HD3C04 – Routing in SAPUI5

|  |  |
| --- | --- |
| **Product and Focus**  HANA Platform/SAPUI5 | **MOTIVATION**  This case uses a simple application to explain routing in SAPUI5 applications.  **PREREQUISITES**  HD1C01w – The Hello World MVC Application |
| **Target Audience**  Undergraduate/Graduate Beginner to Intermediate |
| **Author**  Ross Hightower |
| https://bgoerke.files.wordpress.com/2013/05/section1.png | |

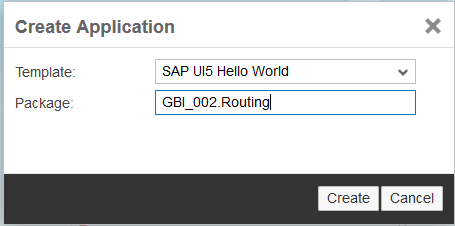
# Routing in SAPUI5

The previous case (HD1C01w) introduced the basic application architecture for SAPUI5 applications used in this curriculum. This case introduces the concept of routing which is one method to navigate between views in SAPUI5 applications.

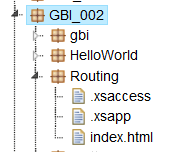
# Create the Application

Log on to the WDW and open the **Editor**. Right-click your package and select **Create Application.**

Select the **SAP UI5 Hello World** template and add .**Routing** to the package name. This will create a package called Routing and create the application in that package.



When the application is created, three files are automatically created under **Routing** package: **index.html**, **.xsaccess**, and **.xsapp.**



For an explanation of these files, see the previous case.

## Modify the Application

In this section you will rearrange the application into a more robust structure. While this structure is overkill for such a small application, it is the structure used for Fiori applications and is the structure used for application in this curriculum.

### index.html

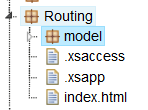
Replace the code in the index.html file with the code shown below.

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta http-equiv="X-UA-Compatible" content="IE=edge" />  <meta charset="UTF-8">    <title>UA SAPUI5 Skills App</title>  <script id="sap-ui-bootstrap"  src=" /sap/ui5/1/resources/sap-ui-core.js"  data-sap-ui-theme="sap\_bluecrystal"  data-sap-ui-libs="sap.m"  data-sap-ui-resourceroots='{  "routing": "./"  }'>  </script>  <script>  // now create a new, reusable component called ui5 (like our namespace)  sap.ui.getCore().attachInit(function() {  new sap.m.Shell("shell",{  app : new sap.ui.core.ComponentContainer({  height : "100%",  name : "routing"  })  }).placeAt("content"); });  </script>  </head>  <body class="sapUiBody" id="content">  </body>  </html> |

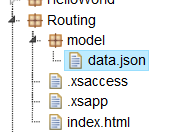
Listing

### data.json

Use the context menu of the Routing package (accessed by right-clicking the Routing package) to create a new package called **model.**



Create a file inside the model folder called **data.json**.



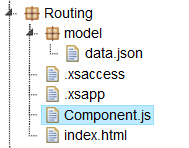
Add the following code to the data.json file.

|  |
| --- |
| {  "collection" : [  {  "prop1" : "A",  "subcol" : [  {  "subProp1" : "One"  },  {  "subProp1" : "Two"  }  ]  },  {  "prop1" : "B",  "subcol" : [  {  "subProp1" : "Three"  },  {  "subProp1" : "Four"  }  ]  }    ]  } |

Listing

### Component.js

Use the context menu to create a file in the Routing package called **Component.js**.



Insert the following code into the Component.js file.

|  |
| --- |
| jQuery.sap.declare("routing.Component");  sap.ui.core.UIComponent.extend("routing.Component",{  metadata: {  },    init: function() {    jQuery.sap.require("sap.m.routing.RouteMatchedHandler");  jQuery.sap.require("sap.ui.core.routing.HashChanger");    //call createContent  sap.ui.core.UIComponent.prototype.init.apply(this, arguments);    this.\_router = this.getRouter();    //initlialize the router  this.\_routeHandler = new sap.m.routing.RouteMatchedHandler(this.\_router);  this.\_router.initialize();    },    createContent: function() {    var oView = sap.ui.view({  id: "app",  viewName: "routing.view.App",  type: "XML",  viewData: {component: this}  });    var oModel = new sap.ui.model.json.JSONModel("model/data.json");  oView.setModel(oModel);    return oView;    }    }); |

