Chapter 6, Models and Bindings, discusses the general usage and types of models in SAPUI5. Here, we also explain the different types of data binding that can be performed in SAPUI5, including property, element, aggregation, and expression binding.

“Models and Bindings”

Contents

Index

The Authors

Paul Modderman, Christiane Goebels, Denise Nepraunig, Thilo Seidel

SAPUI5: The Comprehensive Guide

737 Pages, 2020, $89.95
ISBN 978-1-4932-1900-1

www.sap-press.com/4987
Chapter 6
Models and Bindings

This chapter covers the general usage and types of models in SAPUI5 and includes a discussion on data binding.

In the previous chapter, we discussed how models work as data containers that hold all the business data your application will work with. SAPUI5 comes with several predefined model classes ready for use. In the example application, you’ve seen both a JSON model and a resource model in action. In this chapter, we’ll explain these model types in more depth and learn more about the binding functionality you’ll use throughout your applications.

Data binding involves connecting data residing in a model to part of the user interface (UI) on the screen. This connection can be either one-way or two-way. One-way will only serve the data from the model and will leave it untouched when the UI changes. The latter will keep the data from the UI and the model in sync, as you’ve seen in the running example with input fields in the previous chapter.

For most models, you can decide whether you want to use one-way or two-way binding. One-way binding doesn’t mean that data in a model can’t be updated at all. Instead, updating the model requires special actions. This concept makes more sense when we take a closer look at the amount of data our application can access and where this data comes from.

To begin, let’s look at a JSON example.

6.1 Using Models: A JSON Sample

JSON models are client-side models. After you instantiate the model object and load the data, the application has access to it.

Other models—such as the OData model, for example—are server-side models. Data doesn’t load automatically on instantiating a model. You have to request and load it
on demand. This can happen through application code, such as calling the corres-
ponding functions on the model, or it can happen automatically when data from
such a model is bound to some element on the screen using certain binding types.
The model object itself will then take care of requesting the data from the service. In
Chapter 8, we’ll dive into the depths of the OData model itself, where you’ll see more
details.

Now let’s see how to use a JSON model and apply the different SAPUI5 binding types.

### 6.1.1 Instantiation and Loading of Data

First, let’s recap the model structure of the example application and revisit what
we’ve achieved so far. We have all the business data we need for our application in a
JSON structure. Use the constructor function of the JSON model class:

```javascript
// model creation and loading the data
var oAppModel = sap.ui.model.json.JSONModel(this.getMetadata().
    getConfig().serviceUrl);
```

Next, we need some data for this model, which we want available to the controls in
our application. As before, we’ll load the data from a separate file to simulate the
answer we would get from a service.

In the background, the JSON model constructor distinguishes between a URL or a
JSON object parameter. We’ve passed a URL this time, so it tries to load the file from
the URL. You can see from the constructor’s source code that it’s using the `setData`
function of the model (see Listing 6.1).

```javascript
Listing 6.1 sap.ui.model.JSONModel Constructor
```

When the data has successfully been retrieved, it’s stored in the model object in a
property called `OData`. Note that this property isn’t meant for direct access and is
mentioned here only for clarity.

There is another option to trigger the request for the data. The JSON model also has a
method called `loadData`, which loads data manually, as shown in Listing 6.2.

```javascript
// model creation and loading the data
var oAppModel = sap.ui.model.json.JSONModel();
// loading the JSON data from the URL and storing it in the model
oAppModel.loadData("service/data.json");
```

**Listing 6.2 Instantiating the Model and Loading the Data Manually**

This convenience method is actually a wrapper for an XMLHttpRequest. It gives us a
means to directly influence some of the default behavior inherited from the parent
sap.ui.model.Model class.

We’ll leave the default behavior as is for the moment, however, and look closer at
what exactly happens when the `loadData` function is invoked (see Listing 6.3; we’ve
omitted code lines that you currently don’t need to focus on).

```javascript
JSONModel.prototype.loadData =
    function(sURL, oParameters, bAsync, sType, bMerge, bCache, mHeaders){
        
        this.fireRequestSent({
            url: sURL,
            type: sType,
            async: bAsync,
            headers: mHeaders,
            info: "cache=" + bCache + ";bMerge=" + bMerge,
            infoObject: {
                cache: bCache,
                merge: bMerge
            }
        });
        this._ajax({
            
            success: function(oData) {
                
            }
        });
    }
```

This convenience method is actually a wrapper for an XMLHttpRequest. It gives us a
means to directly influence some of the default behavior inherited from the parent
sap.ui.model.Model class.

We’ll leave the default behavior as is for the moment, however, and look closer at
what exactly happens when the `loadData` function is invoked (see Listing 6.3; we’ve
omitted code lines that you currently don’t need to focus on).

```javascript
JSONModel.prototype.loadData =
    function(sURL, oParameters, bAsync, sType, bMerge, bCache, mHeaders){
        
        this.fireRequestSent({
            url: sURL,
            type: sType,
            async: bAsync,
            headers: mHeaders,
            info: "cache=" + bCache + ";bMerge=" + bMerge,
            infoObject: {
                cache: bCache,
                merge: bMerge
            }
        });
        this._ajax({
            
            success: function(oData) {
                
            }
        });
    }
```

This convenience method is actually a wrapper for an XMLHttpRequest. It gives us a
means to directly influence some of the default behavior inherited from the parent
sap.ui.model.Model class.

We’ll leave the default behavior as is for the moment, however, and look closer at
what exactly happens when the `loadData` function is invoked (see Listing 6.3; we’ve
omitted code lines that you currently don’t need to focus on).

```javascript
JSONModel.prototype.loadData =
    function(sURL, oParameters, bAsync, sType, bMerge, bCache, mHeaders){
        
        this.fireRequestSent({
            url: sURL,
            type: sType,
            async: bAsync,
            headers: mHeaders,
            info: "cache=" + bCache + ";bMerge=" + bMerge,
            infoObject: {
                cache: bCache,
                merge: bMerge
            }
        });
        this._ajax({
            
            success: function(oData) {
                
            }
        });
    }
```

This convenience method is actually a wrapper for an XMLHttpRequest. It gives us a
means to directly influence some of the default behavior inherited from the parent
sap.ui.model.Model class.

We’ll leave the default behavior as is for the moment, however, and look closer at
what exactly happens when the `loadData` function is invoked (see Listing 6.3; we’ve
omitted code lines that you currently don’t need to focus on).

```javascript
JSONModel.prototype.loadData =
    function(sURL, oParameters, bAsync, sType, bMerge, bCache, mHeaders){
        
        this.fireRequestSent({
            url: sURL,
            type: sType,
            async: bAsync,
            headers: mHeaders,
            info: "cache=" + bCache + ";bMerge=" + bMerge,
            infoObject: {
                cache: bCache,
                merge: bMerge
            }
        });
        this._ajax({
            
            success: function(oData) {
                
            }
        });
    }
```
Apart from the actual request, events fire that inform listeners about its status. These events help you determine when you can start working with the data from the model, or when the request fails. You may want to react upon such a failure. These events fire on the model, so you can attach event handlers to them.

The model fires the requestSent event when it sends the request. But the requestCompleted event is more important for our next experiments. This event fires on success or failure of requests.

### 6.1.2 Accessing Model Values

You’ll now learn how to access and manipulate values in a JSON model via the `getProperty` and `setProperty` methods. However, to be sure about when the data should be available in the model and when we can start working on it, we need to attach to the requestCompleted event.

Let’s switch to a smaller and more isolated application example for the remainder of this introduction to data binding, which we’ll call Databinding First Steps; this example will help keep our first tentative steps in the model more transparent. We’ll return to the larger application example from time to time also to implement what we’ve learned.

The `index.html` file for this new, tiny sample application looks like Listing 6.4.

```html
<!DOCTYPE HTML>
<html>
<head>
<meta http-equiv="X-UA-Compatible" content="IE=edge" />
<meta charset="UTF-8">
<title>JSON First Steps</title>
<script id="sap-ui-bootstrap"
src="../resources/sap-ui-core.js"
data-sap-ui-libs="sap.m"
data-sap-ui-theme="sap_bluecrystal"
data-sap-ui-compatVersion="edge"
data-sap-ui-xx-bindingSyntax="complex">
</script>
<script>
// instantiate the model
var oModel = sap.ui.model.json.JSONModel();

//load the data asynchronously
oModel.loadData("service/data.json");
// attach to the requestCompleted event in order
// to know when manipulation of the data is safe:
oModel.attachRequestCompleted(function(oEvent){
  //get and manipulate particular value:
  var sSupplierName = oModel.getProperty("/Suppliers/0/Name");
  sSupplierName = sSupplierName + " Sammamish";
oModel.setProperty("/Suppliers/0/Name", sSupplierName);
});

sap.ui.getCore().setModel(oModel);
var oText = new sap.m.Text({text: "/Suppliers/0/Name"});
oText.placeAt("content");
</script>
</head>
<body class="sapUiBody" role="application">
</div>
</body>
</html>
```
Listing 6.4  First JSON Sample Application

This sample application also includes JSON data in a file called data.json within a folder called service, just as in earlier examples; the file currently resembles Listing 6.5.

```
{
    "Suppliers": [
    {
        "ID": 0,
        "Name": "Exotic Liquids",
        "Address": {
            "Street": "NE 228th",
            "City": "Sammamish",
            "State": "WA",
            "ZipCode": "98074",
            "Country": "USA"
        }
    },
    {
        "ID": 1,
        "Name": "Tokyo Traders",
        "Address": {
            "Street": "NE 40th",
            "City": "Redmond",
            "State": "WA",
            "ZipCode": "98052",
            "Country": "USA"
        }
    }
    ]
}
```

Listing 6.5  JSON Model

Running this application will leave you with a not overly populated screen (see Figure 6.1).

In this application, we'll first retrieve the data from our model file manually by invoking the loadData function. Next, we want to access and manipulate one of the values in the model. We can use the JSON model methods `getProperty` and `setProperty` for this, which are implemented for the JSON model itself. With these methods, you can operate on particular nodes in your model, accessing the values with the path syntax you saw briefly in Chapter 5. You'll learn how to determine the path you need soon, but first let's address what seems to be a timing problem we've encountered here. This problem becomes the most visible when you run this application in the browser and inspect the model object during runtime.

If you want to see the effect for yourself, put a breakpoint right after the line of code in which the model is instantiated, and look at the model object in the debugger of your favorite browser. You can see that there's no data in the model to work on yet (see Figure 6.2).

When we put the next breakpoint into the function, we call on `requestCompleted`. If the request has been successful, you'll immediately see a result like this when you reach that breakpoint (see Figure 6.3).
You can see that now the `data` property of the model has been populated with the corresponding data from the `data.json` file.

Now, let's start working on the data using the methods mentioned previously; the callback function is only invoked when the `requestCompleted` event has been fired:

```javascript
model.attachRequestCompleted(function(oEvent){
    // callback function here
});
```

Let's populate this callback function now. To access any value within the model, you just need the path to the node you want to work with.

Most models in SAPUI5 have a hierarchical structure that allows you to access nodes and values by specifying their respective paths. The accessor syntax separates the model node levels leading to the desired value with slashes:

```javascript
/Suppliers/0/Name
```

This path will access the value of the `Name` property assigned to the first supplier in the model of the sample application. The first part of the path points to the collection of suppliers in the model, the next part allows us to pick the order of the supplier in the model, and the last part leads to the particular property we want to bind our control property to. This path accesses the highlighted values in the model code in Listing 6.6.

### Listing 6.6 Path to a Particular Node in the Model

Note that the leading slash indicates that this path starts at the model root. When a leading slash is present, we're looking at an absolute path. We can also use relative paths, which will become more useful when we talk about binding contexts starting in Section 6.2.

Now that we know the path, we can use it as a parameter in the `getProperty` and `setProperty` methods in the application within the callback function. We just pass the path to the value we want to access as a parameter into the `getProperty` method on the model.
Next, we’ll manipulate the value, and then pass it back into the model again using the 
setProperty method. This method takes the path as the first parameter, along with 
the value that you want to assign, so that the code finally looks like Listing 6.7.

```
MOD6L1AttachRequestCompleted(function(oEvent){
    //get and manipulate particular value:
    var sSupplierName = oModel.getProperty("/Suppliers/0/Name");
    sSupplierName = sSupplierName + " Sammamish";
    oModel.setProperty("/Suppliers/0/Name", sSupplierName);
});
```

**Listing 6.7 Final Callback Function to the requestCompleted Event**

When you save the code and display this page, you’ll see that the value for the first 
supplier indeed has been changed on the model as well as on the UI. The latter hap-
pens because the model itself detects that data has been changed and notifies the UI 
that there is a value to be updated.

