




Join GitHub today[Dismiss](#)

GitHub is home to over 40 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)

UI5 Code Jam conducted at Gurgaon on 14th September, 2019

 **5** commits **1** branch **0** releases **1** contributorBranch: **master** ▾[New pull request](#)[Find File](#)[Clone or download](#) ▾

 PavanVasist Update README.md	Latest commit 83effff0 2 days ago
 README.md	Update README.md 2 days ago
 localService.zip	Add files via upload 2 days ago

 **README.md**

ui5codeJamHandsOn

UI5 Code Jam conducted at Gurgaon on 14th September, 2019

Introduction

In this session, you will learn about developing web application with SAPUI5. We will use the SAP Web IDE as development environment and build a small application to get familiar with our latest developer tools and recommendations.

Scenario Introduction

Our customer "Keep Cool Inc." is a maintainer of several icehouses across the country. Recently those have been upgraded with new sensors with internet connection, so that their measuring values are available as a service. To make use of this data and to improve their internal workflows the company asked us to provide an application leveraging this sensor data, visualize it and provide an overview of the current state of each sensor, so that they can react quickly on any issues.

Exercises

1. Setup WebIDE
2. Navigation and Libraries in Manifest
3. Adding Content to the View
4. GridList and DataBinding
5. Improve the List Item Visualization
6. Filtering with the IconTabBar
7. Fragment containing a SelectDialog
8. Second View with Navigation
9. Card with NumericHeader
10. Chart with DataBinding

Project Setup

Access your SAP WebIDE

TODO

SAPUI5 Application Template

As a first step, we will generate a freestyle app using the SAPUI5 Application template. SAP Web IDE templates are preconfigured app projects that serve as a good starting point for creating best practice apps in SAPUI5. The app simply shows an empty page, which we will fill in the next steps to create our "Icehouse Management" app.

Application from Template

1. Enter SAP WebIDE with the provided Link.
2. Go to the code editor.
3. File -> New Project from Template
4. Choose SAPUI5 Application
5. Under Basic Information enter the following:
 - i. Project Name: icehouse
 - ii. Namespace: ui5codejam
6. Template Customization:
 - i. View Type: XML
 - ii. View Name: App
7. Click finish

Hands-On Development

In this section we will start to add content to our application. An `sap.ui.core.mvc.XMLView` showing multiple sensors will be the first part of our app.

Navigation and Libraries in manifest.json

Our first XML View will be the `Sensors.view.xml` listing several sensors. But before we add content, we define the `Sensors.view.xml` as home view in the router configuration of the `manifest.json`.

Create the first View

Create a new file in the `view` folder and name it `/Sensors.view.xml`. We will use this file to add additional content lateron.

Open `manifest.json` file and adapt home route and target to `Sensors.view.xml`. This will enable the navigation to our new View.

- Replace the existing route in `routes` with the route `RouteSensors`
- Add the new Target `TargetSensors` to `targets` which references to our View

webapp/manifest.json

```
"routes": [{
  "name": "RouteSensors",
  "pattern": "",
  "target": ["TargetSensors"]
}],
"targets": {
  "TargetSensors": {
    "viewType": "XML",
    "transition": "slide",
    "clearControlAggregation": false,
    "viewId": "sensors",
    "viewName": "Sensors",
    "viewLevel": 1
  }
}
```

1. We are going to use several libraries like the `sap.m` or `sap.f` in our application. Therefore, we need to add them to the `dependencies` under the `sap.ui5` section. With the list of libraries under the `libs` property we tell the SAPUI5 Core which libraries to load for the usage in our component.

webapp/manifest.json

```

"dependencies": {
  "minUI5Version": "1.60.1",
  "libs": {
    "sap.ui.core": {},
    "sap.ui.layout": {},
    "sap.m": {},
    "sap.f": {},
    "sap.suite.ui.microchart": {}
  }
},

```

1. Now we can have a look at our application by clicking on "run" and select the `index.html`. The application still looks quite empty, so let's add some content.