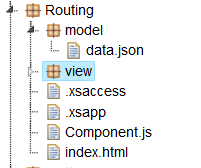
Listing

The first line declares the component, referencing the Component.js file. The routing prefix uses the resourceroots namespace created in the bootstrap element in the index.html file. The basic component is extended with the information specific to this application. In our case, we are not declaring any metadata yet but we will add metadata that configures the routing between views later. The init function initializes the component which, among other things, causes it to call the createContent function.

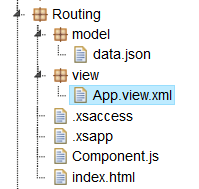
The createContent function loads the initial view of the application. The name of the view is App and it is located in the view folder. Note the use of the routing namespace prefix again. This indicates the view package is located in the application root package. Finally, a JSON model is created and added to the App view. Because the App view is the root view for the application, this model will be available to all views.

### App.view.xml

Create a new package called **view** in the Routing package the same way you created the model package.



Right-click the view package and create a new file called **App.view.xml**.



Insert the following code in the App.view.xml file so that it matches the code shown below.

|  |
| --- |
| <core:View xmlns:core="sap.ui.core"  xmlns:mvc="sap.ui.core.mvc"  xmlns="sap.m"  xmlns:html="http://www.w3.org/1999/xhtml">  <SplitApp  id="idAppControl" />  </core:View> |

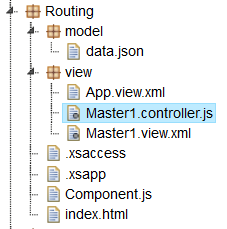
Listing

The App view is essentially the bootstrap for the rest of the application. It has no visual component itself but it creates the application object which manages the behind the scenes functions of the application. The application object, in this case, is a SplitApp which creates a structure that has a smaller container for master views on the left side of the screen and a larger container for detail pages on the right side of the screen.

Note the namespace indicates the App.view.xml file is located in the view folder which is directly under the root folder of the project.

### Master1.view.xml

Create two new files called **Master1.view.xml** and **Master1.controller.js** in the view package.



Open the Master1.view.xml file and insert the following code.

|  |
| --- |
| <core:View xmlns:core="sap.ui.core"  xmlns:mvc="sap.ui.core.mvc"  xmlns="sap.m" controllerName="routing.view.Master1"  xmlns:html="http://www.w3.org/1999/xhtml">  <Page title="Master1">  <content>    </content>  </Page>  </core:View> |

Listing

### Master1.controller.js

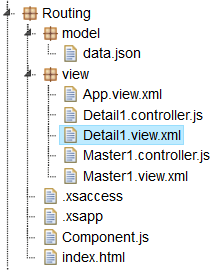
Open the Master1.controller.js file and insert the following code.

|  |
| --- |
| sap.ui.controller("routing.view.Master1", {  }); |

Listing

### Detail.view.xml

Create two new files called **Detail1.view.xml** and **Detail1.controller.js** in the view package.



Open the Detail.view.xml file and insert the following code**.**

|  |
| --- |
| <core:View xmlns:core="sap.ui.core"  xmlns:mvc="sap.ui.core.mvc"  xmlns="sap.m" controllerName="routing.view.Detail1"  xmlns:html="http://www.w3.org/1999/xhtml">  <Page title="Detail1">  <content>  </content>  </Page>  </core:View> |

Listing

### Detail1.controller.js

Open the Detail1.controller.js file and insert the following code.

|  |
| --- |
| sap.ui.controller("routing.view.Detail1", {    }); |

Listing

## Add A Default Route

If you attempt to run the application now you will receive an error because we haven’t indicated what views to load into the SplitApp control. The SplitApp control has two aggregations. One is masterPages which will be the views that appear on the left side of the screen and the second is detailPages which will be the views that appear on the right side of the screen. We need to have at least one route that loads a view into the masterPages. Another type of application control is the App control. The difference between the App control and SplitApp control is that the App control creates a full page application so it has only one aggregation for views which is called pages.