---

**getProperty**

The `getProperty` method doesn’t necessarily return a simple value, but will return an 
object or array of objects depending on the nature of the data found at the corre-
spanding path. Pointing the `getProperty` function to the path `/Suppliers/0` will 
return a JavaScript object with properties populated from the JSON model.

---

Next, let’s look at how this particular connection is established because the controls 
within our application are already making use of it.

---

### 6.2 Property Binding

Now that the model is in place, we have the information necessary about its struc-
ture, its content, and which values from the model we want to display in the applica-
tion. As discussed in Chapter 5, one way to get values from a model displayed on 
screen is property binding. Property bindings allow you to connect the properties of 
controls with particular values in a model. After a property is bound, any change to 
the value in a model will also be reflected on the control property. Because we’re cur-
rently using a two-way binding mode, the reverse is true: changes to the control 
property will be reflected in the model.

To establish a binding from a control property to a value in the model, you need two 
parameters:

- The path to the value in the model
- The name of the property that you want to bind

Next, we’ll look at the methods for binding a control’s property before diving into the 
use of data types and defining custom data types.

#### 6.2.1 Methods for Binding a Control’s Property

There are two basic methods you can use to bind a control’s property:

- **Control settings**
  
  You can use the `settings` object in the constructor of a control when you’re using 
  nondeclarative view types, or otherwise just use the `settings` object in the value 
  specified for the attribute when you’re in an XML view, for example.

- **setProperty**
  
  You can use this method at the control anytime in your code to notify the system 
  to which value in the model a property should be bound.

In the next two subsections, we’ll look at these two methods and provide examples of 
their use. We’ll then look at using data types and defining custom data types.

**Control Settings**

In the control settings method of binding a control’s property, curly braces enclosing 
a property value usually indicate that normal string parsing will be suspended, and 
the data-binding mechanisms will take over instead.

In Figure 6.4, we instantiated a simple text control by using the constructor function 
in the JavaScript code:

```
var oText = new sap.m.Text({
    text: "/{Suppliers/0/Name}" });
```

As you can see, we’re passing a value to the constructor that is assigned to the text 
property of the control. This value now contains the path to the value in the JSON 
model. Whenever the JSON model value updates, the value of the control property 
will update as well.
In a two-way binding scenario, this will also work the other way around. Replacing the text control with an input will immediately give the user an opportunity to adjust the model values. We can prove this by adding another control to our sample application: sap.m.Input.

In Listing 6.8, we instantiate an sap.m.Input control and place it in the content div.

```javascript
var oInput = new sap.m.Input({
  value: "{/Suppliers/0/Name}"
});
oInput.placeAt("content");
```

Listing 6.8 sap.m.Input Instantiation

When you add this code to the sample, the application should look like Figure 6.4.

![Figure 6.4 Sample Application with sap.m.Input](image)

You can see that the input field has been assigned the same value as the text above it. With two-way binding, there's more to this than what you see right away. As mentioned previously, both of these controls are now connected to the same value from the model. When we manipulate either one, the model is updated as well. The model then in turn notifies all controls connected to it that the value has changed. This notification will cause a re-rendering of the respective controls and will hence be reflected in every one.

Try entering a different value into the text field, and see what happens after you press the Enter key or tab out of the field. The value from the input is immediately taken over for the text of the text control and displayed above the input field.

### Live Value

The live value of the input field is only reflected in the value attribute of the control when a change event has been triggered. This change event is usually triggered if a user presses a key such as Enter or otherwise leaves the field by clicking or tabbing out of it.

In XML views, the particular properties of a control can be bound in the same way as in the settings of a control instantiation in JavaScript. Simply put, the curly brackets into the attribute at the corresponding tag for your control, like this:

```xml
<Input value="{/Suppliers/0/Name}" />
```

### bindProperty

If you want to bind the property of a control to a model value not on instantiation but at a later point in your code, you can use the bindProperty method that every control in SAPUI5 provides:

```javascript
oInput.bindProperty("value", path: "{/Suppliers/0/Name}");
```

If you want to suppress the automatic data flow and deactivate the two-way binding mode for one particular control, you can specify this at the control as follows:

```javascript
oInput.bindProperty("value", {
  path: "{/Suppliers/0/Name}"
  mode: sap.ui.model.BindingMode.OneWay
});
```

In the latter example, we make use of the object-literal syntax to define the binding, as we showed in the binding specified in the constructor call.

The application programming interface (API) documentation tells us that the bindProperty method is available for every class inheriting from the sap.ui.base.ManagedObject parent class. For many control properties, aggregations, and associations, typed methods are also available. Whether a control offers a typed method for a property is determined by the bindable flag in the control metadata.

In the case of our input, the control has inherited the value property from the InputBase class. When you open the InputBase.js file in the sap.m library, you can see these lines in the constructor, defining the property value as bindable (see Listing 6.9).

```javascript
var InputBase = Control.extend("sap.m.InputBase", /
  ** @lends sap.m.InputBase.prototype */ { metadata: {
    library: "sap.m",
    properties: {
      value: {
        type: "string",
        group: "Data",
        defaultValue: null,
      }},
```
For each property defined with the `bindable` flag, the following methods will be automatically generated and added to the prototype of the class:

- `get[PropertyName]`
- `set[PropertyName]`
- `bind[PropertyName]`
- `unbind[PropertyName]`

For our input, this will result in the following methods being available:

- `getValue`
- `setValue`
- `bindValue`
- `unbindValue`

These methods will allow for the same parameters as the untyped variants. In the previous examples, therefore, we also could have used the lines of code shown here:

```javascript
oInputbindValue({
  path: "{/Suppliers/0/Name}"
  mode: sap.ui.model.BindingMode.OneWay
});
```

The typed functions are simply convenient methods that can make your code easier to write and read—nothing more, nothing less.

Note that complementary to the `bindProperty` method, there is also an `unbindProperty` method that will remove the connection of your control property to the model. When you use this method, the last value set on the property at the control is maintained; it’s no longer updated when the value changes on the model.

The logic used for the property binding on a JSON model is implemented into the `sap.ui.model.json.JSONPropertyBinding` class. Most of its functionality is available for all property bindings throughout SAPUI5 because it’s inherited from a common `sap.ui.model.ListBinding.js` class. All property bindings therefore share the getters and setters for value, type, formatter, external value, and binding mode (see Listing 6.10).

```javascript
var PropertyBinding = Binding.extend("sap.ui.model.PropertyBinding",
/** @lends sap.ui.model.PropertyBinding.prototype */ {
  constructor : function (oModel, sPath, oContext, mParameters) {
    Binding.apply(this, arguments);
  },
  metadata : {
    "abstract" : true,
    publicMethods : [
      "getValue", "setValue", "setType", "getType", "setFormatter",
      "getFormatter", "getExternalValue", "setExternalValue",
      "getBindingMode"
    ]
  }
});
```

### Listing 6.10  Inherited Getters and Setters for Property Bindings

#### 6.2.2 Using Data Types

When you use property binding on certain model properties—for example, float values—you’ll notice that sometimes, it may be useful to make the data type known not only to the model but also to the UI. This can be helpful if, for example, you want to ensure that only a number is inserted into a text field for a price or amount, or only a date for a date of birth.

To achieve this and a certain amount of automatic data formatting, parsing, and validation, you can use data types on the bindings in SAPUI5.

A set of simple types are delivered with SAPUI5 and can be used out of the box:

- `sap.ui.model.type.Integer`
- `sap.ui.model.type.Float`
- `sap.ui.model.type.String`
- `sap.ui.model.type.Boolean`
- `sap.ui.model.type.Date`
- `sap.ui.model.type.Time`
- `sap.ui.model.type.DateTime`
Returning to the input example, we can directly pass a particular data type we want to use as a parameter to the constructor, as shown in Listing 6.11.

```javascript
new sap.m.Input({
  value: {
    path: "/Suppliers/0/Address/ZipCode",
    type: new sap.ui.model.type.Integer({
      minimum: 5,
      maximum: 8
    })
  }
});
```

Listing 6.11 Specifying the Type in the Control Constructor Call

We can also stick to the `bindProperty` approach and simply pass the type into this method as a third parameter, as shown in Listing 6.12.

```javascript
oInput.bindProperty("value", path: "/Suppliers/0/Name",
new sap.ui.model.type.Integer({
  minimum: 5,
  maximum: 8
})
);
```

Listing 6.12 Specifying the Type in the bindProperty Call at Control

SAPUI5 also comes out of the box with several composite types. Composite types can take multiple values as input and then output a formatted representation. The types themselves know how to split, merge, or otherwise change the input to produce a harmonized representation. Here are the composite types:

- `sap.ui.model.type.Currency`
- `sap.ui.model.type.DateInterval`
- `sap.ui.model.type.DateTimeInterval`
- `sap.ui.model.type.unit`

Specifying multiple values for binding is handy, so let’s look at a quick example with the `DateInterval` type. You’ll see later in Section 6.4.2 that our product data includes two date/time properties: `ReleaseDate` and `DiscontinuedDate`. Our data looks like this:

```json
"Products": [{
  "ID": 3,
  "Name": "Havina Cola",
  "Description": "The Original Key Lime Cola",
  "ReleaseDate": "2005-10-01T00:00:00",
  "DiscontinuedDate": "2006-10-01T00:00:00",
  "Rating": 3,
  "Price": 19.9
},
```

This means we could create a text control that displays a nice date interval like this:

```javascript
new sap.m.Text({
  text: {
    parts: [
      {path: "/Suppliers/0/Products/2/ReleaseDate",
       type: new sap.ui.model.type.DateTime()},
      {path: "/Suppliers/0/Products/2/DiscontinuedDate",
       type: new sap.ui.model.type.DateTime()}]
    },
    type: new sap.ui.model.type.DateInterval()
  })
```

The control then renders the Text field with the following text: “Oct 1, 2005 - Oct 1, 2006”.

Let’s investigate the details and functions involved when a data type is used along with the binding.

Each simple data type in SAPUI5 has three functions:

- `formatValue`
  - When a control property is bound to a value on the model, the `formatValue` function will be executed when a change in model data is supposed to be reflected on the control. This throws a `FormatException`.

- `parseValue`
  - The `parseValue` function acts as a formatter in the other direction; it’s executed...
whenever a property value is changed on the UI side, and this change should be transported back into the model. This throws a ParseException.

- `validateValue`
  The `validateValue` function is executed to see if any value constraints were violated. This throws a `ValidateException`.

To react to any exceptions when they are thrown, it’s not necessary to catch the exceptions yourself. Rather, you’re supposed to attach to the events fired by SAPUI5 in such cases via the following controls:

- `attachFormatError`
- `attachParseError`
- `attachValidationError`
- `attachValidationSuccess`

You can easily react to a user’s erroneous input by attaching a `validationError`, for example, which will be fired when a user’s input violates constraints defined by the type.

Let’s look at what happens when we use an integer type within our sample application. The constructor of the `sap.ui.model.type.Integer` takes two optional parameters of the type object—`oFormatOptions` and `oConstraints`—so a new instance can be instantiated like this:

```javascript
var oType = new sap.ui.model.type.Integer(oFormatOptions, oConstraints);
```

When neither of the two parameters are specified, default values for the type are used, as follows:

```javascript
var oInput = new sap.m.Input({
    value: "{/Suppliers/0/Name}"
});
oInput.placeAt("content");
```

We’ll add another input that will take the zip code of the supplier’s address as a value (see Listing 6.13). We expect an integer-type input.

```javascript
var oZipInput = new sap.m.Input({
    value: {
        path: "/Suppliers/0/Address/ZipCode",
        type: new sap.ui.model.type.Integer({
            minimum: 1,
            maximum: 99999999
        })
    }
});
oZipInput.placeAt("content");
```

Listing 6.13 Using a Type with Constraints for an Input

Note that we’ve left the form at options parameter for the `Integer` constructor as an empty object for the moment; we’ll soon see what the parameter is best used for.

