WEEK 4 UNIT 5
CREATING CUSTOM CONTROLS

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

Table of Contents

Preview

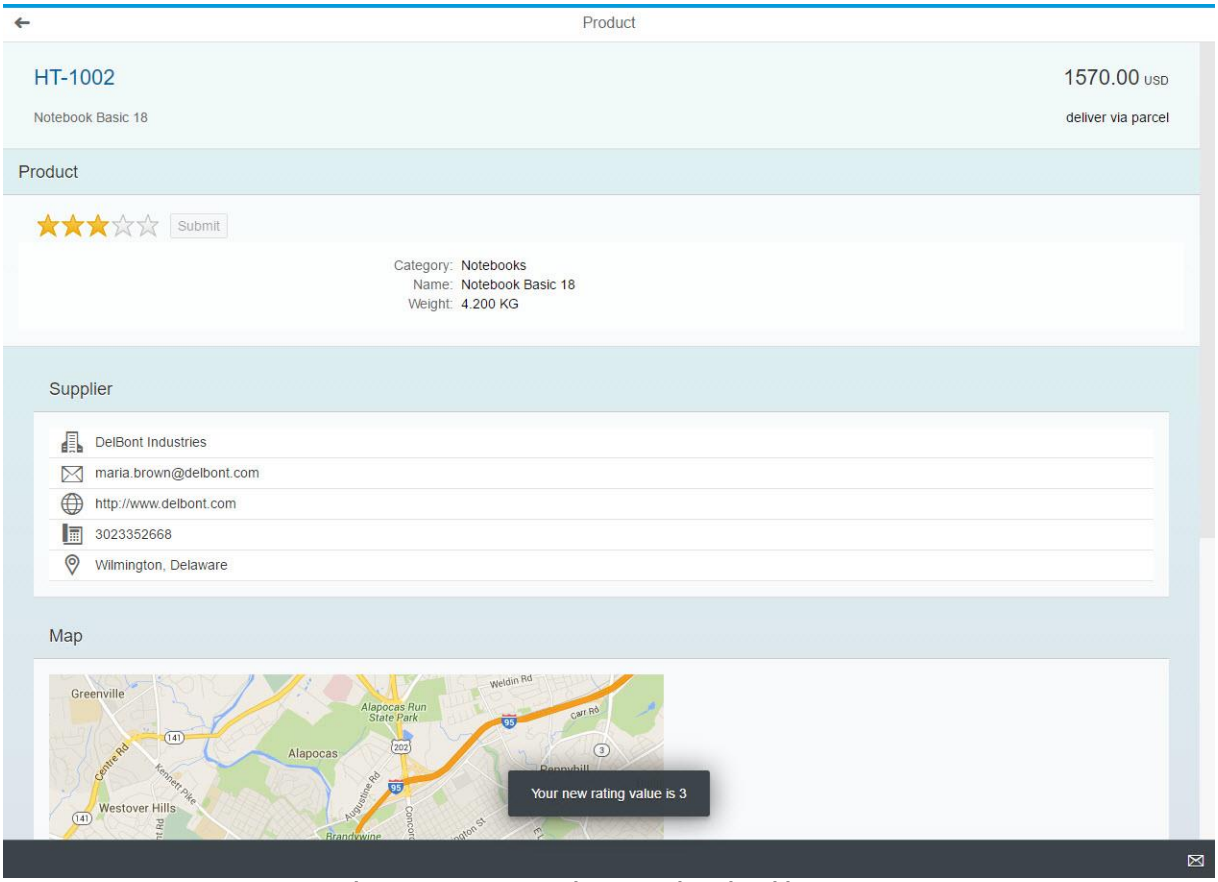

Figure 1 - A custom rating control on the object page

1 CREATING THE SKELETON FOR A NEW CUSTOM CONTROL

webapp/control/ProductRate.js (NEW)

```
sap.ui.define(["sap/ui/core/Control"], function(Control) {

  "use strict";


  return Control.extend("opensap.manageproducts.control.ProductRate", {

     metadata : {
        properties : {},
        aggregations : {},
        events : {}
     },

     init : function() {

     },

     renderer : function(oRm, oControl) {
        oRm.write("<div>Hello World!</div>");
     }

  });
});
```

In this step we want to create a first simple control as base for the target design:

Figure 1 - Target Design for the Custom Control

- We want to give our users the possibility to rate a product
- The rating for the product can be submitted by clicking a button
- The new control should throw an event after the user's submission that exposes the current vote value
- The button should only be enabled if the user has changed his vote. Initially, the button should be disabled

We use the already known `sap.ui.define` syntax and require the `sap.ui.core.Control` base class that we will extend. We add the metadata section to it. It is a simple object where we can later define the properties, aggregations and events to be offered by the control.

As methods we add the `init` function that is called when the control gets initialized, and the `renderer` function that will be used to write the control DOM. We use the render manager to write the HTML.