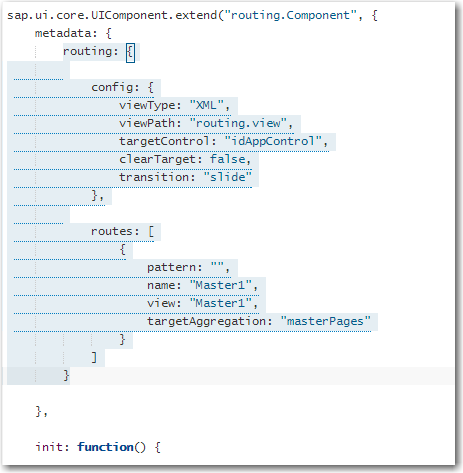
### Component.js

Open the Component.js file and add the code inside the metadata object.

|  |
| --- |
| routing: {  config: {  viewType: "XML",  viewPath: "routing.view",  clearTarget: false,  transition: "slide"  },  routes: [  {  pattern: "",  name: "Master1",  view: "Master1",  targetControl: "idAppControl",  targetAggregation: "masterPages"  }  ]  } |

Listing

The image below shows where the code is inserted.



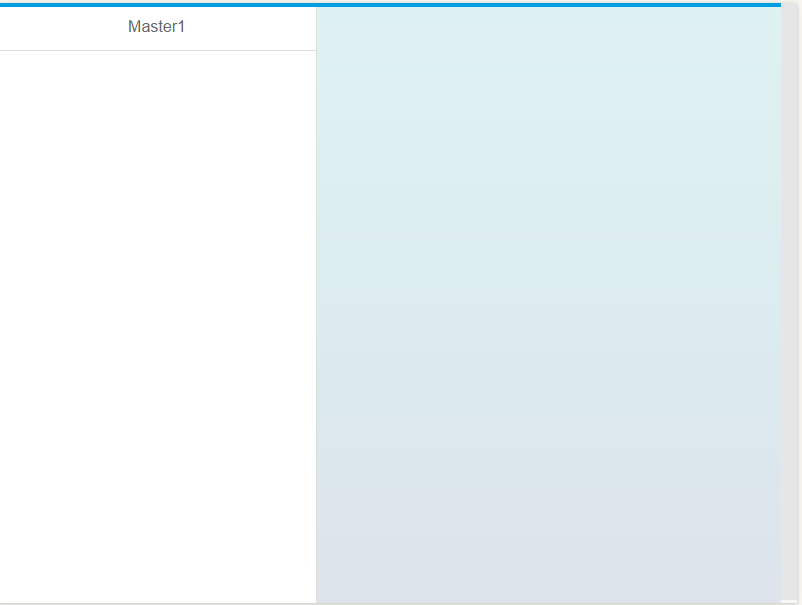
The config section of the router configuration sets some default values. Note the value of the targetControl property is the id of the SplitApp control defined in the App.view.xml view.

After the config section, the routes are defined. This code only defines one route. The pattern defines the pattern of the URL used to access the route. In this case, the “” indicates this is the default route so it will load when the user enters the base URL.



The name of the route is Master1 and the view to load is Master1.view.xml. The viewType and viewPath properties in the config section help the router find the view. Notice that the view is loaded into the masterPages aggregation of the targetControl. If the targetControl was an App control, the targetAggregation would be pages.

Run the application and you will see the screen show below. The Master1 view is loaded into the master pane but note there is nothing loaded in the detail view.