Let’s now attach to the corresponding events to be able to give the user feedback when he tries to input something incorrectly (see Listing 6.14).

```javascript
oZipInput.attachParseError(function(oControlEvent){
    alert("Parse Error occurred - this is no integer");
});
oZipInput.attachValidationError(function(oControlEvent){
    alert("Validation error occurred - some constraints were violated: " + oControlEvent.getParameters().newValue + " is not between minimum and maximum");
});
```

Listing 6.14 Attaching to the Validation Events from the Type

When you reload the application and try to insert a number larger than specified in the constraints, you’ll see an alert indicating that you’ve violated the minimum or maximum length. When you insert some nonnumerical characters, you’ll see the alert from the `parseError` instead.

For each error that occurs, the attached event handler takes over. We can access the erroneous value by using a parameter from the `oControlEvent` object that the event handler receives. This provides the user with information about what’s gone wrong before we remove the invalid value.

The new value the user has tried to set is available as the `newValue` parameter. You can also access the old value, data type, and element that initiated the model update. Additional constraints are only taken into account when validation occurs on model update. For the `sap.ui.model.type.Integer`, only the two we used are available.
There are other parameters you can pass to the `Integer` type. These parameters are part of the pattern used when the source value of a control is supposed to be transformed into a string because a control property is of this type. This means that whenever you are outputting a source value of an `Integer` type with a certain number of digits—for example, to a `sap.m.Input` value property—it applies a pattern when it writes to this control property because the property is a string.

For `sap.ui.model.type.Integer`, you can specify the following values, for example:

- `minIntegerDigits: 1 // minimum number of nonfraction digits`
- `maxIntegerDigits: 99 // maximum number of nonfraction digits`

These parameters are part of the `formatOptions` object we left empty in the first attempt to use this data type. When you add some of these format options to the sample, you can see immediately what happens when you try to insert a value that doesn’t follow the pattern.

Let’s specify a minimum number of digits, for example, and change the input field for the zip code as shown in Listing 6.15.

```javascript
var oZipInput = new sap.m.Input({
  value: {
    path: "/Suppliers/0/Address/ZipCode",
    type: new sap.ui.model.type.Integer({
      minIntegerDigits: 5,
      minimum: 1,
      maximum: 99999
    })
  }
});
```

Listing 6.15 Specifying Format Options for a Type

When you reload the application now and try to insert a one-digit number, you’ll see that as soon as the change event on the input is fired, the value is transformed according to the pattern we specified, turning 1 into 00001 (see Figure 6.5).

There are several options for each of the other types. For float values, you can define how many decimals a number should be allowed to have. For the complete list of options, see `sap.ui.model.type` in the API Reference tab of the SAPUI5 Demo Kit.

### 6.2.3 Defining a Custom Data Type

You’ve seen which important methods the data types in SAPUI5 contains. Now we’ll look at how to define our own custom data types.

Assuming you have a data type you want to apply in more than one place in your application, you can write your own custom type by extending the `sap.ui.model.SimpleType` class. You can then realize your particular requirements by implementing the three methods discussed in Section 6.2.2.

Say you need a reusable special data type for phone numbers, following the pattern `+01 1234 456789`. It makes sense to define your own type because you’re expecting a particular format for phone numbers. Because the sample application will also accept user input, it will be useful to have the `validateValue` function from the type ensure that the information entered fulfills the format requirements. Otherwise, it will throw an error and fire the `validationFailed` event.

Let’s first enhance the model with the phone number for the supplier, as shown in Listing 6.16.

```javascript
{
  "Suppliers": [
    {
      "ID": 0,
      "Name": "Exotic Liquids",
      "Address": {
        "Street": "NE 228th",
        "City": "Sammamish",
        "State": "WA",
        "ZipCode": "98074",
        "Country": "USA",
        "PhoneNumber": "+1-123-123-1234"
      }
    }
  ]
}
```

Figure 6.5 Application with Zip Code Input
Listing 6.16 Enhanced JSON Model Data

In the next step, we’ll create a new type by extending the sap.ui.model.SimpleType class and providing the three methods mentioned previously (see Listing 6.17).

```javascript
sap.ui.model.SimpleType.extend("sap.test.phoneNumber", {
    formatValue: function(oValue) {
        return oValue;
    },
    parseValue: function(oValue) {
        return oValue;
    },
    validateValue: function(oValue) {
        if (!/\+\*\D*\[0-9]+([-\[2-9]\d{2}])\D\[\d{2}\]([-\d{2}\d{4})\D/)
            throw new sap.ui.model.ValidateException("phone number must follow the pattern +1-234-567-8900!");
    }
});
```

Listing 6.17 Defining a New Type

For the first two methods, leave the value untouched and return it as is from the model. For the third method, use a regular expression to see if the user input matches the pattern we predefined for phone numbers.

We can now set the type of a new input for the phone number in the sample application to the custom type (see Listing 6.18).

```javascript
var oPhoneInput = new sap.m.Input({
    value: {
        path: "/Suppliers/0/Address/PhoneNumber",
        type: new sap.test.phoneNumber()
    }
});
```

Listing 6.18 Using a Custom Type

Finally, we’ll attach to the validationError event. This time, we’ll use the message from the validation exception we defined ourselves to inform the user about what went wrong (see Listing 6.19).

```javascript
oPhoneInput.attachValidationError(function(oControlEvent){
    alert("Validation error occurred - constraints were violated: " + oControlEvent.getParameter("message"));
});
```

Listing 6.19 Using the Validation Exception to Output an Error to the User

The sample application will now resemble Figure 6.6.

Figure 6.6 Phone Number Added to Sample Application
Now, when a user types in something that doesn’t match the regular expression from the custom type, a new alert will indicate that the user hasn’t followed the predefined pattern.

6.3 Using Formatters

In addition to formatting values by data types, you can also define your own formatters if you want to influence not the value in the model, but rather the output on the UI. A formatter generally makes sense wherever the formatting of a model value is one-way only, as opposed to a type, which is far more powerful, as you’ve learned.

We can pass a formatter function as a parameter into the constructor of the control, as shown:

```javascript
oText.bindProperty("value", "/Suppliers/0/Name", function(sValue) {
    return sValue && sValue.toUpperCase();
});
```

Or we can pass it when calling the `bindProperty` method for the corresponding property (see Listing 6.20).

```javascript
oText = new sap.m.Text({
    value: {
        path: "/Suppliers/0/Name",
        formatter: function(sValue) {
            return sValue && sValue.toUpperCase();
        }
    }
});
```

Listing 6.20 Using Formatter for the sap.m.Text Control

Note that in both cases, the formatter function is an anonymous callback function that will be executed when the view is instantiated.

You can also define a formatter function nonanonymously in a view controller, or, if it will be reused across your application, it may even make sense to have a separate `formatter.js` file in your application that can be referenced by multiple views.

In the first case, all you need to do is implement the function in the controller as shown in Listing 6.21.

```javascript
onInit: function(){
    ...
},
onExit: function(){
    ...
}
toUpperCase: function(sName){
    return sName && sName.toUpperCase();
}
```

Listing 6.21 Formatter Code in the Controller

You can then use this formatter instead of the anonymous function from the previous example, like this:

```javascript
oText.bindProperty("value", "/Suppliers/0/Name", toUpperCase);
```

Returning to our model-view-controller (MVC) application, we can also make use of the formatter by transferring the formatter from Listing 6.21 to the controller of the detail view. Insert the lines from Listing 6.21 into the `Detail.controller.js` file. The detail controller will now look like Listing 6.22.

```javascript
sap.ui.define([
    "sapui5/demo/mvcapp/controller/BaseController"
], function(BaseController) {
    "use strict";
    return BaseController.extend("sapui5.demo.mvcapp.controller.Detail", {
        onInit: function() {
            this.getRouter().getRoute("detail").attachPatternMatched(
                this._onObjectMatched, this);
        }
    });
```

Listing 6.22 Detail Controller Using Formatter
6.3 Using Formatters

In the detail view, we can now use the formatter as shown in Listing 6.23.

```xml
<mvc:View
    controllerName="sapui5.demo.mvcapp.controller.Detail"
    xmlns="sap.m">
    <Page
        id="page"
        navButtonPress="onNavPress"
        showNavButton="true"
        title="Supplier Details">
        <content>
            <ObjectHeader
                id="objectHeader"
                title="{path: 'Name', formatter: '.toUpperCase'}"/>
        </content>
    </Page>
</mvc:View>
```

Listing 6.22 Detail Controller in the MVC Application

In the detail view, we can now use the formatter as shown in Listing 6.23.
**Listing 6.23** Detail View in the MVC Application

The "." in front of the formatter function name tells the framework that it should be looking for a formatter within the current controller. Running the application and loading one of the supplier's details will now produce the screen shown in Figure 6.7. As you can see, the supplier name is now displayed in uppercase letters. The value on the model isn’t touched by this change.

![Figure 6.7 Detail View with the Formatter in Action](image)

However, the greatest reusability will be gained by transferring formatter functions to a separate file that will be available across the application. When writing larger applications, it’s good practice to collect all formatters you want to use in more than one view. Let’s define our own `formatter.js` file for our applications. We’ll put this file into a new `model` folder because it belongs to functionality related to the model and its values. When you transfer the `toUpperCase` function to this file, the complete code will look like Listing 6.24.

```javascript
sap.ui.define([], function() {
    "use strict;"
    return {
        formatter: formatter,
        /* lifecycle methods */
        /* */
        /* */
        /* */
        /* */
        /* */
        /* */
    }
});
```

**Listing 6.24** Formatter.js for the MVC Application

When this formatter is loaded from a controller, it returns an object with the `toUpperCase` function available as member function.

We’ll now make sure the controllers also know about this additional function by including it in the controller definition of the detail view. The controller will now look like Listing 6.25.

```javascript
sap.ui.define(["sapui5/demo/mvcapp/controller/BaseController", "sapui5/demo/mvcapp/controller/formatter" ], function(BaseController, formatter) {
    "use strict;"
    return BaseController.extend("sapui5.demo.mvcapp.controller.Detail", {
        formatter: formatter,
        /* */
        /* lifecycle methods */
        /* */
        /* */
        /* */
        /* */
        /* */
        /* */
        /* */
    });
});
```

**Listing 6.25** Controller for the MVC Application

```javascript
/**
 * Formats a given string to uppercase.
 * @function
 * @param {string} sName string to be formatted
 * @public
 */
toUpperCase: function(sName) {
    return sName && sName.toUpperCase();
};
```
Listing 6.25 Including the Formatter in the Controller

Note that within the controller, the corresponding formatter function has been deleted.

In the view, we now need to access the formatter slightly differently. As mentioned, we now load the formatter file and pass the object it returns to the controller as a parameter. It’s then available as a property called formatter at the controller. We therefore need to change ObjectHeader in Detail.view.xml where we want to use the formatter as shown in Listing 6.26.

```
<ObjectHeader
  id="objectHeader"
  title="{
    path: 'Name',
    formatter: '.formatter.formatUpperCase'
  }"
  number="{ID}"
  <ObjectAttribute
    text="{Address/Country}" />
</ObjectHeader>
```

Listing 6.26 Excerpt from Detail.view.xml

When we’ve done everything correctly, reloading the application will still show the same result in the browser. We haven’t changed anything about the formatting results, but have just made the formatter’s location more reusable. From here, we could include it in multiple controllers across our application.

We’ll now start displaying some more information to our supplier by adding some additional fields to the object header in the detail view. We want to add the address of
the supplier and his phone number now, using the data type from the previous example for the phone number.

Create a types.js file similar to the formatters.js file within the model directory of the MVC application, and copy and paste the data type for the phone number from Section 6.2.3. We'll make use of the define syntax again and will make the module return an object with particular types as properties.