2 BUILD A FIRST VERSION OF THE PRODUCT RATING CONTROL

webapp/control/ProductRate.js

```
sap.ui.define([
    'sap/ui/core/Control',
    "sap/m/RatingIndicator",
    "sap/m/Button"
 ],
 function(Control, RatingIndicator, Button) {

 "use strict";

return Control.extend("opensap.manageproducts.control.ProductRate", {

   metadata : {
      properties : {
         value : {type : "float", defaultValue : 0}
      },
      aggregations : {
         _rating : {type : "sap.m.RatingIndicator", multiple : false,
            visibility : "hidden"},
         _button : {type : "sap.m.Button", multiple : false, visibility :
              "hidden"}
      },
      events : {
         valueSubmit : {
            parameters : {
               value : {type : "float"}
            }
         }
      }
   },

   init : function() {
      this.setAggregation("_rating", new RatingIndicator({
         value : this.getValue(),
         maxValue : 5
      }).addStyleClass("sapUiTinyMarginEnd"));

      this.setAggregation("_button", new Button({
         text : "{i18n>productRatingButtonText}",
         enabled : false
      }));
   },

   renderer : function(oRm, oControl) {
      oRm.write("<div");
      oRm.writeControlData(oControl);
      oRm.addClass("sapUiSmallMarginBeginEnd");
      oRm.writeClasses();
      oRm.write(">");

      oRm.renderControl(oControl.getAggregation("_rating"));
      oRm.renderControl(oControl.getAggregation("_button"));

      oRm.write("</div>");
   };
});
```

In this step we add the definition for aggregations, events and properties to the control metadata and add the implementation for the `init` method and the renderer.

In the `metadata` section of the control we therefore define several properties that we make use of in the implementation:
Properties
Getter and setter functions for these properties will automatically be created and we can also bind them to a field of the data model in an XML view if we like.

- **Value**

We define a control property `value` that will hold the value that the user rated.

Aggregations

We need two internal controls to realize our rating functionality, a rating and a button. We therefore create two hidden aggregations by setting the `visibility` attribute to `hidden`, which indicates that these aggregations are private to the control and not a part of its API. This way, we can use the models that are set on the view also in the inner controls and SAPUI5 will take care of the lifecycle management and destroy the controls when they are not needed anymore.

- **_rating**

An `sap.m.RatingIndicator` control for user input

- **_button**

An `sap.m.Button` to submit the rating

Note:
Aggregations can also be used to hold arrays of controls but we just want a single control in each of the aggregations, so we need to adjust the cardinality by setting the attribute `multiple` to `false`.

Events
Applications can register to events and process the result using the event parameters.

- **ValueSubmit**
  We specify a `valueSubmit` event that the control will fire when the rating is submitted. It contains the current value as an event parameter.

In the `init` function that is called by SAPUI5 automatically whenever a new instance of the control is created, we set up our internal controls. We instantiate the two controls and store them in the internal aggregations by calling the framework method `setAggregation` that has been inherited from `sap.ui.core.Control`. We pass the name of the internal aggregations that we specified above and the new control instances. We specify some control properties to make our custom control look nicer. The initial text for the button is referenced from our i18n model.

With the help of the SAPUI5 render manager and the control instance that are passed as a reference, we can now render the HTML structure of our control. We render the start of the root tag `<div` and call the helper method `writeControlData` to render the ID and other basic attributes of the control inside the div tag.

Next, we add a standard margin class provided by SAPUI5. This CSS class and others that have been added in the view are then rendered by calling `writeClasses` on the render manager instance. Then we close the surrounding div tag and render the two internal controls by passing the content of the internal aggregations to the render manager's `renderControl` method. This will call the renderer of the controls and add their HTML to the page. Finally, we close our surrounding `<div>` tag.

webapp/i18n/i18n.properties

```
…
#~~~ Object View ~~~~~~~~~~~~~~~~~~~~~~~~~~

#XTIT: Object view title
objectTitle=Product

#XTIT: Submit Rating Button text
productRatingButtonText=Submit

…
```

The resource bundle is extended with the strings that we reference from the custom control.

3 ADD EVENT HANDLING AND FINALIZE THE CONTROL

webapp/control/ProductRate.js

```
sap.ui.define([
    'sap/ui/core/Control',
    "sap/m/RatingIndicator",
    "sap/m/Button"
 ],

 function(Control, RatingIndicator, Button) {

 "use strict";

 return Control.extend("opensap.manageproducts.control.ProductRate", {

    …

    init : function() {
       this.setAggregation("_rating", new RatingIndicator({
          value : this.getValue(),
          maxValue : 5,
          liveChange : this._onRate.bind(this)
       }).addStyleClass("sapUiTinyMarginEnd"));

       this.setAggregation("_button", new Button({
          text : "{i18n>productRatingButtonText}",
          press : this._onSubmit.bind(this),
          enabled : false
       }))
    },

    _onSubmit : function() {
       this.fireEvent("valueSubmit", {
          value : this.getValue()
       });
       this.getAggregation("_button").setEnabled(false);
    },

    _onRate : function(oEvent) {
       this.setValue(oEvent.getParameter("value"));
       this.getAggregation("_button").setEnabled(true);
    },

    renderer : function(oRm, oControl) {
       …
    }

 });
});
```

In this step we finalize our control by adding handlers for the internal events and implementing the event provided by the control itself.