### Master1.view.xml

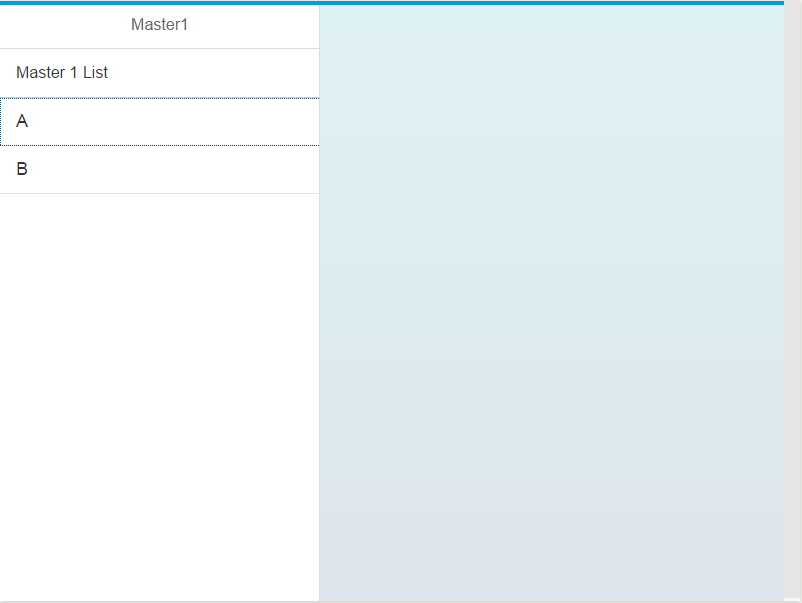
Add the code below between the <content> and </content> tags in the Master1.view.xml file.

|  |
| --- |
| <List  headerText="Master 1 List"  items="{/collection}" >  <StandardListItem  type="Active"  press="handleListItemPress"  title="{prop1}" />  </List> |

Listing

This creates a List control. The items aggregation is bound to the collection array in the JSON model that was created in the Component.js file. The aggregation binding will create a list item for each item in the collection array and the template for the list items is specified by the StandardListItem control. The title attribute of the list items is bound to the prop1 property in the collection array objects and the list items are bound to a press event handler called handleListItemPress which will be defined in the controller file later.

If you run the application now you will see the list in the master pane.



## Add the Detail View

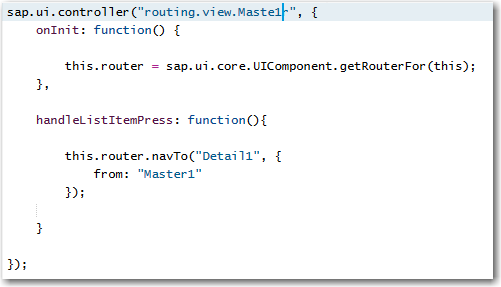
To add the detail view you must implement the handleListItemPress function to handle the press event on the list items in the Master1 view and then add the route to the Component.js file.

### Master1.controller.js

Insert the code in the Master1.controller.js file with the code shown below.

|  |
| --- |
| onInit: function() {    this.router = sap.ui.core.UIComponent.getRouterFor(this);  },  handleListItemPress: function(){  this.router.navTo("Detail1", {  from: "Master1"  });    } |

Listing



The onInit() function is invoked when the view is first instantiated. It retrieves a reference to the router object created in the Component.js file and stores it in this.router which will be available to other functions in the controller.

The handleListItemPress function uses the navTo method of the router to navigate to Detail1.

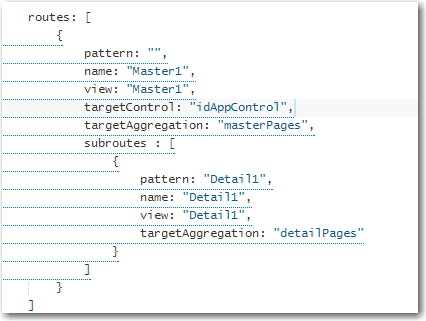
### Component.js

Modify the routing metadata in the Component.js file by adding the highlighted portions below.