The complete code within the types.js file looks like Listing 6.27.

```javascript
sap.ui.define([[
  "sap/ui/model/SimpleType"
], function (SimpleType) {
  "use strict";
  return {
    /**
     * Data Type for phone numbers.
     * @public
     */
    PhoneNumber: SimpleType.extend("sap.test.phoneNumber", {
      formatValue: function (oValue) {
        return "Phone number: " + oValue;
      },
      parseValue: function (oValue) {
        return oValue;
      },
      validateValue: function (oValue) {
        if (!/\+*\D*\[0-9\]*\-(\[2-9\]\d{2})\D*\[2-9\]\d{2})\D*\d{4}\D/ .test(oValue)) {
          throw new sap.ui.model.ValidateException("phone number must follow the pattern \+1 234-567-8901!");
        }
      }
    })
  };
});
```

Listing 6.27  Separate types.js File to Be Used in the Application

Next, we have to extend the model in this application a little, similar to what we've done before in the smaller sample application, to have the information ready for display. The model.json file will now look like Listing 6.28.

```json
{"Suppliers": ["ID":0, "Name":"Exotic Liquids", "Address":{
  "Street":"NE 228th", "City":"Sammamish", "State":"WA", "ZipCode":"98074", "Country":"USA", "PhoneNumber": "+1-123-123-1234"
}, [...],
"ID":1, "Name":"Tokyo Traders", "Address":{
}, [...]
},
```

Listing 6.28  Model Excerpt: Added Phone Number to Suppliers

Now we need to tell the controller again where to find the type we want to use in the view. As with the formatter, simply add the types to the modules loaded in the define section of the controller (see Listing 6.29).
sap.ui.define([
    "sapui5/demo/mvcapp/controller/BaseController",
    "sapui5/demo/mvcapp/model/formatter",
    "sapui5/demo/mvcapp/model/types"
], function (BaseController, formatter, types) {
    "use strict";
    return BaseController.extend("sapui5.demo.mvcapp.controller.Detail", {
        formatter: formatter,
        types: types,
    });
});

Listing 6.29 Excerpt from Detail.controller.js

Finally, we can combine the new model values with the data type and formatter to
enhance the detail view. Add the additional object attribute to the view as shown:

```xml
<ObjectAttribute text="{
    path: 'Address/PhoneNumber',
    type: '.types.PhoneNumber'
}" />
```

When you run the application and pick a supplier from the list, it should now look like Figure 6.8.

![Figure 6.8 Detail View with Type for Phone Number](image)

6.4 Aggregation Binding

So far, we’ve looked at the property bindings within the application, but we can also
use aggregation binding to automatically generate items for a control aggregation
from model data.

An example of this is given in Master.view.xml, as shown in Listing 6.30.

```xml
<table
    id="table"
    width="auto"
    class="sapUiResponsiveMargin"
    items="/Suppliers"
    noDataText="No data"
    growing="true"
    growingScrollToLoad="true">
</table>

```

Listing 6.30 Excerpt from Master View Showing Aggregation Binding

The items aggregation for the table is filled with one item per model node referenced
by the path in the items attribute in Listing 6.30. For each item, a template needs to
be applied. This template is then cloned for each value from the model, setting the
binding path of the aggregated element to the corresponding model node.

As noted in Chapter 5, when you want to instantiate the parent control within a Java-
Script view and pass the aggregation binding parameters directly to the control con-
structor, use the code shown in Listing 6.31.

```javascript
var aColumns = [
    new sap.m.Column({
        header: new sap.m.Text({
            text: "ID"
        })
    }),
];
```
new sap.m.Column(
    {header: new sap.m.Text({
      text: "Name"
    })
  });

var oTemplate = new sap.m.ColumnListItem(
  {cells: [
    new sap.m.ObjectIdentifier({
      text: "{ID}"
    }),
    new sap.m.ObjectIdentifier({
      text: "{Name}"
    })
  ]}
);

var oTable = new sap.m.Table(
  {columns: aColumns,
    items: {
      path: "/Suppliers",
      template: oTemplate
    }
  });

Listing 6.31 Aggregation Binding in JavaScript

Within the XML view, this template is declaratively specified within the <items></items> aggregation boundaries.

Some controls provide a default aggregation, which you can identify from the control metadata in the corresponding control class. If you look at sap.m.ObjectHeader in the sample application, you’ll see the following bolded line in its metadata (see Listing 6.32):

Listing 6.32 Default Aggregation Defined in the Metadata of a Control

These default aggregations are automatically used when you don’t specify the aggregation within an XML view (see Listing 6.33).

Listing 6.33 Excerpt from Detail.view.xml without Explicit Usage of Default Aggregation

The code from Listing 6.33 and from the following Listing 6.34 will produce the very same Document Object Model (DOM), even if the latter contains extra tags for opening and closing the default aggregation.

Listing 6.34 Aggregation Binding
Listing 6.34  Excerpt from Detail.view.xml with Explicit Usage of Default Aggregation

It’s a good practice to use the default aggregations without explicitly using the tags to indicate them. The corresponding boundaries only need to be given for nondefault aggregations.

Note that controls created within an aggregation binding take part in the lifecycle of their parents. They’re instantiated when the parent controls are instantiated and are destroyed along with their parents.

We’ll enhance our sample application with another aggregation binding on the detail view now by adding another sap.m.Table control that contains all products offered by a particular supplier.

Let’s add this table to the Detail.view.xml file, right below the closing </ObjectHeader> tag, as shown in Listing 6.35.

Listing 6.35  Products Table in Detail View

When you add this aggregation, there will be one item instantiated for every product associated with the supplier in the model. The application should now look like Figure 6.9.

Figure 6.9  Detail View with Products
6.4.1 bindAggregation

Instead of declaring the aggregation template and binding within the XML view, you can use the bindAggregation method on the parent control. This method, just like the bindProperty method, can be used when you want to defer binding the aggregation to a later point in your code.

The function takes two parameters: the name of the aggregation and an oBindingInfo object. This object can contain several parameters itself; we’ll look at some of them in the following subsections.

As for the bindProperty method, some controls also offer typed methods for aggregation binding. In the control metadata, this is determined by the bindable flag at the aggregation definition. Note that these methods are generated automatically by the sap.ui.base.ManagedObject parent class but can be overridden at the particular control.

In the case of sap.m.Table in the previous example, we could also have called the following:

```javascript
oTable.bindItems("Products", oBinding);```

Complementary to the bindAggregation method, there is also an unbindAggregation method, analogous to the unbindProperty method you already know.

6.4.2 Using a Factory

One major advantage of the programmatic approach is that you can use a factory along with the bindAggregation function. A common use case for a factory is the dynamic creation of your UI, for example, conditionally displaying child controls.

Let’s assume that in our application we want to adjust the display for products with certain ingredients. This part of the exercise will be slightly off the path the sample application will take during the remainder of the chapter, so please make sure you save a version of your work before you implement the next steps. We’ll get back to the running example after this section.

Let’s first enhance the JSON model to reflect this information, then remove the declarative aggregation binding within our view, and instead programmatically implement the products aggregation we want to show in the controller (see Listing 6.36).

```javascript
{ 'Suppliers': [{
  "ID": 0,
  "Name": "Exotic Liquids",
  "Address": {
    "Street": "NE 228th",
    "City": "Sammamish",
    "State": "WA",
    "ZipCode": "98074",
    "Country": "USA",
    "PhoneNumber": "001555789789789"
  },
  "Location": {
    "type": "Point",
    "coordinates": [-122.03547668457, 47.6316604614258]
  }
},
"Products": [{
  "ID": 1,
  "Name": "Milk",
  "Description": "Low fat milk",
  "ReleaseDate": "1995-10-01T00:00:00",
  "DiscontinuedDate": null,
  "Rating": 3,
  "Price": 3.5
}, {
  "ID": 2,
  "Name": "Vint soda",
  "Description": "Americana Variety - Mix of 6 flavors",
  "ReleaseDate": "2000-10-01T00:00:00",
  "DiscontinuedDate": null,
  "Rating": 3,
  "Price": 3.5
}]
}
<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Description</th>
<th>ReleaseDate</th>
<th>DiscontinuedDate</th>
<th>Rating</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pink Lemonade</td>
<td>36 Ounce Cans (Pack of 3)</td>
<td>2006-11-05T00:00:00</td>
<td>null</td>
<td>3</td>
<td>18.8</td>
</tr>
<tr>
<td>2</td>
<td>Cranberry Juice</td>
<td>16-Ounce Plastic Bottles (Pack of 12)</td>
<td>2006-08-04T00:00:00</td>
<td>null</td>
<td>3</td>
<td>22.8</td>
</tr>
<tr>
<td>3</td>
<td>Havina Cola</td>
<td>The Original Key Lime Cola</td>
<td>2005-10-01T00:00:00</td>
<td>2006-10-01T00:00:00</td>
<td>3</td>
<td>20.9</td>
</tr>
<tr>
<td>4</td>
<td>Fruit Punch</td>
<td>Mango flavor, 8.3 Ounce Cans (Pack of 24)</td>
<td>2003-01-05T00:00:00</td>
<td>null</td>
<td>3</td>
<td>22.99</td>
</tr>
<tr>
<td>5</td>
<td>Cranberry Juice</td>
<td>16-Ounce Plastic Bottles (Pack of 12)</td>
<td>2006-08-04T00:00:00</td>
<td>null</td>
<td>3</td>
<td>22.8</td>
</tr>
<tr>
<td>6</td>
<td>Pink Lemonade</td>
<td>36 Ounce Cans (Pack of 3)</td>
<td>2006-11-05T00:00:00</td>
<td>null</td>
<td>3</td>
<td>18.8</td>
</tr>
<tr>
<td>7</td>
<td>Green Lemonade</td>
<td>36 Ounce Cans (Pack of 3)</td>
<td>2006-11-05T00:00:00</td>
<td>null</td>
<td>3</td>
<td>18.8</td>
</tr>
</tbody>
</table>

### Listing 6.36 Altered JSON Model to Display Additional Product Information

You can see that some products have now been enhanced with information about allergens they contain. To clearly show this information in the list of products, we’ll add a control to the list item when this information is present in the model.

We’ll first remove part of the table definition from the previous example again so that the detail view code now looks like Listing 6.37.
When you reload the application now in the browser, you’ll still see the table itself, but it won’t display any line items; it will only show the noDataText.

In the detail controller, in the _onObjectMatched function, we’ll now make a call to another private function, like this:

createProductsAggregation(sObjectPath);

Next, we have to implement this function, which will create the items aggregation in the table. In this function, we want to use a particular control, namely, VerticalLayout from the sap.ui.layout library. We need to make sure that the controller knows this control, so we include it in the define part of the controller, as in Listing 6.38.

_createProductsAggregation: function() {
  var oTable = this.getView().byId("table");
  oTable.bindAggregation("items", "Products", function(sId, oContext) {
    var sAllergens = oContext.getProperty("Allergens");
    oColumnListItem.addCell(new sap.m.ObjectIdentifier({
      text: "{ID}"
    }));
    if (sAllergens) {
      // we have allergens, so we provide a VerticalLayout instead
      // of just displaying the product name. It then takes
      // the product name plus the allergens into
      // its own content aggregation
      oColumnListItem.addCell(new VerticalLayout({
        content: [
          new sap.m.Text({
            text: "{Name}"
          }),
          new sap.m.Text({
            text: "{Allergens}"
          })
        ]
      }));
    } else {
      // we don’t have allergens, so we can
      // just display the product name
      oColumnListItem.addCell(new sap.m.Text({
        text: "{Name}"
      }));
    }
  });
}

Listing 6.37  Detail View without Items Aggregation for the Table

Listing 6.38  Aggregation Binding
// no allergens there, we display the name as usual
oColumnListItem.addCell(new sap.m.ObjectIdentifier({
text: "(Name)"
}));

oColumnListItem.addCell(new sap.m.ObjectNumber({
    number: "(Price)",
    unit: "USD"
}));
return oColumnListItem;
});

Listing 6.38 Binding Items Aggregation Using a Factory

You can see in the code that we instantiate the controls within the aggregation by passing a factory function into the bindAggregation method. When the parent control is instantiated and the aggregation is created, the factory executes for every entry in the model. Depending on whether a product contains an allergen or not, we either display just the name of the product as usual or make a column containing the sap.ui.layout.VerticalLayout control type, which will allow us to specify more than one content control for the vertical layout.

We then use two text controls that both will go into the content aggregation of VerticalLayout. This shows the power of using factories with aggregation binding because you can decide whether you want to display the same information a bit differently depending on your data or even exchange a complete control for another.

When you reload the page, you can see that some products carry additional information within another control inside the description cell, and other products do not.

Note that the controls generated for the aggregation via the factory behave like any other aggregation element, meaning they also take part in the lifecycle of their parent.

With the newly added items, the application’s detail view should now look like Figure 6.10.

6.5 Element Binding

You’ve seen that when we switch from the master view to the detail view, the information displayed depends on which supplier is picked. In the aggregation binding, we’ve now also made use of a relative binding path to the products. To make this relative binding path adapt to the chosen supplier, the parent of the List control that contains the aggregation needs to know how to resolve this path correctly. To achieve this, we’ll use a third type of binding: element binding. An element binding allows you to create a context for all relative bindings inside an element. To understand what this means, let’s look at an example.