2. Adding Content to the View

After we have prepared the manifest, we are going to add content to the `Sensors.view.xml`. Copy the yellow passages inside your XML view. For the moment we add an empty `sap.m.IconTabBar`.

view/Sensors.view.xml

```

<mvc:View
  xmlns:mvc="sap.ui.core.mvc"
  xmlns="sap.m"
  displayBlock="true">
  <Page id="page" title="{i18n>title}">
    <content>
      <IconTabBar id="iconTabBar" class="sapUiResponsiveContentPadding">
        <content>
          </content>
        </IconTabBar>
      </content>
    </Page>
  </mvc:View>

```

3. GridList and DataBinding

We would like to show the sensor data in a `sap.f.GridList`. This requires some preparations in our XMLView.

1. Add `sap.f` and `sap.ui.layout.cssgrid` to xml namespace to make sure that the required resources are available in the View.

view/Sensors.view.xml

```

...
<mvc:View
  xmlns:mvc="sap.ui.core.mvc"
  xmlns:grid="sap.ui.layout.cssgrid"
  xmlns:f="sap.f"
  xmlns="sap.m"
  displayBlock="true">
...

```

1. Add `sap.f.GridList` to the `content` aggregation of the `IconTabBar`. Inside the `GridList` we are having the `sap.m.CustomListItem` as list element.

view/Sensors.view.xml

```

...
<f:GridList id="sensorsList" noDataText="No sensors">
  <f:customLayout>
    <grid:GridBoxLayout/>
  </f:customLayout>
  <f:items>
    <CustomListItem>
    </CustomListItem>
  </f:items>

```

```

    </f:items>
  </f:GridList>
  ...

```

1. Data Binding: Import Sensor Data

Our sensor data would ideally be provided by a live service. However, as no such service exists (yet), we import a json file to our project that mocks this behavior. It is a common practice to use mock data during the development process and later exchange it with a live service. Please download the [mockdata](#)

Right-Click on webapp. Select Import. Click on File or Project. Click on Browse and select the localService.zip(Downloaded from [localService](#) folder) file Click on OK. Click on Import to complete. The localService folder should appear in your project tree under the webapp folder.

1. Define data source in manifest

To be able to bind the data to our `sap.f.GridList` we need to declare the models in our application manifest.

- Define `dataSources` under the `sap.app` section of the `manifest.json`. This will enable access to our local service and make it available in the `models` section.
- Define `sensors` model under the `models` section. This ensures, that we will have access to this model in all the Views inside our Component. We are going to configure a declarative model for the mock data we just imported. In order to use the mock data, we define a JSONModel with the name `sensors`. We can do this manually in our app controller or declaratively in the application manifest. If we add it to the manifest inside the `sap.ui5` section, the model will automatically be initialized on application start.

webapp/manifest.json

```

"sap.app": {
  "dataSources": {
    "sensors": {
      "type": "JSON",
      "uri": "./localService/sensors.json"
    }
  }
},
"sap.ui5": {
  "models": {
    "sensors": {
      "type": "sap.ui.model.json.JSONModel",
      "dataSource": "sensors"
    }
  }
}

```

1. Data Binding: Bind GridList to sensors model

Bind the `items` aggregation of the `sap.f.GridList` to path `sensors>/sensors`. `sensors` references our recently defined model and `/sensors` points to the property inside. As this is an Array with several entries we additionally would like to define a sorting and grouping with the according direction. In the `sorter` we can configure this by using the according properties.

view/Sensors.view.xml

```

...
<f:GridList id="sensorsList"
  items="{path: 'sensors>/sensors', sorter: {path:'customer', group:true, descending: false}}"
  noDataText="No sensors">
...

```

2. The list items will be defined one time as a template. This template will then be repeated to represent each entry of the `sensors` array. Furthermore we add the location details to our `sap.m.CustomListItem`. `location` hereby references the property of each of the displayed sensor items.

view/Sensors.view.xml

```
...
<CustomListItem>
  <Title text="{sensors>location}"/>
</CustomListItem>
...
```

In the preview you should see a list of items showing the location and grouped by the customer name.