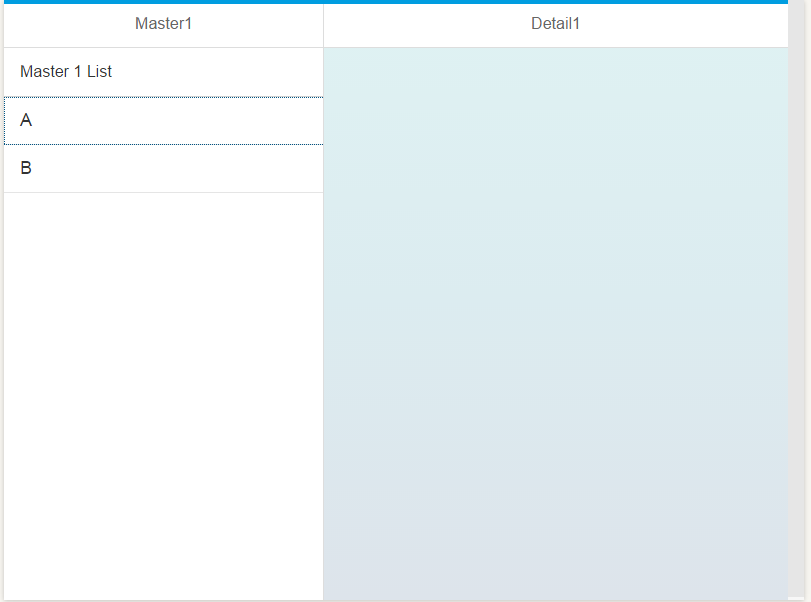
|  |
| --- |
| routes: [  {  pattern: "",  name: "Master1",  view: "Master1",  targetControl: "idAppControl",  targetAggregation: "masterPages",  **subroutes : [**  **{**  **pattern: "Detail1",**  **name: "Detail1",**  **view: "Detail1",**  **targetAggregation: "detailPages"**  **}**  **]**  }  ] |

Listing

This adds a subroute to the default route. Note that the targetAggregation is detailPages and the view is the Detail1.view.xml view.



If you run the application now and click an item on the list, the Detail1 view is loaded although it contains no content yet.



## Add Content to Detail1

The detail view should show the contents of the subcol array for the item selected in the Master1 view list. If item A is clicked, the detail view should show the subProp1 properties with values One and Two.

|  |
| --- |
| {  "collection" : [  {  "prop1" : "A",  "subcol" : [  {  "subProp1" : "One"  },  {  "subProp1" : "Two"  }  ]  },  {  "prop1" : "B",  "subcol" : [  {  "subProp1" : "Three"  },  {  "subProp1" : "Four"  }  ]  }    ]  } |

Listing

To accomplish this we need to:

1. Update the press event handler in Master1.controller.js so that it passes the index of the clicked item to the router.
2. Update the route in Controller.js so that it has a parameter to contain the index passed from Master1.controller.js.
3. Create the correct binding context in the Detail1.controller.js file and bind it to the view.
4. Add the list with appropriate bindings to Detail1.view.xml.

### Master1.controller.js

Update the handleListItemPress function with the highlighted portions shown below.

|  |
| --- |
| handleListItemPress: function(oEvent){  var entity = oEvent.getSource().getBindingContext().getPath().split("/");  this.router.navTo("Detail1", {  from: "Master1",  entity: entity[2]  });    } |

Listing

The first line retrieves the binding context from the oEvent parameter passed to the function. The oEvent function contains information about the control pressed, in this case a list item. The binding context is an object that contains information about the data bound to a control. One property of the binding context is the path to the bound data in the model. In our case the paths of the two list items will be /collection/0 and /collection/1 where the 0 and 1 indicate the array index of the item.

The split function is a JavaScript function that will split a string on the provided character and place the results in an array. In this case, the split function will create an array with three items that have the values empty string, “collection” and the index (0 or 1). The index is passed to the router in the property called “entity”.

### Component.js

Modify the Detail1 route as shown in the highlighted portion below.

|  |
| --- |
| subroutes : [  {  pattern: "Detail1/{entity}",  name: "Detail1",  view: "Detail1",  targetAggregation: "detailPages"  }  ] |

Listing

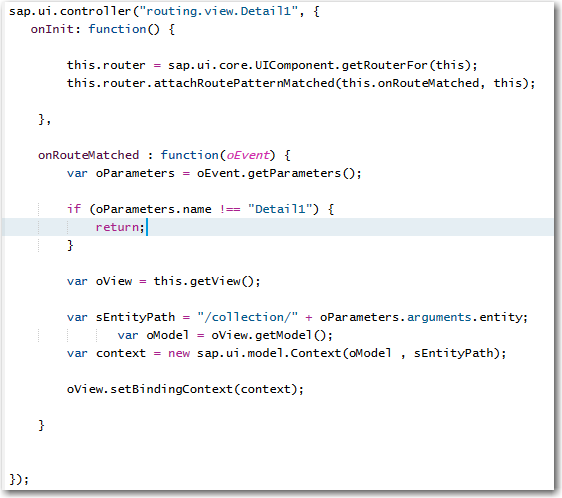
The term entity is the same that was used in the navTo function.