Within the detail view are some property bindings that will display information about the chosen supplier. The same applies to sap.m.List, where we’ve defined our items aggregation so that it displays the products of a particular supplier. None of these paths begin with a slash, which indicates that you’re looking at a relative binding path.
Let's look at where the information about how to resolve these paths is coming from. When you inspect the code instantiating the detail controller, you can see an element binding used in the method that handles the `routeMatched` event (see Listing 6.39).

```javascript
_onObjectMatched : function (oEvent) {
  var sObjectPath = '/Suppliers/' + oEvent.getParameter('arguments').ID;
  this._bindView(sObjectPath);
},

_bindView : function (sObjectPath) {
  var oView = this.getView();
  oView.bindElement(sObjectPath);
}
```

Listing 6.39  Binding the View to an Element

When someone clicks on a supplier in the master view, the information about the supplier's ID is passed to the detail view via routing. The parameter is then available at the event that is passed into the corresponding event handler.

We can hence determine which supplier the detail view is supposed to display by putting the path information together, combining the name of the collection in the model with the ID from the event parameter. This is done within the `onObjectMatched` event.

Next, we use this information within our private `_bindView` method, where we explicitly call `bindElement` on the view. The `bindElement` function then creates a binding context for an SAPUI5 control and receives the model name as well as the path to an item in a configuration object. This will trigger an update of the UI controls that we connected with the model fields.

When the element binding is set to a different binding context—for example, if we switch to a different supplier—all the controls inside the view will be updated with the corresponding values again.

The `bindElement` function is inherited from the `sap.ui.base.Element` class and can be invoked on any child. As with the other binding types, there is also an `unbindElement` function that can be used to remove the specific binding context created at an element.

---

### 6.6 Expression Binding and Calculated Fields

You've now seen how values from the models can be used for different binding types and how they can be formatted, either using formatter functions or by using a data type. However, so far we've only operated on one value at a time.

If you want to combine several values from the model in one control property, that's where expression bindings or calculated fields come into play. Let's combine some of the supplier's information in the detail view in one place using the city and the state as an example.

#### 6.6.1 Calculated Fields

Calculated fields can be passed into the property you want to bind similarly to the usual binding. The only difference is that you now need to pass more than one part of the value, which is why the implementation lets you specify these as just that—parts.

In the detail view, let's change `ObjectHeader` with the supplier information as in Listing 6.40.

```xml
<ObjectHeader
  id="objectHeader"
  title="{
    path: 'Name',
    formatter: '.formatter.formatUpperCase'
  }
  number="ID: {ID}"
>
<attributes>
  <ObjectAttribute
    text="{Address/City}, {Address/Country}"
  />
  <ObjectAttribute
    text="{
      path: 'Address/PhoneNumber',
      type: '.types.PhoneNumber'
    }" />
</attributes>
</ObjectHeader>
```

Listing 6.40  Object Header in Detail View
As you can see, we’re simply passing the two values into the detail view, which will make SAPUI5 perform an automatic concatenation of the two. The framework creates a composite binding automatically, and this composite binding will take care of combining the data and providing it to the UI. When you reload the page now, you’ll see that this behavior makes the calculated field appear, as shown in Figure 6.11.

![Figure 6.11 Object Header: Calculated Field](image)

When you want to use this kind of binding within JavaScript code, you need to explicitly specify the parts of the binding, as in Listing 6.41.

```javascript
var oObjectHeaderAttribute = new sap.m.ObjectAttribute({
  text: {
    parts: [
      {path: "Address/City", type: new sap.ui.model.type.String()},
      {path: "Address/Country"}
    ],
    formatter: function(sCity, sCountry){
      if (sCity && sCountry) {
        return sCity + ", " + sCountry;
      }
    }
  }
});
```

**Listing 6.41 Creating an Object Header**

You can specify a different type for each path, but it's not mandatory. In this case, however, you’ll have to attend to the formatting yourself, which is why you can also pass in a custom formatter. If you don’t do so, the framework will use the default concatenation mechanism, which combines the two strings.

Passing in a formatter is easy. You either pass it within the object literal, as in Listing 6.42, or you can pass it as a parameter into the typed or untyped bindProperty method at the control.

```javascript
oObjectHeaderAttribute.bindText({
  parts: [
    {path: "Address/City", type: new sap.ui.model.type.String()},
    {path: "Address/Country"}
  ],
  formatter: function(sCity, sCountry){
    if (sCity && sCountry) {
      return sCity + ", " + sCountry;
    }
  }
});
```

**Listing 6.42 Passing the Formatter to the Binding Method**

You can combine as many values into a calculated binding as you like, and only need to worry about the kind of concatenation you want to see within this function. When the layout becomes more complex, however, you’ll want to fall back to using more than one control, most likely in combination with some kind of SAPUI5 layout control. You shouldn’t try to output HTML code here because it will be hard to format and may even break the structure of your page.

**Special Note**

When using calculated fields binding, you need to be aware that it only works one way. There is currently no mechanism that will allow a two-way binding for such fields because the algorithm that would be needed to cover all potential use cases for backward calculation would become impossibly complex for the framework to foresee and implement.

When you want to give your users the opportunity to change the value of an input with a calculated field, you need to implement the logic that will reflect these changes into the model yourself.

### 6.6.2 Expression Binding

Expression binding is an additional data binding feature in SAPUI5 that can be used instead of a formatter if you need your binding to be based on certain conditions.
Whenever you find yourself programming a rather trivial formatter that will only compare two values, you should think about whether expression binding might also do the job.

You could think about rewriting the code from the factory in Section 6.4.2 and simply use an expression to determine whether you want to have the text control visible that shows the allergens for a product. In that case, you would have to simply alter the view a little instead of enhancing the controller. The items part of your detail view would then look like Listing 6.43.

```xml
<items>
  <ColumnListItem>
    <cells>
      <ObjectIdentifier
        text="{ID}"/>
      <layout:VerticalLayout>
        <Text text="{Name}"/>
        <Text text="{Allergenics}" visible="{= ${Allergenics} !== '' }"/>
      </layout:VerticalLayout>
      <ObjectNumber
        number="{Price}" unit="USD" />
    </cells>
  </ColumnListItem>
</items>
```

Listing 6.43 Using Expression Binding Instead of Formatting

Expression binding can also express multiconditional Boolean logic. Take the example from Listing 6.43, and imagine that you want to show users when a product with allergens will be discontinued. Because much of the data has a null value for the DiscontinuedDate, we only want to show the discontinued date for allergen-containing products when such a date is available. We can use & and to escape the JavaScript double ampersand conditional AND logic to create a Boolean expression with two conditions:

```xml
<Text text="{DiscontinuedDate}" visible="{= ${Allergenics} !== '' &amp;&amp; ${DiscontinuedDate} !== null }"/>
```

Finally, expression binding can also make use of other JavaScript built-in functions. Again, take our example from Listing 6.43 and imagine that you want a special warning text to appear when milk is among the allergens present in the product. You could add a text field with visibility conditional on the RegExp function matching a string in the Allergenics property of the data:

```xml
<Text text="Milk Warning!" visible="{ RegExp('milk', 'i').test({Allergenics}) }"/>
```

Expression binding is only recommended when the expressions aren’t overly complex. The downside of this approach is that the vertical layout would still be rendered as such, even if one of the controls inside it was invisible and would hence pollute the DOM with unnecessary elements. For more complex use cases, the other implementation options we covered—formatters or factories for aggregation bindings—may be more suitable.

There are several operators you can use within expressions—equality operators, multiplicative and additive operators, and several more. See the API Reference tab in the SAPUI5 Demo Kit for details.

### 6.7 Resource Models and Internationalization

A resource model is a special type of client-side model used for internationalization purposes within SAPUI5. It acts as a wrapper for resource bundles. A resource bundle is a file used to store the translatable texts used within your application. These files have a simple internal structure, which contains a set of name/value pairs separated by an equals sign. For instance:

```plaintext
appTitle=Suppliers And Products
```

It’s important to understand that within a `properties` file, there is no concept of an object hierarchy. The structure is entirely flat.

In this section, we’ll look at the file locations, file naming conventions, and code page for resource bundles. We’ll then look at using a resource model in our example application.

#### 6.7.1 File Location

Resource bundle files are typically placed into the `i18n` folder within your application’s directory structure. All these files end with the suffix `properties`, which is why they are also referred to as “dot properties” files.
When we inspect our sample application closely, you'll see that some texts aren't part of the business data supplied in the JSON model. Instead, there are texts that act as labels or titles that should be translated should the user decide to run the application in a different language.

### 6.7.2 File Naming Convention

SAPUI5 resource bundle files follow the same naming convention as is used for Java resource bundles. Any file ending in `.properties` is assumed to contain simple name-value pairs; the language for these files will be determined by locale identifiers present in the file name. Table 6.1 shows an example of a few resource bundle files.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i18n.properties</code></td>
<td>No language is defined; therefore, the default language is assumed. Exactly which language acts as the default is determined by the larger context within which your application is being run. For instance, if your default corporate language is English, then the texts contained within this file will be in English. However, this is an assumed default and can't be derived from the file name itself.</td>
</tr>
<tr>
<td><code>i18n_en.properties</code></td>
<td>This file is a resource bundle containing English text.</td>
</tr>
<tr>
<td><code>i18n_en_GB.properties</code></td>
<td>This file is a resource bundle containing text in British English.</td>
</tr>
<tr>
<td><code>i18n_de.properties</code></td>
<td>This file is a resource bundle containing German text.</td>
</tr>
<tr>
<td><code>i18n_de_AT.properties</code></td>
<td>This file is a resource bundle containing text in the Austrian dialect of German.</td>
</tr>
</tbody>
</table>

Table 6.1 Resource Bundle Naming Conventions

When determining which language resource file to open, the browser always requests the most specific file first. For instance, if your browser is set to a UK locale, the browser will request `i18n_en_GB.properties`. If this file can't be found, then the browser requests the next most-specific file, which in this case would be `i18n_en.properties`. If this file can't be found, then it removes any locale values from the file name and simply requests the resource bundle containing text in the default language—`i18n.properties`. This is known as the language determination fallback process.

### 6.7.3 Code Page

It’s important to understand that the specification for resource bundle files states that all text will belong to the ISO-8859-1 (LATIN-1) code page. This has two important consequences:

- The use of the LATIN-1 code page immediately assumes the text belongs to a left-to-right language.
- Text belonging to languages that require the use of some other code page, for instance, far eastern languages, must be entered into the resource bundle file using Unicode encoding and not using the language's native script.

For instance, if you want to enter Hebrew text into a resource bundle file, then you must account for the fact that (1) Hebrew is a non-LATIN-1 language, and 2) it’s a right-to-left language. Hebrew for “Hello World” translates into English as “Shalom Olam.” To remedy this, first, you must create a resource bundle file called `i18n_he.properties`; however, it’s incorrect to enter the Hebrew script directly into this file. In spite of the fact that the Hebrew script is perfectly correct, the following example is incorrect:

```
Hello World =RAINBOW
```

Instead, you should enter the following:

```
Hello World =RAINBOW
```

Not only have the Hebrew characters been converted to their hexadecimal Unicode values, but the character order of the Hebrew phrase has had to be reversed. This is important, because all text in a `.properties` file is assumed to be specified in left-to-right format—even if such text belongs to a right-to-left language!

The browser will automatically reverse the order of characters within this string so that it’s rendered correctly. See the `dir=”RTL”` attribute for UI controls for more information.

### 6.7.4 Using a Resource Model

Let’s go through with our example application now and replace all texts that you think should be translatable with values from a `.properties` file.

First, you need to create the `i18n` directory within your application root folder. Next, you need to create the `.properties` file within the `i18n` directory. Within this directory, you can have `.properties` files for every language you want your application to be
translatable into. A typical .properties file for our application could look like Listing 6.44.

```properties
notFoundTitle=Not Found
notFoundText=The resource wasn’t found
backToMaster=Back to Master List
masterViewTitle=Supplier Overview
tableNoDataText=No data
masterTableTitle=Supplier
tableIDColumnTitle=ID
tableNameColumnTitle=Name
detailTitle=Supplier Detail
ID=ID
```

Listing 6.44 Default Properties File for the Example Application

In the next step, we need to instantiate the resource model. We’ll do this in the component because this model should be available across the application.