7. Improve the List Item Visualization

In this chapter we will work on the visual parts of the `CustomListItem` which we have added in the previous exercise. We want to display more details on the list item and make it more assessable by implementing a formatter function in our view controller. To get the layouting done easily we use the `sap.m.HBox` and `sap.m.VBox` controls .

To give the customer the best possible overview, let us add some color to our application! Therefore we introduce a new layout and structure for the item and also show an `sap.ui.core.Icon` there.

1. Add xml namespace `xmlns:core="sap.ui.core"` to the view to have the `sap.ui.core.Icon` available

view/Sensors.view.xml

```
...
<mvc:View
  xmlns:core="sap.ui.core"
  xmlns:mvc="sap.ui.core.mvc"
  xmlns:grid="sap.ui.layout.cssgrid"
  xmlns:f="sap.f"
  xmlns="sap.m"
  displayBlock="true">
...
```

1. Add temperature icon and layouting to the `sap.m.CustomListItem`

`sapUiSmallMarginTop` and `sapUiSmallMarginEnd` are predefined css classes which add spacing in between controls. `sap.m.HBox` and `sap.m.VBox` are helpers for layouting your application.

view/Sensors.view.xml

```
<CustomListItem type="Active">
  <layoutData>
    <FlexItemData growFactor="1" shrinkFactor="0"/>
  </layoutData>
  <HBox justifyContent="SpaceBetween">
    <VBox justifyContent="SpaceBetween" class="sapUiSmallMarginTop sapUiSmallMarginBegin">
      <Title text="{sensors>location}"/>
      <Label text="Distance:"/>
    </VBox>
    <core:Icon src="sap-icon://temperature" size="2.5rem"
      class="sapUiSmallMarginTop sapUiSmallMarginEnd"/>
  </HBox>
  <HBox justifyContent="SpaceBetween" class="sapUiTinyMarginTop sapUiSmallMarginBottom sapUiSmallMarginB
    <ObjectNumber number="{sensors>distance}" unit="km"/>
  </HBox>
</CustomListItem>
```

1. Create a new file in the folder `controller` named `Sensors.controller.js` . All functions defined in the Controller can be used in our View. This gives us more flexibility to implement specific functionality to improve the visualization in our View.

*controller/Sensors.controller.js**

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

  return Controller.extend("ui5codejam.icehouse.controller.Sensors", {
```

```
});
});
```

2. Add modules `sap/ui/core/IconColor` and `sap/m/MessageToast` as dependencies to the `Sensors.controller.js`. We will use them later on during this step.

*controller/Sensors.controller.js**

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/core/IconColor",
    "sap/m/MessageToast"
], function (Controller, IconColor, Toast) {
    "use strict";

    ...
});
```

3. Assign the `Sensors.controller.js` to the XML View by adding the `controllerName`. This is necessary as then the Controller's functions can be used in the View.

```
<mvc:View
    controllerName="ui5codejam.icehouse.controller.Sensors"
    xmlns:core="sap.ui.core"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns:grid="sap.ui.layout.cssgrid"
    xmlns:f="sap.f"
    xmlns="sap.m"
    displayBlock="true">
    ...
</mvc:View>
```

4. Implement `onInit` function

In the `onInit` method of the controller that is executed once when the view is loaded, we access our sensors model. We are going to use the threshold data for implementing the formatter function in the next step.

controller/Sensors.controller.js

```
onInit: function() {
    this._aCustomerFilters = [];
    this._aStatusFilters = [];

    this._oSensorModel = this.getOwnerComponent().getModel("sensors");
    this._oSensorModel.dataLoaded().then(function() {
        this._oThreshold = this._oSensorModel.getProperty("/threshold");
        Toast.show("All sensors online!");
    }).bind(this);
}
```

5. Implement formatter function `formatIconColor` and add it to the `Sensors.controller.js`. Based on the actual temperature of the sensor we want to display the icon in the according color.

