entry is not submitted to the server, you discard it in the onNavBack event handler by calling deleteCreatedEntry on the OData model and passing the binding context that you remember from onMetadataLoaded.

### webapp/controller/Add.controller.js

sap.ui.define([

"opensap/manageproducts/controller/BaseController",

"sap/ui/core/routing/History"**,**

**"sap/m/MessageToast"**

], function(BaseController, History**, MessageToast**) {

"use strict";

…

\_onMetadataLoaded: function () {

// create default properties

var oProperties = {

ProductID: "" + parseInt(Math.random() \* 1000000000, 10),

TypeCode: "PR",

TaxTarifCode: 1,

CurrencyCode: "EUR",

MeasureUnit: "EA"

};

// create new entry in the model

var oContext = this.getModel().createEntry("/ProductSet", {

properties: oProperties**,**

**success: this.\_onCreateSuccess.bind(this)**

});

// bind the view to the new entry

this.getView().setBindingContext(this.\_oContext);

},

**\_onCreateSuccess: function (oProduct) {**

**// navigate to the new product's object view**

**var sId = oProduct.ProductID;**

**this.getRouter().navTo("object", {**

**objectId : sId**

**}, true);**

**// unbind the view to not show this object again**

**this.getView().unbindObject();**

**// show success messge**

**var sMessage = this.getResourceBundle().getText("newObjectCreated", [oProduct.Name]);**

**MessageToast.show(sMessage, {**

**closeOnBrowserNavigation : false**

**});**

**},**

…

You register a callback onCreateSuccess for the successful creation of the product on the server. You do this in the code where you call createEntry on the model.

In this callback, you get the complete product object as parameter oProduct. You access the router and trigger the navigation to the object route with the product ID as parameter objectId. You unbind the view to not show this object again.

You finally show a MessageToast that indicates the successful creation of the product to the user. The additional parameter closeOnBrowserNavigation prevents the routing to the close the message toast automatically on navigation, so that the user can actually see it.

**Run the App, Enter Data and Save**

When calling the app, navigate to the add page and enter the following values to create a new product:

* A **name** of your choice.
* **Notebooks** as **category**.
* **0100000000** as **business partner ID**.(0, 1, 8 zeros)
* A **price** of your choice.

Press **Save** and make sure that the new product has been created.

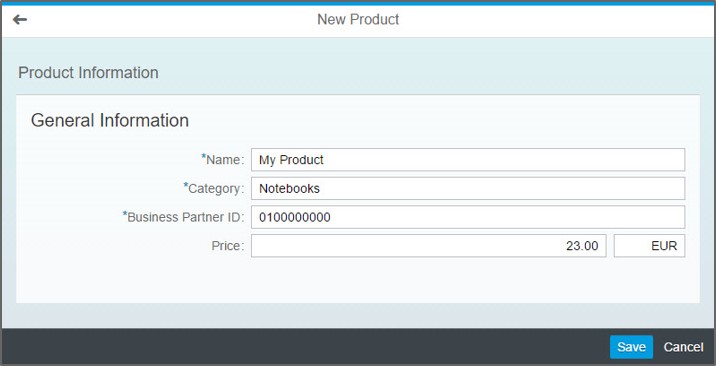


Figure 4 – Addition of a new product

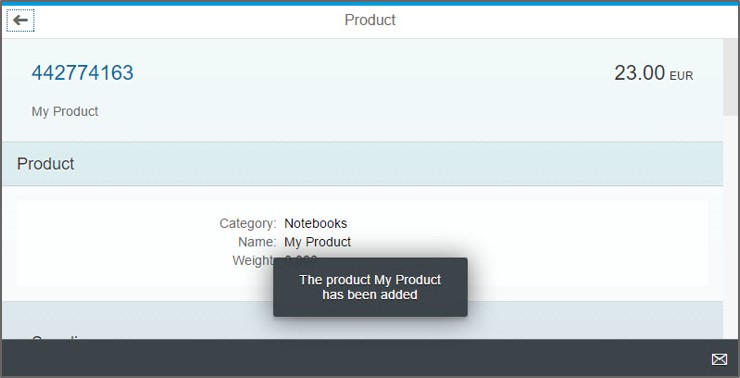


Figure 5 - The new product has been added

# Network Trace

When troubleshooting issues, it is helpful to check the **network** **trace** of Chrome to understand which HTTP requests have been sent by the OData model and analyze the error information in the responses.

| Explanation | **Screenshot** |
| --- | --- |
| 1. Open the app in Chrome and press **F12** to open the developer tools. 2. Switch to the **Network** tab. 3. Search for **batch.** 4. Now the request list only shows the requests send by the OData model. |  |
| 1. Find the request that triggered the product creation on the server. When batch processing is enabled for the OData model, this is the $batch request that contains the **POST** request in the **Headers.** |  |

# Beyond this Session

This unit only shows the first steps in setting up the complete update scenario. The following requirements have been omitted:

* **Busy Indication**: The **Add** view needs to be set busy while the server is processing the request. You find sample code for this in the **Worklist** and **Object** view of the generated app by searching for “busy”.
* **Error Handling**: The generic ErrorHandler generated with the application will show a generic error dialog., You might want to display more specific error messages. Also the busy indication needs to be reset on errors.
* **Input Validation**: The data needs to be validated and the resulting messages displayed to the user. SAPUI5 offers support for this with the [Message Handling](https://sapui5.hana.ondemand.com/sdk/#docs/guide/62b1481d3e084cb49dd30956d183c6a0.html)
* **Value Help**: The user needs assistance when entering the data. Smart fields provide reuse functionality for this. [Learn More…](https://sapui5.hana.ondemand.com/sdk/#docs/guide/3361e270c62c46c9893eaefb2966d62e.html)
* **Data Loss Confirmation**: If the user tries to leave the screen without saving entered data, he should be prompted to confirm the loss of data. You can use method hasPendingChanges of the OData model to check this.

## Related Information

[Message Handling](https://sapui5.hana.ondemand.com/sdk/#docs/guide/62b1481d3e084cb49dd30956d183c6a0.html)

[Smart Controls Tutorial](https://sapui5.hana.ondemand.com/sdk/#docs/guide/64bde9a8879d4f418e2849f7285dd757.html)

**Coding Samples**

Any software coding or code lines/strings (“Code”) provided in this documentation are only examples and are not intended for use in a productive system environment. The Code is only intended to better explain and visualize the syntax and phrasing rules for certain SAP coding. SAP does not warrant the correctness or completeness of the Code provided herein and SAP shall not be liable for errors or damages cause by use of the Code, except where such damages were caused by SAP with intent or with gross negligence.