There are two areas in the component code that we’ll change. The first is part of the component metadata—namely, the configuration. Define the path to the model, as shown in Listing 6.45.

```javascript
metadata : {
  "rootView": "sapui5.demo.mvcapp.view.App",
  "dependencies": {
    "minUI5Version": "1.28.0",
    "libs": [ "sap.ui.core", "sap.m" ]
  },
  "config": {
    "i18nBundle": "sapui5.demo.mvcapp.i18n.i18n",
    "serviceUrl": "webapp/service/data.json"
  }
},
```

Listing 6.45 Excerpt from Component Metadata

The second area is the init method of the component. Here, access the configuration value, instantiate the resource model from the path you’ve retrieved, and then set it to the component, which will in turn propagate it to its children, that is, the views (see Listing 6.46).

```javascript
init : function () {
  var mConfig = this.getMetadata().getConfig();
  // call the base component’s init function
  UIComponent.prototype.init.apply(this, arguments);

  // set the internationalization model
  this.setModel(new ResourceModel({
    bundleName : mConfig.i18nBundle
  }), "i18n");
},
```

Listing 6.46 Init Function in the Component

We can now replace the hard-coded strings in the application with values from the model. Because this is a named model, we can access the values within the bindings with a prefix, such as i18n>detailTitle, as you can see in Listing 6.47.

```xml
<mvc:View controllerName="sapui5.demo.mvcapp.controller.Detail"
  xmlns:mvc="sap.ui.core.mvc"
  xmlns:layout="sap.ui.layout"
  xmlns="sap.m">
  <Page id="page"
    navButtonPress="onNavPress"
    showNavButton="true"
    title="{i18n>detailTitle}">
    <content>
      <ObjectHeader
        id="objectHeader"
        title="{
          path: 'Name',
          formatter: '.formatter.formatUpperCase'
        }"
        number="{i18n>ID}: {ID}">
Listing 6.47 Excerpt from the Detail View with the Resource Model
As you can see from the object header carrying the ID value from the supplier, it’s also possible to combine values from two different models—say, from the resource model and business data model—into one binding.

You should go through the application now and replace the remainder of the hard-coded strings to make the application fully translatable. If you wish, you can add another .properties file to the i18n folder afterwards, using the suffix for your desired target language, and translate the properties from your source file.

When you switch the application language in the browser, you can see how certain strings are replaced with the corresponding language. As an example, we’ve provided an additional German .properties file, which will be called i18n_de.properties (see Listing 6.48).

Listing 6.48 i18n.properties File with Translatable Texts for the Application

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>notFoundTitle</td>
<td>Nicht gefunden</td>
</tr>
<tr>
<td>notFoundText</td>
<td>Die Resource wurde nicht gefunden</td>
</tr>
<tr>
<td>backToMaster</td>
<td>Zurück zur Master-Liste</td>
</tr>
<tr>
<td>masterViewTitle</td>
<td>Anbieter-Überblick</td>
</tr>
<tr>
<td>tableNoDataText</td>
<td>Keine Daten</td>
</tr>
<tr>
<td>tableNameColumnTitle</td>
<td>Name</td>
</tr>
<tr>
<td>tablePriceColumnTitle</td>
<td>Preis</td>
</tr>
<tr>
<td>detailTitle</td>
<td>Anbieter-Detail</td>
</tr>
<tr>
<td>detailTableHeader</td>
<td>Anbieter-Produkte</td>
</tr>
<tr>
<td>ID</td>
<td>ID</td>
</tr>
<tr>
<td>detailFilterLabel</td>
<td>Produkte Filtern</td>
</tr>
</tbody>
</table>

You can switch between different languages by specifying the language parameter for your application in the browser’s address bar, like this:  

```
```

To identify a language, the framework generally uses a language code of type string. Typically, this includes strings such as the BCP-47 standard, de, en-US, zh-Hans-CN, and so on, but the Java locale syntax and SAP proprietary language codes also are accepted. See the Developer Guide for more details.

The current language is determined from several sources. The following list shows all sources in the reversed order of their priority, which means the last in the list gets precedence over the others:

- Hard-coded SAPUI5 default locale “en”
- Potentially configured browser language (window.navigator.browserLanguage); the language of the operating system for Internet Explorer
- Potentially configured user language (window.navigator.userLanguage); the language in the Region settings for Internet Explorer
- General language information from the browser (window.navigator.language)
- Android: Language contained in the user agent string (window.navigator.userAgent)
- Locale configured in the application coding (jsdoc:symbols/sap.ui.core.Configuration)
- Locale configured via URL parameters

If you want to use the API to access the current language, you can do so via the corresponding method at the configuration object:

```
var sCurrentLocale = sap.ui.getCore().getConfiguration().getLanguage();
```

If you need to access a value from a resource model directly from the code—for example, if you need to make use of a value within a formatter—you can use the jQuery.sap.resources module. This module contains an API that allows you to retrieve a resource bundle, depending on the URL and locale you pass to the module.

Note, however, that you need to make sure the module is indeed present and loaded in your application. You can do so by explicitly requiring the module as follows:

```
jQuery.sap.require("jquery.sap.resources");
```

Then, you can use the module to retrieve the desired resource bundle:

```
var oBundle = jQuery.sap.resources({url: sUrl, locale: sLocale});
```

For more information, see jQuery.sap.resources in the API Reference tab of the SAPUI5 Demo Kit. After you have the resource bundle loaded within your application, you can use the getText method on the bundle to access any particular text by its key:

```
var sText = oBundle.getText(sKey);
```
6.8 View Models and the Device Model

The last two models covered in this chapter aren’t actually different model types. They can be any kind of model, but we’ll again be making use of JSON models within the example application.

6.8.1 Using View Models

A view model can keep track of the states of different elements on your view. It can establish a connection between values calculated by application logic within your controller and the UI elements that need to be enabled/disabled, visible/invisible, or active/inactive, to name just a few examples.

Whenever you can’t pass a value from your business data model to your view using the typical binding methods but have to calculate values based on several conditions, it’s a good practice to use a view model. It helps keep the controller logic and view declaration separate because it prevents the controller from needing to know something about the particular controls within the view and their respective states.

Let’s look at an example to make this clearer. We’ll implement a function that will allow us to flip through the suppliers within the application. When a user enters the detail view for one supplier, he’ll see a forward and/or backward button in the upper-right corner of the view. When the user clicks one of the buttons, it will trigger a routing event and send him to the corresponding next or previous supplier.

The detail view in the application will now look like Listing 6.49.

```xml
<mvc:View controllerName="sapui5.demo.mvcapp.controller.Detail"
xmlns:mvc="sap.ui.core.mvc"
xmlns:layout="sap.ui.layout"
xmlns="sap.m">
  <Page id="page"
    navButtonPress="onNavPress"
    showNavButton="true"
    title="{i18n>detailTitle}"
  >
    <subHeader>
      <Toolbar>
        <ToolbarSpacer/>
        <Button icon="sap-icon://slim-arrow-up" press="onPageUp" />
        <Button icon="sap-icon://slim-arrow-down" press="onPageDown" />
      </Toolbar>
    </subHeader>
    <content>
    ...
  </Page>
</mvc:View>
```

Listing 6.49  Altered Page Header in the Detail View

Note that we’re making use of special icons for the two buttons, which were designed for paging purposes. (ToolbarSpacer makes sure the buttons are right-aligned). The challenge now is to enable and disable these buttons depending on whether there is a next or previous supplier. When the first supplier in the list is displayed, it doesn’t make sense to have an active Back button.

We could go forward and use the methods we’ve worked with previously to retrieve information about the position of the supplier in the list. However, to dynamically enable and disable the buttons from the controller, we’d then have to write code into the controller that would update the state of the button whenever the binding context in the view changes to a different supplier. In addition, the controller would have to know exactly which buttons are in the view by accessing them by their IDs, for example. To decouple this to make our code as reusable and maintainable as possible, it’s a better practice to keep the controller logic independent of knowing which buttons are in the view.

Fortunately, there’s an easy way out of this dilemma: create another model in the onInit event of the detail view. This model is a normal JSON model, which we’ll set to the view as a named model. It doesn’t hold much data (in this particular case, only two values are necessary), which is why we can specify the model’s values right where the model is instantiated.

The onInit handler in the detail view will now look like Listing 6.50.

```javascript
onInit : function () {
  this.getRouter().getRoute("detail").attachPatternMatched(
    this._onObjectMatched, this);
  var oModel = new sap.ui.model.json.JSONModel({
    buttonPrev: false,
    buttonNext: false
  });
  this.getView().setModel(oModel, "viewModel");
},
```

Listing 6.50  Instantiating a View Model
You can see that the two properties hold Boolean values, which we’ll use to bind the enabled properties from the buttons. We can do this by using the normal property-binding syntax in the view. Therefore, let’s add another property to each of the two buttons—namely, the enabled property—and bind these to the view model, as shown in Listing 6.51.

```xml
<Toolbar>
    <ToolbarSpacer/>
    <Button icon="sap-icon://slim-arrow-up" press="onPageUp" enabled="[viewModel>/buttonPrev]" />
    <Button icon="sap-icon://slim-arrow-down" press="onPageDown" enabled="[viewModel>/buttonNext]" />
</Toolbar>
```

Listing 6.51 Binding to the View Model

When you reload the page, you’ll see that the buttons are both disabled now, and no matter which supplier you go to, you won’t be able to click on either one (see Figure 6.12).

![Supplier Detail with Object Header](image)

Figure 6.12 Supplier Detail with Object Header

What we’re still missing is the logic that will update the view model whenever the bindingContext element for the view changes and the functionality for the navigation.

We need to implement this logic in such a way that it will be triggered whenever the bindElement function on the view is invoked, which happens in the onObjectMatched method on the detail view.

For the sake of code clarity, we’ll write our own function to determine the correct state and then call this function from within the onObjectMatched method. The new function needs to retrieve the current position of a supplier in the model and calculate the state the buttons should be in based on this information. We’ll also store the information about the position in the view model because it will be reused in the navigation event handler.

We can retrieve the information and update the view model by adding the lines of code in Listing 6.52 to the detail controller.

```javascript
/**
 * Updates the view model according to whether there are previous and/or next suppliers.
 */
@function
@param {string} sObjectID path to the current supplier object
@private
*/
_updateViewModel : function() {
    //find out if there is a next object in the line:
    var oModel = this.getView().getModel();
    var oViewModel = this.getView().getModel("viewModel");
    var nextObjectId = parseInt(this.sObjectID) + 1;
    var prevObjectId = parseInt(this.sObjectID) - 1;
    // check if there is a next object by adding +1 to the supplier ID
    //we assume we get a field we can safely order from the server
    var bNext = !!oModel.getProperty("/Suppliers/" + nextObjectId);
    var bPrev = !!oModel.getProperty("/Suppliers/" + prevObjectId);
    oViewModel.setProperty("/buttonNext", bNext);
    oViewModel.setProperty("/buttonPrev", bPrev);
}
```

Listing 6.52 Binding the View to the Correct Context and Calculating Button Enablement

Here, we try to retrieve the next and previous suppliers by adding or subtracting one from the current ID, which assumes that the ID field is a field we can safely use to determine an order (which probably wouldn’t work most of the time). Usually, the service should be able to deliver information about the next or previous object in a collection.

Next, we need to add the line of code triggering this function (see Listing 6.53).
/**
* Binds the view to the object path and maintains the paging
* button state.
*
* @function
* @param {sap.ui.base.Event} oEvent pattern match event in
* route 'object'
* @private
*/

_onObjectMatched : function (oEvent) {
    this.sObjectID = oEvent.getParameter("arguments").ID;
    this.sObjectPath = "/Suppliers/" + this.sObjectID;
    this._bindView();
    this._updateViewModel();
},

Listing 6.53 Updating the Bound Supplier and the View Model from the _onObjectMatched Function

Note how we also updated the documentation for this method. When you now reload
the page, you can see that depending on whether we're on the first or the last sup-
plier, the correct button state is maintained. However, we still can't navigate with
these buttons because the corresponding event handlers for the press events are still
missing.

We'll add the onPagingButtonPress method to both the XML view as the event handler
for the buttons, and to the controller, where the actual navigation is implemented.
Because we're already using proper routing within the application, we can now sim-
ply make use of this routing by invoking the corresponding navTo function with the
information about the next or previous supplier.

In our scenario, as mentioned, we can just assume we can increment or decrement
the index of the supplier by one to get to the next or previous supplier. We therefore
simply need to write the two functions in Listing 6.54.

Listing 6.54 Navigation Functions for Paging through Suppliers via Arrow Buttons

We're using the path from the event object passed into the event handler, and we're
extracting the current supplier number out of that path. We convert it to an integer,
as adding +1 would cause a string concatenation. Then, we either subtract one, in case
a user clicks the arrow up, or we add one, when the user navigates down. We pass the
result to the navTo function as the ID parameter, so the navigation to the next or pre-
vious supplier will be triggered, and they will be displayed accordingly.

6.8.2 Using the Device Model

The device model is a normal JSON model, which we treat separately because it has a
special meaning within the application. This model isn't implemented for a particu-
lar view, however, but is more generally used throughout the application. It keeps
track of control properties that need to be adapted depending on whether the appli-
cation is displayed on a desktop or on a mobile (touchscreen) device.

This makes sense particularly for some properties that influence appearance, such as
the active states of a control you've tapped or clicked. Because this model can be used
in more than one view, we implement it separately and instantiate it within the com-
ponent.

The implementation for the model itself will go into a separate file, which we'll put
into the existing model directory of the application, and we'll call it models.js. The
code for this model is shown in Listing 6.55.

Listing 6.55 Device Model Definition
"use strict";
return {
    createDeviceModel: function () {
        var oModel = new JSONModel(Device);
        oModel.setDefaultBindingMode("OneWay");
        return oModel;
    }
};
Listing 6.55 Implementing the Device Model for the Application

The parameters passed into the `sap.ui.define` call are two classes that we'll need within this model to get all the information about our current client from the `sap.ui.Device` API and be able to store it within a JSON model. When this module is invoked, it will return a new function named `createDeviceModel`. Within this function, we're creating a new JSON model from the data retrieved from the device API, and because this model delivers its information in key-value pairs, we can pass the object to the JSON model constructor.

We then set `bindingMode` to `OneWay` because it doesn't make any sense to manipulate the values within this model from the application code. Finally, we return the newly created model. When we now go back to the application component, we just need to include the new model in the component.

We need to add the corresponding resource to the `define` section of the component as in Listing 6.56.