*controller/Sensors.controller.js**

```
formatIconColor: function(iTemperature) {
    if (iTemperature < this._oThreshold.warning) {
        return IconColor.Default;
    } else if (iTemperature >= this._oThreshold.warning && iTemperature < this._oThreshold.error) {
        return IconColor.Critical;
    } else {
        return IconColor.Negative;
    }
}
```

6. Open the `Sensors.view.xml` file. Bind `color` property to path `sensors>temperature/value` and assign `formatIconColor` function as formatter

view/Sensors.view.xml

```
<core:Icon src="sap-icon://temperature" size="2.5rem"
  color="{path: 'sensors>temperature/value', formatter:'.formatIconColor'}"
  class="sapUiSmallMarginTop sapUiSmallMarginEnd"/>
```

If you run the app, you should see more details displayed on the list items and a temperature icon. Some of the icons should have a different color according to the temperature information.

8. Filtering with the IconTabBar

As our customer needs the full overview to make decisions quickly, we want to give him an option to narrow down the List, based on the actual temperature of a sensor. Therefore we enhance our `sap.m.IconTabBar`.

1. Add `sap.m.IconTabFilter` elements to the items aggregation of the `IconTabBar`. Those will be visible as Icons on top of the bar and the user can click them to filter the list.

view/Sensors.view.xml

```
...
<IconTabBar id="idIconTabBar" class="sapUiResponsiveContentPadding">
  <items>
    <IconTabFilter showAll="true" text="Sensors" key="All"/>
    <IconTabSeparator/>
    <IconTabFilter icon="sap-icon://status-positive" iconColor="Positive" text="Ok" key="Ok"/>
    <IconTabFilter icon="sap-icon://status-critical" iconColor="Critical" text="Warning" key="Warning"
    <IconTabFilter icon="sap-icon://status-error" iconColor="Negative" text="Error" key="Error"/>
  </items>
</IconTabBar>
...
```

2. Add module `sap/ui/model/Filter` as dependency to the `Sensors.controller.js`.

controller/Sensors.controller.js

```
sap.ui.define([
  "sap/ui/core/mvc/Controller",
  "sap/ui/core/IconColor",
  "sap/m/MessageToast",
  "sap/ui/model/Filter"
], function (Controller, IconColor, Toast, Filter) {
  "use strict";
```

3. Implement `onSensorSelect` function for filtering the sensor list items after statuses. We make also use of the previously defined threshold and use some settings of the filter to narrow down the result. `LT` for example means "lower-than".

controller/Sensors.controller.js

```
...

onSensorSelect: function (oEvent) {
  this._aCustomerFilters = [];
  this._aStatusFilters = [];

  var oBinding = this.getView().byId("sensorsList").getBinding("items"),
      sKey = oEvent.getParameter("key");

  if (sKey === "Ok") {
    this._aStatusFilters = [new Filter("temperature/value", "LT", this._oThreshold.warning, false)];
  } else if (sKey === "Warning") {
    this._aStatusFilters = [new Filter("temperature/value", "BT", this._oThreshold.warning, this._oThr
  } else if (sKey === "Error") {
    this._aStatusFilters = [new Filter("temperature/value", "GT", this._oThreshold.error, false)];
  } else {
    this._aStatusFilters = [];
  }

  oBinding.filter(this._aStatusFilters);
}
```

Press CTRL+S to save.

4. Open the `Sensors.view.xml`. Bind the `onSensorSelect` function to `select` event of the `IconTabBar`. Each time one of the Icons is clicked, this function will be called.

view/Sensors.view.xml

```
<IconTabBar id="idIconTabBar" select=".onSensorSelect" class="sapUiResponsiveContentPadding">
```

5. Making use of an **Expression Binding** we show the total count of our filter list on the first `IconTabFilter`.

view/Sensors.view.xml

```
<IconTabBar id="idIconTabBar" class="sapUiResponsiveContentPadding">
  <items>
    <IconTabFilter showAll="true" count="{=${sensors>/sensors}.length}" text="Sensors" key="All"/>
    <IconTabSeparator/>
  ...
</IconTabBar>
```

Press CTRL+S to save. In the preview you should see the `IconTabFilters` which can be pressed. Select one of the icon tabs to filter the sensor list.