First, we add handlers for the `press` and `liveChange` events of the aggregations. With every change of the rating, we will change the `value` property of our control accordingly and enable the button. If the user submits his rating by clicking the button, we will fire the control event with the current value as an event parameter.

4 ADD THE CONTROL TO OUR APP

webapp/controller/Object.controller.js

```
sap.ui.define([
    "opensap/manageproducts/controller/BaseController",
    "sap/ui/model/json/JSONModel",
    "sap/ui/core/routing/History",
    "opensap/manageproducts/model/formatter",
    "sap/m/MessageToast"
 ], function (
    BaseController,
    JSONModel,
    History,
    formatter,
    MessageToast
 ) {
    "use strict";
    return Controller.extend("opensap.manageproducts.controller.Object", {

       formatter: formatter,

       onInit: function(){
          …
       },

       /* ============================================================ */
       /* event handlers                                               */
       /* ============================================================ */

       onRatingChanged: function(oEvent) {
          var iValue = oEvent.getParameter("value"),
            sMessage =
this.getResourceBundle().getText("productRatingSuccess", [iValue]);
          MessageToast.show(sMessage);
       },
…
});
  }
);
```

Now let's add an event handler function `onRatingChanged` to the controller. It retrieves the value parameter from the event object that we have filled inside the custom control.

Note that we do not add it to the `ProductDetails` controller used for the `ProductDetails` view so far, but to the `Object` controller. This allows to simplify the code and is reasonable in this case because the `ProductDetails` does not contain any important logic. We will update the controller name in the view definition later.

Then we load the dependency for `sap.m.MessageToast` and show a message to give feedback about the successful interaction. Here we will use an i18n text with parameter from our `ResourceBundle`. To make this work, we hand over the parameter as an argument to the `getText` function call.

webapp/i18n/i18n.properties

```
…
#~~~ Object View ~~~~~~~~~~~~~~~~~~~~~~~~~

#XTIT: Object view title
```

```
objectTitle=Product

#XTIT: Submit Rating Button text
productRatingButtonText=Submit

#YMSG: Submit Rating Success Message
productRatingSuccess=Your new rating value is {0}
```

Finally, we add this text to our i18n.properties file. We add the parameter placeholder with index 0 into curly brackets at the position we want it to be displayed.

webapp/view/ProductDetails.view.xml

```
<mvc:View
  controllerName="opensap.manageproducts.controller.Object"
  xmlns="sap.m"
  xmlns:mvc="sap.ui.core.mvc"
  xmlns:form="sap.ui.layout.form"
  xmlns:course="opensap.manageproducts.control">
  <Panel
    headerText="{i18n>productTitle}"
    expandable="{device>/system/phone}"
    expanded="true">
    <content>
      <course:ProductRate valueSubmit="onRatingChanged"/>
      <form:SimpleForm id="simpleForm">
        <form:content>
          <Label text="{i18n>productCategoryLabel}"/>
          <Text text="{Category}"/>
          <Label text="{i18n>productNameLabel}"/>
          <Text text="{Name}"/>
          <Label text="{i18n>productWeightLabel}"/>
          <Text text="{= ${WeightMeasure} + ' ' + ${WeightUnit}}"/>
        </form:content>
      </form:SimpleForm>
    </content>
  </Panel>
</mvc:View>
```

First, we change the controller used for the view from the `ProductDetails` controller to the `Object` controller where we added the new event handler.

Now we add the control to the `ProductDetails` view. A new namespace `course` is added to the `Object` view so that we can reference the custom control in the view.

We then add an instance of the `ProductRate` control to our detail page and register our event handler for the `valueSubmit` event.

Now run the app and try out your custom control. It should work as specified above!

5 MORE CUSTOM CONTROL EXAMPLES

Custom controls can be implemented in many different ways. In addition to the simple example of this unit, we have linked some more complex controls on **JSbin** below. You can take them as a reference for your own developments:

Extending an Existing Control: Lightbox
Using External Libraries (D3.js): Chart
Custom Rendering/Composite Control: Rotary Knob

Note:
If you struggle with the implementation of a custom control, you can always have a look at the controls that are delivered with SAPUI5.,They apply the same concepts and can be analyzed easily using the debugging tools of your browser.

Related Information

Developing SAPUI5 Controls
Defining the Control Metadata
API Reference: sap.ui.core.Control