```
sap.ui.define([],
    function (UIComponent, ResourceModel, JSONModel, Device) {
        "use strict";

        Listing 6.56 Adding the Resources for the Device Model
```

Within the `init` function of the component, insert the code for the additional model instantiation as in Listing 6.57.

```
init : function () {
    var mConfig = this.getMetadata().getConfig();
    // call the base component's init function
    UIComponent.prototype.init.apply(this, arguments);
    // set the internationalization model
    this.setModel(new ResourceModel({
        bundleName : mConfig.i18nBundle
    }), "i18n");
    // create the device model here
    var oModel = new JSONModel(Device);
    oModel.setDefaultBindingMode("OneWay");
    this.setModel(oModel, "Device");
    // create the views based on the url/hash
    this.getRouter().initialize();
},
Listing 6.57 Device Model Instantiated in the init Function of the Application
```

Now, we can access the device model from our views. We can use the information about which device is accessing the application to determine if we want to show or hide controls, alter some of their behavior, or show information in a different control altogether. We could, for instance, choose to display the paging buttons we created for the suppliers only on phone screens.

In this case, all we had to do was change the lines in the detail view to instantiate the corresponding text control, as shown in Listing 6.58.

```
<Button icon="sap-icon://slim-arrow-up"
    press="onPageUp" enabled="{viewModel>/buttonPrev}"
    visible="{device>/system/phone}" />
```

Listing 6.58 Detail View for the Phone Device
enabled="{viewModel>/buttonNext}" 
visible="{device>/system/phone}"/>
</Toolbar>

Listing 6.58 Using Properties from the Device Model for Binding

You’ll see that the application now looks a bit different when in desktop mode, in which the arrow buttons in the detail view have disappeared. However, we can bring them back by switching into emulation mode in Google Chrome to see what the application now looks like on the phone.

6.9 Summary

In this chapter, you learned about client-side models and their usage in SAPUI5. You’ve seen how models are instantiated and have made use of the typical data-binding methods in the sample application.

Using this knowledge, we extended the application to a point at which the most common features in enterprise-grade applications have been touched upon. We’ve made the application multilingual by using a resource model and have made it more cross-platform compatible by implementing a device model for those features that need to be handled differently on a desktop than on typical mobile devices.

In the next chapter, we’ll enhance the application even more by adding the means to manipulate data in a model from the application—and this time, we’ll be operating on a real service instead of using local mock-up data.
## Preface

### Preface

Costello, John 2017

### Acknowledgments

Costello, John 2017

## PART I Introduction

### SAPUI5 at a Glance

1. What It Is and Where to Get It

2. History and Evolution
   - History
   - Evolution

3. Features
   - SAPUI5 Demo Kit
   - Model-View-Controller in SAPUI5
   - Cross-Browser Compatibility
   - Theming
   - Localization
   - Accessibility
   - Open Source in SAPUI5
   - Web Components

4. Use Cases

5. Product Comparison

6. SAPUI5 and OpenUI5

7. Summary

## Architecture

1. The Libraries
## Contents

### PART II Building Applications with SAPUI5

#### 3 Integrated Development Environment Setup

3.1 SAP Web IDE ................................................................. 63  
3.1.1 Creating an Account on SAP Cloud Platform .................. 64  
3.1.2 Accessing SAP Web IDE ........................................... 67  
3.1.3 SAP Web IDE Templates ........................................... 69  
3.2 WebStorm ............................................................... 79  
3.3 Visual Studio Code with UIS Command-Line Tools .................. 87  
3.4 Summary ................................................................. 89  

#### 4 Hello, SAPUI5 World

4.1 Coding Guidelines ........................................................... 91  
4.1.1 Global Variables ...................................................... 92  
4.1.2 Private Object Members ........................................... 93  
4.1.3 Code Formatting .................................................... 93  
4.1.4 Variable Naming Conventions .................................... 94  
4.2 Setup ........................................................................ 95  
4.2.1 Setting Up Your HTML Start Page ......................... 96  

### 5 Building Model-View-Controller Applications

5.1 Models, Views, and Controllers ........................................ 119  
5.2 Structure .................................................................. 120  
5.2.1 Application Overview ............................................. 122  
5.2.2 Building the First Page ............................................ 123  
5.2.3 Table Coding .......................................................... 128  
5.3 Building a Simple View ............................................... 130  
5.3.1 Namespaces and Resource Path ................................. 131  
5.3.2 Creating the Master JavaScript View ......................... 133  
5.3.3 Creating the Master Controller .................................. 135  
5.3.4 Creating a Detail View and Controller ...................... 139  
5.4 View Types ................................................................ 144  
5.4.1 XML Views ........................................................... 151  
5.4.2 Transforming JavaScript Views into XML Views .......... 153  
5.5 Components ................................................................ 163  
5.5.1 Creating the Component File ................................... 163  
5.5.2 Adding a Shell around the Component ...................... 167  
5.5.3 Enhancing the Look of a Table ................................. 169  
5.5.4 Component Metadata ............................................. 170
| 6.1 Using Models: A JSON Sample .......................... | 191 |
| 6.1.1 Instantiation and Loading of Data ............... | 192 |
| 6.1.2 Accessing Model Values ............................ | 194 |
| 6.2 Property Binding ........................................... | 200 |
| 6.2.1 Methods for Binding a Control's Property ......... | 201 |
| 6.2.2 Using Data Types ......................................... | 205 |
| 6.2.3 Defining a Custom Data Type ........................ | 211 |
| 6.3 Using Formatters ............................................. | 214 |
| 6.4 Aggregation Binding ......................................... | 224 |
| 6.4.1 bindAggregation ......................................... | 230 |
| 6.4.2 Using a Factory .......................................... | 230 |
| 6.5 Element Binding ............................................. | 237 |
| 6.6 Expression Binding and Calculated Fields ............. | 239 |
| 6.6.1 Calculated Fields ....................................... | 239 |
| 6.6.2 Expression Binding ..................................... | 241 |
| 6.7 Resource Models and Internationalization ............ | 243 |
| 6.7.1 File Location ............................................. | 243 |
| 6.7.2 File Naming Convention .............................. | 244 |
| 6.7.3 Code Page .................................................. | 245 |
| 6.7.4 Using a Resource Model ............................... | 245 |
| 7 CRUD Operations .................................................. | 259 |
| 7.1 REST and CRUD ............................................... | 260 |
| 7.2 Connecting to REST Services ............................ | 260 |
| 7.2.1 Configuring the Mock Service .................... | 262 |
| 7.2.2 Extending the JSON Model ......................... | 265 |
| 7.3 Using CRUD Operations ...................................... | 267 |
| 7.3.1 Editing an Existing Entry ............................. | 267 |
| 7.3.2 Creating a New Entry .................................. | 278 |
| 7.3.3 Deleting an Entry ........................................ | 289 |
| 7.4 Sorting, Filtering, and Grouping in JSON Models .... | 291 |
| 7.4.1 Sorting ...................................................... | 292 |
| 7.4.2 Filtering ..................................................... | 297 |
| 7.4.3 Grouping ..................................................... | 304 |
| 7.5 Summary .......................................................... | 307 |
| 8 Using OData .......................................................... | 309 |
| 8.1 OData at a Glance ............................................. | 309 |
| 8.1.1 Northwind OData Service ........................... | 310 |
| 8.1.2 Service Document ...................................... | 311 |
| 8.1.3 Service Metadata Document ....................... | 312 |
| 8.1.4 Accessing Data ......................................... | 314 |
| 8.2 OData Model at a Glance .................................... | 320 |
| 8.2.1 Service Metadata ....................................... | 323 |
| 8.2.2 Instantiating the OData Model in SAP Web IDE .... | 324 |
## 8.3 Reading Data
8.3.1 Reading Data Manually .......................................................... 328
8.3.2 Accessing Data via Data Binding ........................................ 333
8.3.3 Best Practices ......................................................................... 336
8.3.4 Displaying Additional Product Information .......................... 342
8.3.5 Displaying Navigation Properties .......................................... 343

## 8.4 Filter, Sort, Expand, and Group
8.4.1 Filtering with $filter .............................................................. 347
8.4.2 Sorting with $orderby .......................................................... 352
8.4.3 Expanding with $expand .................................................... 356
8.4.4 Grouping with group .......................................................... 360

## 8.5 Paging and Thresholds .......................................................... 361

## 8.6 Batch Mode .......................................................................... 366

## 8.7 One-Way and Two-Way Bindings
8.7.1 One-Way Binding ............................................................... 368
8.7.2 Two-Way Binding ............................................................... 371

## 8.8 Writing Data
8.8.1 Creating an Entry ................................................................. 378
8.8.2 Updating an Entry ............................................................... 383
8.8.3 Deleting an Entry ............................................................... 386

## 8.9 Function Imports .................................................................. 388

## 8.10 Concurrency Control .......................................................... 392

## 8.11 Summary ............................................................................ 395

## 9 Application Patterns and Examples ........................................ 397

### 9.1 Layouts
9.1.1 Full-Screen Layout: sap.m.App .......................................... 403
9.1.2 Split Screen Layout: sap.m.SplitApp .................................... 405
9.1.3 Flexible Column Layout: sap.f.FlexibleColumnLayout .......... 410

### 9.2 Floor Plans
9.2.1 Worklist .............................................................................. 414

## 10 Advanced Concepts ................................................................ 481

### 10.1 Writing Your Own Controls
10.1.1 SAPUI5 Control Structure ................................................. 482
10.1.2 Implementing a Composite Control ................................... 488

### 10.2 Using Fragments
10.2.1 Creating Fragments .......................................................... 498
10.2.2 Embedding Fragments into Views ...................................... 502
10.2.3 Using Dialogs in Fragments ............................................... 507

### 10.3 SAP OData Annotations
10.3.1 Custom SAP OData 2.0 Annotations ................................... 511
10.3.2 OData 4.0 Annotations ..................................................... 513

### 10.4 Smart Controls
10.4.1 Smart Tables and Smart Filters Bar .................................... 515
10.4.2 Smart Form and Smart Fields with Value Help ................... 523
PART III  Finishing Touches

11  Adding Enterprise-Grade Features 543

11.1 Theming ...................................................................................................................... 543
  11.1.1 Manual Restyling ..................................................................................................... 544
  11.1.2 UI Theme Designer .................................................................................................. 548

11.2 Security .......................................................................................................................... 554
  11.2.1 Input Validation ........................................................................................................ 554
  11.2.2 Cross-Site Scripting .................................................................................................. 554
  11.2.3 URLs and Whitelist Filtering .................................................................................... 555
  11.2.4 frameOptions and Central Whitelisting ................................................................. 556

11.3 Performance ................................................................................................................... 557
  11.3.1 Bundling and Component Preload ............................................................................. 557
  11.3.2 Minification and Uglification .................................................................................... 558

11.4 Accessibility ................................................................................................................... 575
  11.4.1 Importance of Inclusion and Accessibility ............................................................. 575
  11.4.2 SAPUI5 Accessibility Features ............................................................................... 579
  11.4.3 Making Your Applications Accessible ....................................................................... 581

11.5 Summary .......................................................................................................................... 583

12  Debugging and Testing Code 585

12.1 Debugging ...................................................................................................................... 586
  12.1.1 Tricks to Know ........................................................................................................... 586
  12.1.2 Debugging Support Tools ........................................................................................ 588

12.2 Writing Unit Tests ......................................................................................................... 597
  12.2.1 Setting Up a QUnit Test Page ................................................................................... 599
  12.2.2 Writing a Unit Test for a Custom Control ............................................................. 602
  12.2.3 Unit Tests for Applications ...................................................................................... 607

12.3 One-Page Acceptance Tests ........................................................................................... 614
  12.3.1 Architecture ............................................................................................................. 615
  12.3.2 OPA Test Structure .................................................................................................. 616
  12.3.3 Writing waitFor Functions ....................................................................................... 616
  12.3.4 Writing a One-Page Acceptance Test ..................................................................... 622
  12.3.5 Application Lifecycle Handling .............................................................................. 627
  12.3.6 Structuring Tests with Page Objects ....................................................................... 628
  12.3.7 Full Application Test Setup ...................................................................................... 629

12.4 Mocking Data: Using the Mock Server ......................................................................... 634
  12.4.1 Basic Instantiation and Configuration ................................................................. 635
  12.4.2 Advanced Concepts and Configuration ............................................................... 636

12.5 Linting Code .................................................................................................................. 640