### Detail1.controller.js

Insert the code shown below into Detail1.controller.js file.

|  |
| --- |
| onInit: function() {    this.router = sap.ui.core.UIComponent.getRouterFor(this);  this.router.attachRoutePatternMatched(this.onRouteMatched, this);  },  onRouteMatched : function(oEvent) {  var oParameters = oEvent.getParameters();  if (oParameters.name !== "Detail1") {  return;  }  var oView = this.getView();  var sEntityPath = "/collection/" + oParameters.arguments.entity;  var oModel = oView.getModel();  var context = new sap.ui.model.Context(oModel , sEntityPath);    oView.setBindingContext(context);  } |

Listing



As with the Master1.conroller.js file, a reference to the application’s router object is retrieved in the onInit() function. The second line assigns a function called onRouteMatched in this controller file to handle the event raised when the view’s route is matched; when a navigation to this view occurs.

In the onRouteMatched() function the parameters of navigation event are retrieved using the oEvent parameter passed to the function. Among other things, these parameters include the entity parameter passed to the route. The if statement ensures that the Detail1 view is the route that was matched and not a parent route of Detail1. Next, a reference to the Detail1 view is retrieved and then a path is constructed using the entity parameter. This would be the path to the item that was clicked on the Master1 list and would be /collection/0 if the first item was clicked. A reference to the application’s model is retrieved and then used along with the path to create a binding context that is bound to the view. This whole process makes the data in the clicked item available to the controls in the Detail1 view.

### Detail1.view.xml

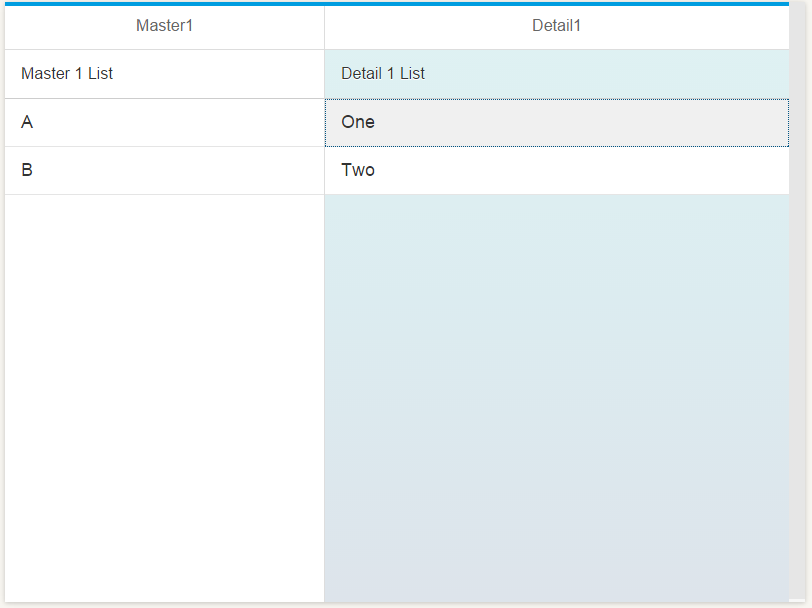
Add the code shown below between the <content> and </content> tags in the Detail1.view.xml file.

|  |
| --- |
| <List  headerText="Detail 1 List"  items="{subcol}" >  <StandardListItem  type="Active"  press="handleListItemPress"  title="{subProp1}" />  </List> |

Listing

This list is similar to the List control used in Master1.view.xml. The difference is that the items aggregation is bound to subcol rather than /collection. Note the lack of the leading /. The leading / in /collection indicates an absolute binding path from the root of the JSON object in the model. By leaving out the leading / here we are creating a binding path that is relative to the binding context of the view. So assuming the user pressed item one (that has index 0), the path for the binding context of view would be /collection/0. The binding for the list is relative to this so it would be /collection/0/subcol.

The image below shows the result of A is clicked on the Master1 list.



# Exercise

Note that we included a press event handler in the Detail1 list. Create a new view called Detail2.view.xml and implement the press event handler to navigate to the Detail2 view in the detailPages aggregation as shown below. This would also be a subroute of the Master1 route. Routes are separated by commas.