9. Fragment containing a SelectDialog

For the employees, not all of their customers might be relevant. We add some kind of basic personalization to the application, by providing a dialog, in which they can select only the relevant customers.

In the following exercise we are going to implement an XML Fragment containing a dialog. In this dialog, we will display a selectable list which allows us to filter our sensors by customer.

Right click on view folder. Select New. Click on File. Enter "**CustomerSelectDialog.fragment.xml**" as File Name. Click on OK.

1. In the created `view/CustomerSelectDialog.fragment.xml` with the following content. Bind items aggregation of the `SelectDialog` to the list of customers with path `sensors>/customers`. The `StandardListItem` is used as template here.

view/CustomerSelectDialog.fragment.xml

```
<core:FragmentDefinition
  xmlns="sap.m"
  xmlns:core="sap.ui.core">
  <SelectDialog
    title="Select Customers"
    contentHeight="38.3%"
    rememberSelections="true"
    items="{
      path: 'sensors>/customers',
      sorter: {path: 'name'}
    }" >
    <StandardListItem title="{sensors>name}"/>
  </SelectDialog>
</core:FragmentDefinition>
```

2. Implement the `onCustomerSelect` function to open the dialog. In there we load the Fragment and set the according model and properties. Add the "**sap/ui/core/Fragment**" module to the dependencies. We will use `Fragment` class to load our XML Fragment.

controller/Sensors.controller.js

```
sap.ui.define([
  "sap/ui/core/mvc/Controller",
  "sap/ui/core/IconColor",
  "sap/m/MessageToast",
  "sap/ui/model/Filter",
  "sap/ui/core/Fragment"
```



```

], function (Controller, IconColor, Toast, Filter, Fragment) {
...
onCustomerSelect: function (oEvent) {
    if (this._oDialog) {
        this._oDialog.open();
    } else {
        Fragment.load({
            type: "XML",
            name: "ui5codejam.icehouse.view.CustomerSelectDialog",
            controller: this
        }).then(function(oDialog) {
            this._oDialog = oDialog;
            this._oDialog.setModel(this._oSensorModel, "sensors");
            this._oDialog.setMultiSelect(true);
            this._oDialog.open();
        }).bind(this);
    }
}
...

```

3. Add Menu Button to Page header and bind the `press` event to `onCustomerSelect` function in ***Sensors.view.xml***.

view/Sensors.view.xml

```

...
<Page>
    <headerContent>
        <Button icon="sap-icon://menu" press=".onCustomerSelect" tooltip="Select Customer" />
    </headerContent>
    <content>
...

```

5. Add the `onCustomerSelectChange` function which performs the filtering based on the text which is entered in the field of the `sap.m.SelectDialog`.

controller/Sensors.controller.js

```

...
onCustomerSelectChange: function(oEvent) {
    var sValue = oEvent.getParameter("value");
    var oFilter = new Filter("name", "Contains", sValue);
    var oBinding = oEvent.getSource().getBinding("items");
    oBinding.filter([oFilter]);
}
...

```

6. Add the `onCustomerSelectConfirm` function to apply the filtering to our sensor list.

controller/Sensors.controller.js

```

...
onCustomerSelectConfirm: function(oEvent) {
    var aSelectedItems = oEvent.getParameter("selectedItems");
    var oBinding = this.getView().byId("sensorsList").getBinding("items");
    this._aCustomerFilters = aSelectedItems.map(function(oItem) {
        return new Filter("customer", "EQ", oItem.getTitle());
    });
    oBinding.filter(this._aCustomerFilters.concat(this._aStatusFilters));
}
...

```

6. Bind `onChange` and `onConfirm` function to the `onCustomerSelect` and `confirm` events of the `SelectDialog`

view/CustomerSelectDialog.fragment.xml

```

...
<SelectDialog
    title="Select Customers"
    contentHeight="38.3%"
    rememberSelections="true"

```

```
confirm=".onCustomerSelectConfirm"
liveChange=".onCustomerSelectChange"
...

```

Run index.html. On the top right corner you should see the menu button which can be pressed to open the dialog. Click on the menu icon in the Page header. Select one or more customers from the list. Click on Select to confirm and close the Dialog.