12.6 Summary .......................................................................................................................... 642

13  Accessing and Connecting to the Backend 643

13.1 Same-Origin Policy ........................................................................................................ 643

13.2 Disable Web Security in Google Chrome ....................................................................... 648

13.3 SAP Cloud Platform Destinations .................................................................................. 650
  13.3.1 Simple Backend Destination ................................................................................. 651
  13.3.2 Northwind Destination ......................................................................................... 655
  13.3.3 ESS Demo Gateway Destination ............................................................................. 657

13.4 Summary .......................................................................................................................... 658
## Index

<table>
<thead>
<tr>
<th>_onObjectMatched</th>
<th>_onRouteMatched</th>
<th>@sapUIDarkestBorder</th>
<th>/UI5/UI5_REPOSITORY_LOAD</th>
<th>Aggregation</th>
<th>cardinality</th>
<th>227, 228, 485</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>default</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>hidden</td>
<td></td>
<td>492</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>table</td>
<td></td>
<td>234</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Aggregation binding</td>
<td>157, 224, 225, 243</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>factory</td>
<td></td>
<td>230</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>alt</td>
<td></td>
<td>582</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Analysis Path Framework (APF)</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Analytical applications</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AnalyticalTable</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Angular</td>
<td></td>
<td>79</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OData 2.0</td>
<td>511</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OData 4.0</td>
<td>513, 514</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>smart control</td>
<td>515</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>smart template</td>
<td>524</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>annotationsLoaded</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Anonymous function</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>API</td>
<td>483</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>class</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>control</td>
<td>109</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>deprecated and experimental</td>
<td>702</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>documentation</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>experimental</td>
<td>702</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OData model documentation</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SAP Fiori launchpad</td>
<td>461</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sap.ui.Device</td>
<td>401</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>whitelist filtering</td>
<td>555</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Application</td>
<td>230</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>accessibility</td>
<td>581</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>architecture</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>break</td>
<td>701</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>directory structure</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>features</td>
<td>445</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lifecycle handling</td>
<td>627</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Manage Products application</td>
<td>472</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>migrate settings</td>
<td>186</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pattern</td>
<td>397</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>startup</td>
<td>703</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>styling</td>
<td>695</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>template</td>
<td>677</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ABAP backend</th>
<th>676</th>
</tr>
</thead>
<tbody>
<tr>
<td>deployment</td>
<td>681</td>
</tr>
<tr>
<td>ABAP Workbench</td>
<td>683</td>
</tr>
<tr>
<td>Absolute binding path</td>
<td>335, 336</td>
</tr>
<tr>
<td>Accessibility</td>
<td>36, 575</td>
</tr>
<tr>
<td>alternative texts and tooltips</td>
<td>582</td>
</tr>
<tr>
<td>benefits</td>
<td>577</td>
</tr>
<tr>
<td>colors</td>
<td>583</td>
</tr>
<tr>
<td>correct label</td>
<td>582</td>
</tr>
<tr>
<td>features</td>
<td>575, 579</td>
</tr>
<tr>
<td>keyboard handling</td>
<td>580</td>
</tr>
<tr>
<td>legal regulations</td>
<td>578</td>
</tr>
<tr>
<td>role</td>
<td>581</td>
</tr>
<tr>
<td>sizing</td>
<td>583</td>
</tr>
<tr>
<td>titles</td>
<td>583</td>
</tr>
<tr>
<td>Action</td>
<td>458, 620</td>
</tr>
<tr>
<td>addAggregationName</td>
<td>107</td>
</tr>
<tr>
<td>addAggregationName</td>
<td>107</td>
</tr>
<tr>
<td>Addressable</td>
<td>105</td>
</tr>
<tr>
<td>Addressable</td>
<td>105</td>
</tr>
<tr>
<td>addStyleClass</td>
<td>695</td>
</tr>
<tr>
<td>Advanced concepts</td>
<td>481</td>
</tr>
<tr>
<td>Aggregation</td>
<td>105, 110, 484, 633</td>
</tr>
<tr>
<td>add to control metadata</td>
<td>484</td>
</tr>
<tr>
<td>adding children</td>
<td>105</td>
</tr>
</tbody>
</table>
Node.js ........................................... ................................... 560
advantages .................................................. 560
setup .......................................................... 561
Northwind .................................................. 514
destination .................................................. 655
Not found handling ................................... 445, 446

O

Object header .................................. 239, 240
Object view ........................................... 436
ObjectNotFound ....................................... 447
OData .................................................. 56, 191, 193, 309
2.0 .......................................................... 511
2.0 annotation .......................................... 511
4.0 .......................................................... 511
4.0 annotation .......................................... 511
accessing data ......................................... 314, 432
annotation .................................................. 513
best practices App view.xml ...................... 340
best practices Component.js ..................... 338
best practices folder structure .................... 337
best practices index.html ......................... 341
best practices manifest.json ...................... 339
best practices master.view.xml ................... 341
class inheritance ......................................... 56
concurrency control .................................. 392
create model ........................................... 383
destination .................................................. 383
display .................................................. 311
expanding .................................................. 356
expression binding .................................. 342
filtering .................................................. 347
function import ........................................ 388
grouping .................................................. 360
model ..................................................... 319, 320
Northwind .............................................. 310, 312
one-way and two-way binding ..................... 368
overview .................................................. 309
reading data ............................................ 627, 330
SAP Web IDE ........................................ 324, 330
Shop application ....................................... 474
sorting .................................................. 352
update model ........................................... 386
write-enabled ........................................... 388
write-support .......................................... 376

OData .................................................. 374
OData model ........................................... 708
listening to events .................................... 710
ODBC .................................................. 310
onAfterRendering ..................................... 49, 55
onBeforeRendering ................................... 50, 55
onDisplay .................................................. 282
One-way binding ..................................... 191, 368, 369
controller .................................................. 371
onExit ...................................................... 49
onInit ..................................................... 49, 266, 302, 504, 706
onPress .................................................. 295
onPressImage .......................................... 103
onSave ...................................................... 288
OPAS .................................................. 586, 614, 623
action ..................................................... 620
architecture ............................................. 615
extend ..................................................... 626
folder structure ......................................... 630
matcher ..................................................... 618
mock server ............................................. 636
page object ............................................. 629
test ......................................................... 622
test structure ............................................ 616
waitFor ..................................................... 618
Open source ............................................. 36
OpenAjax .................................................. 92
OpenUI .................................................. 25, 27, 42, 43, 515
CDN ..................................................... 690
openui5_preload ....................................... 560, 566
SDK ....................................................... 80
Optimistic concurrency ................................ 392
Scope .................................................... 392
Optimized network request ....................... 570
Origin .................................................... 643
oTemplate .............................................. 129
Overflow .................................................. 458
Oversizing ............................................... 696, 698

P

Package.json ........................................... 567
Padding .................................................... 547
Page control ............................................ 268
Page object .............................................. 628, 631, 633
Pageable ............................................... 512
Paging .................................................. 251, 361, 363
parseError .............................................. 209
parseValue .............................................. 207
patternMatched ....................................... 439
Payload .................................................. 380, 384
GET ....................................................... 381
POST ...................................................... 381
Performance ........................................... 557, 583
worst practices ......................................... 700
Pessimistic concurrency ......................... 392
Phoenix ................................................... 26
Platform-as-a-Service (PaaS) .................... 64, 659
PopupMode .............................................. 410
Postman .................................................. 374, 375
create entry ............................................. 380
Press event .............................................. 103
Private method ......................................... 701
Private object member ............................ 93
Product information ................................ 342
Project .................................................... 468
Property binding ..................................... 203
bindProperty ............................................ 203
data type .................................................. 205
format options ......................................... 210
JSON model ............................................ 204
Protected method ...................................... 701
Purchase Order app ................................... 39

Q

Quality .................................................. 640
QUnit .................................................. 585, 586, 597–599, 614, 616, 622, 625, 642
constructor outcome ................................ 603
DOM structure ........................................... 627
test ......................................................... 600
test files .................................................. 601
test page .................................................. 599
test skeleton ............................................ 602
R

RadioButtonGroup ................................... 105, 107
React ...................................................... 42, 79
Relative binding path ................................ 335
Render manager ....................................... 46, 488, 580
Renderer .................................................. 495
Representational State Transfer (REST) ....... 259
requestFailed .......................................... 277
requestSend ............................................. 57
Resource .................................................. 136
folder ..................................................... 132
path ....................................................... 131
Resource bundle ...................................... 243, 278
code page .............................................. 245
file location ............................................. 243
test page ................................................. 244
Resource model ....................................... 243
instance .................................................. 246
usage ..................................................... 245
Responsive behavior ................................ 468
REST ..................................................... 259
connect to services .................................. 260
service .................................................... 259
stateless .................................................. 259
RGB color ............................................... 547
Rich Internet Applications (RIA) ............... 581
RichTextEditor .......................................... 47
rootView .................................................. 188
Routing .................................................. 30, 174, 176, 188, 420, 442
collection .................................................. 175
detail controller ........................................ 181, 182
empty pattern .......................................... 442
handling .................................................. 442
initialization ............................................. 177
master controller ........................................ 179
pattern ..................................................... 180
Routing configuration ................................ 269, 400, 408, 409
Same origin policy ..................................... 643
Same-origin policy ..................................... 322, 324
Paul J. Modderman is a senior product architect and technology evangelist for Mindset Consulting. He has more than 10 years of software development expertise, specializing in ABAP, ABAP Objects, .NET, Python, and Java. At Mindset, Paul ideates and develops innovative software products based on SAP, often integrating SAP with cutting-edge cloud software such as Google for Work.

Christiane Goebels has been in web development ever since starting her career at SAP in 2000. Over the years, she has worked on SAP-related and non-SAP-related projects big and small for customers all over the world. She led her own internet agency from 2005 to 2010, and re-joined SAP in 2012 as part of the central SAPUI5 development team.

Denise Nepraunig is a software developer at SAP in Walldorf, where she creates SAPUI5 applications and was involved in the development of the SAP Web IDE. Before she joined SAP, she worked at an SAP partner company, where she worked with SAPUI5 and OData development with ABAP, gaining hands-on experience with SAP Fiori and the SAP Mobile Platform.

Thilo Seidel is a product owner of SAP Fiori launchpad on the weekdays and an occasional hacker on weekends. He built his first web page back in 2002 and instantly fell in love with the browser. He has taken on various roles since then, including sales, designer thinker, traveler, student, and project manager. Before joining SAPUI5, his technological journey involved freelancing with jQuery, Bootstrap, and Ruby on Rails.