10. Second View with Navigation

For sure, the customer not only wants to have the broad overview of all their icehouses. In case of an error they would like to inspect a sensor for how the temperature developed recently. We introduce a second view, where we display even more of the available data.

In this exercise we will add a second XML view to our application and implement the navigation functionality that allows us to switch back and forth between the views. The second view will show detailed information about a specific sensor.

1. Create a new file in the `view` folder and name it `/SensorStatus.view.xml`
2. In the `manifest.json` add a new route and target to the `routing` configuration. Note, that the new route contains a patch segment in brackets: `{index}`. This passes the currently selected index as parameter to the `routeMatched` event.

webapp/manifest.json

```
...
},
"routes": [{
...
{
  "name": "RouteSensorStatus",
  "pattern": "RouteSensorStatus/{index}",
  "target": ["TargetSensorStatus"]
}
...
"targets": {
...
"TargetSensorStatus": {
  "viewType": "XML",
  "transition": "slide",
  "viewId": "sensorStatus",
  "viewName": "SensorStatus",
  "viewLevel": 2
}
...
}

```

3. Implement the `navToSensorStatus` function to be able to navigate to the new View. Therefore we need the router and pass the index to the `navTo` function.

controller/Sensors.controller.js

```
navToSensorStatus: function(oEvent) {
  var i = oEvent.getSource().getBindingContext("sensors").getProperty("index");
  this.getOwnerComponent().getRouter().navTo("RouteSensorStatus", {index: i});
}

```

4. Add a `sap.m.Page` to our new XML View.

view/SensorStatus.view.xml

```
<mvc:View displayBlock="true"
  xmlns:mvc="sap.ui.core.mvc"
  xmlns="sap.m">

  <Page id="SensorStatusPage" title="{i18n>title}" showNavButton="true">
    </Page>

</mvc:View>

```

5. Bind `navToSensorStatus` function to the `press` event of the `CustomListItem`

view/Sensors.view.xml

```
...
<CustomListItem type="Active" press=".navToSensorStatus">
  <layoutData>
    <FlexItemData growFactor="1" shrinkFactor="0"/>
  </layoutData>
</CustomListItem>
...
```

6. Create `controller/SensorStatus.controller.js`, so that we can enhance our new View with some functionality.

controller/SensorStatus.controller.js

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

  return Controller.extend("ui5codejam.icehouse.controller.sensorStatus", {

  });
});
```

7. Implement navigation to route `"RouteSensors"` in `SensorStatus.controller.js` to be able to return to the initial screen from the second View.

controller/SensorStatus.controller.js

```
return Controller.extend("ui5codejam.icehouse.controller.sensorStatus", {
  navToSensors: function() {
    this.getOwnerComponent().getRouter().navTo("RouteSensors");
  }
});
```

8. Add back navigation to the first page with `navButtonPress=".navToSensors"` and setting the controller. Don't forget to add the controller name in the *SensorStatus.view.xml* - `ui5codejam.icehouse.controller.SensorStatus`

view/SensorStatus.view.xml

```
<mvc:View displayBlock="true"
  controllerName="ui5codejam.icehouse.controller.SensorStatus"
  xmlns:mvc="sap.ui.core.mvc"
  xmlns="sap.m">

  <Page id="SensorStatusPage" title="{i18n>title}" showNavButton="true" navButtonPress=".navToSensor"
  </Page>

</mvc:View>
```

11. Card with NumericHeader

In this step we enhance the page with a `sap.f.Card` to show some detailed data about the sensor's status. We add some layouting with box controls and add a `sap.f.cards.NumericHeader` to properly display the temperature.

- <https://sapui5.hana.ondemand.com/#/topic/5b46b03f024542ba802d99d67bc1a3f4>
- <https://sapui5.hana.ondemand.com/#/api/sap.f.Card>

1. Add `sap.f` and `sap.f.cards` libraries to `SensorStatus.view.xml`

view/SensorStatus.view.xml

```
<mvc:View displayBlock="true" controllerName="ui5codejam.icehouse.controller.SensorStatus"
  xmlns:mvc="sap.ui.core.mvc"
  xmlns="sap.m"
```

```
xmlns:f="sap.f"
xmlns:card="sap.f.cards">
```

```
...
```

1. Add the `sap.f.Card` with a card header to `SensorStatus.view.xml`.
2. Add a the customer as title of the header via databinding.

```
...
```

```
<Page id="SensorStatusPage" title="{i18n>title}" showNavButton="true" navButtonPress=".navToSensor
  <content>
    <VBox class="sapUiContentPadding">
      <f:Card>
        <f:header>
          <card:Header
            title="Customer: {sensors>customer}"
          />
        </f:header>
        <f:content>
          </f:content>
        </f:Card>
      </VBox>
    </content>
  </Page>
```

```
...
```

3. To be able to show the data in our Card we need to enable the databinding through the information we passed to the navigation in one our the previous steps.

We attach a `routeMatched` handler function to retrieve information about the selected index. Based on this information we bind the according element to the current View.

- <https://sapui5.hana.ondemand.com/#/topic/516e477e7e0b4e188b19a406e7528c1e>

controller/SensorStatus.controller.js

```
...
return Controller.extend("ui5codejam.icehouse.controller.sensorStatus", {
  onInit: function () {
    this._oRouter = this.getOwnerComponent().getRouter();
    this._oRouter.getRoute("RouteSensorStatus").attachMatched(this.onRouteMatched, this);
  },
  onRouteMatched : function (oEvent) {
    this.getView().bindElement({
      path: "/sensors/" + oEvent.getParameter("arguments").index,
      model: "sensors"
    });
  },
  ...
});
...
```

4. Open `SensorStatus.view.xml` and add `NumericHeader` to the card to improve the visualization. Additionally we include some more data based on the model. Do not forget to set the scale.

- <https://sapui5.hana.ondemand.com/#/api/sap.f.cards.NumericHeader>

view/SensorStatus.view.xml

```
...
<f:header>
  <card:NumericHeader
    title="Customer: {sensors>customer}"
    subtitle="Location: {sensors>location}, Distance: {sensors>distance}km"
    number="{sensors>temperature/value}"
    scale="°C"/>
  </f:header>
...
```

5. Add a formatter to add some semantic coloring for the card header. The formatter needs to load the thresholds from the model. Based on those it can return the correct `sap.m.ValueColor`. Don't forget to add `"sap/m/ValueColor"` as dependency

controller/SensorStatus.controller.js

```

sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/m/ValueColor"
], function (Controller, ValueColor) {

    ...
    onInit: function () {
        this._oRouter = this.getOwnerComponent().getRouter();
        this._oRouter.getRoute("RouteSensorStatus").attachMatched(this.onRouteMatched, this);

        this._oSensorModel = this.getOwnerComponent().getModel("sensors");
        this._oSensorModel.dataLoaded().then(function () {
            this._oThreshold = this._oSensorModel.getProperty("/threshold");
        }.bind(this));
    },
    ...
    ,
    formatValueColor: function(iTemperature) {
        if (iTemperature < this._oThreshold.warning) {
            return ValueColor.Neutral;
        } else if (iTemperature >= this._oThreshold.warning && iTemperature < this._oThreshold.error) {
            return ValueColor.Critical;
        } else {
            return ValueColor.Error;
        }
    }
}

```

6. For a better visualization we use the state property of the `sap.f.cards.NumericHeader` to display the number in the according color.

view/SensorStatus.view.xml

```

...
<f:header>
<card:NumericHeader
    title="Customer: {sensors>customer}"
    subtitle="Location: {sensors>location}, Distance: {sensors>distance}km"
    number="{sensors>temperature/value}"
    state="{path: 'sensors>temperature/value', formatter: '.formatValueColor'}"
    scale="°C"/>
</f:header>
...

```

Press **CTRL+S** to save. Click **Run** to preview the application. Select a card with status **"Warning"** or **"Error"**. In the preview you should see more details of the selected sensor in the CardHeader. The temperature information should be displayed in a different color depending on its value.

12. Chart with Databinding

To be able to show some historical data we leverage the `temperatureLog` of the sensor data. We use an `sap.suite.ui.microchart.InteractiveLineChart` to add the datapoints there.

- <https://sapui5.hana.ondemand.com/#/topic/9cbe3f06465e47b8a136956034a718ed>
- <https://sapui5.hana.ondemand.com/#/api/sap.suite.ui.microchart.InteractiveLineChart>

1. Add the `sap.suite.ui.microchart` library to the `SensorStatus.view.xml`.

view/SensorStatus.view.xml

```

<mvc:View displayBlock="true" controllerName="ui5codejam.icehouse.controller.SensorStatus"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns="sap.m"
    xmlns:f="sap.f"
    xmlns:card="sap.f.cards"

```

```
xmlns:mc="sap.suite.ui.microchart">
```

```
...
```

2. Add the chart to the `SensorStatus.view.xml` and bind the `temperatureLog` to the points aggregation. For each point we display the `temperature` property.

view/SensorStatus.view.xml

```
<f:content>
  <FlexBox height="450px" alignItems="Center" class="sapUiSmallMargin">
    <mc:InteractiveLineChart points="{sensors>temperatureLog}" displayedPoints="100" selection
      <mc:InteractiveLineChartPoint
        value="{sensors>temperature}"
      />
    </mc:InteractiveLineChart>
  </FlexBox>
</f:content>
...
```

3. Format the datapoint for improved readability. We use an expression to achieve this leveraging standard JavaScript functionality.

view/SensorStatus.view.xml

```
<mc:InteractiveLineChartPoint
  value="{=Number.parseFloat({sensors>temperature}.toFixed(2))}"
/>
```

4. Add semantic color for datapoints with our formatter function.

view/SensorStatus.view.xml

```
<mc:InteractiveLineChartPoint
  value="{=Number.parseFloat({sensors>temperature}.toFixed(2))}"
  color="{path: 'sensors>temperature', formatter: '.formatValueColor'}"
/>
```

5. Add labels for datapoints to get some contextual info.

view/SensorStatus.view.xml

```
<mc:InteractiveLineChartPoint
  value="{=Number.parseFloat({sensors>temperature}.toFixed(2))}"
  color="{path: 'sensors>temperature', formatter: '.formatValueColor'}"
  label="{sensors>time}"
/>
```

6. To improve the readability we format label using a `DataType`. These types are predefined and can be configured individually regarding the input and output format.

- <https://sapui5.hana.ondemand.com/#/topic/07e4b920f5734fd78fdaa236f26236d8>
- <https://sapui5.hana.ondemand.com/#/topic/91f322a06f4d1014b6dd926db0e91070>

view/SensorStatus.view.xml

```
<mc:InteractiveLineChartPoint
  value="{=Number.parseFloat({sensors>temperature}.toFixed(2))}"
  color="{path: 'sensors>temperature', formatter: '.formatValueColor'}"
  label="{
    path: 'sensors>time',
    type: 'sap.ui.model.type.Time',
    formatOptions: {
      source: { pattern: 'timestamp' },
      style: 'short'
    }
  }"
```

```
    />  
  }"
```

Press CTRL+S to save. Click Run. Click one Item of the List. In the preview you should see a chart showing historical data. Some data points should be displayed in different colors depending on their values approaching the thresholds.

CONGRATULATIONS, you have successfully completing the UI5CodeJam Hands-On scenario :-

Your UI5 application is prepared to be easily translated into other languages. The localization folder - i18n, a short name for internationalization - contains *.properties files, one for each language. Our template project contains only one file by default. In real life, you would add more *.properties files manually, depending on your target languages. Now, you might have noticed that "Title" is still shown on the Page header. The title is bound to the title property of the i18n model which was already preconfigured by the application template. To change the title, just change the property value to something like **Keep Cool Inc. - Icehouse Management**.

Summary

You have completed the exercise!

You are now able to: · Create an XML View · Add content to the View · Customize a List Item · Filter with an IconTabBar · Create a Fragment with Controls · Add Navigation between Views · Use Cards · Use Charts