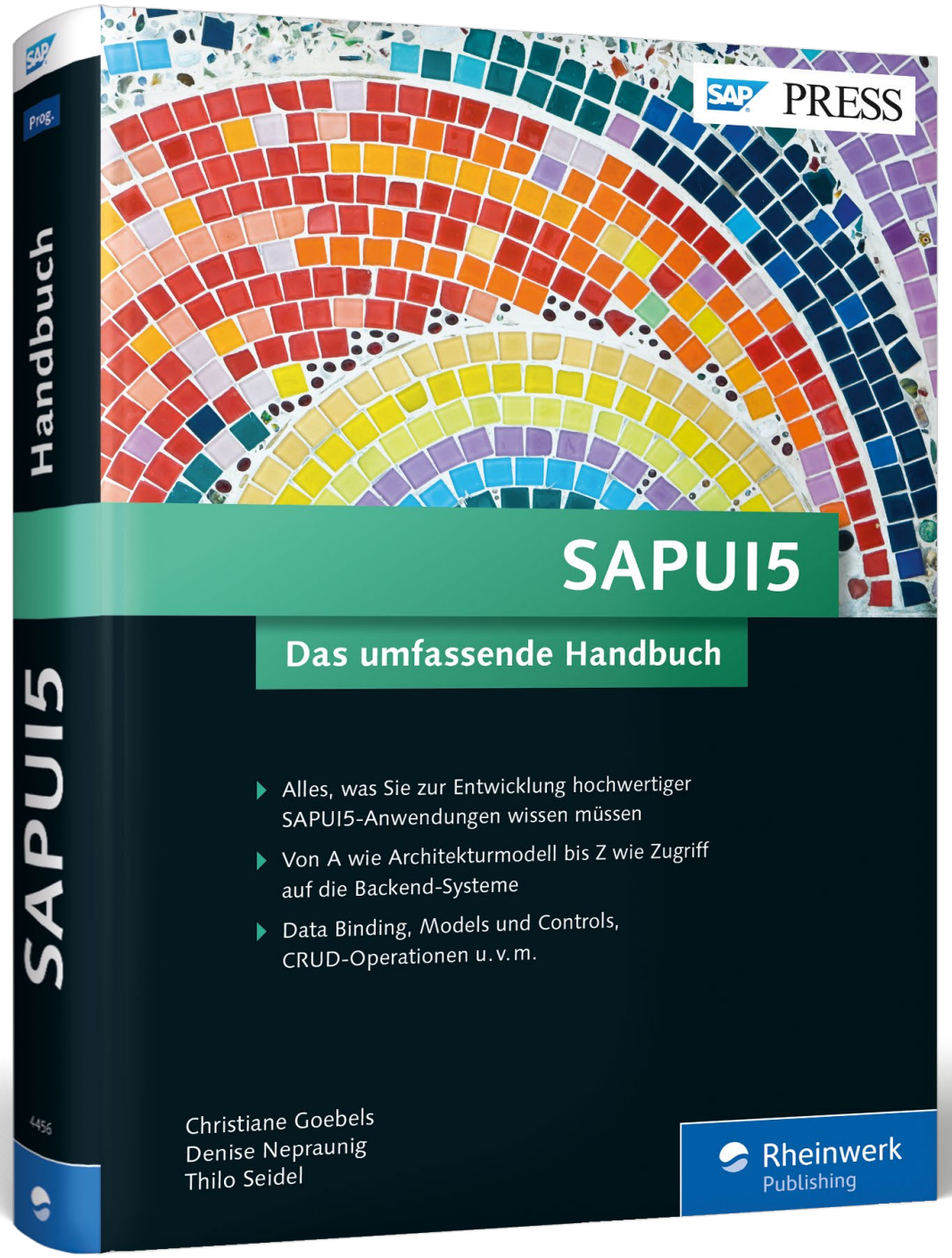


Leseprobe

Mit diesem Buch tauchen Sie ein in die Anwendungsentwicklung mit SAPUI5 und lernen alles, was Sie über Design, Programmierung und Betrieb der Apps wissen müssen. In dieser Leseprobe erläutern die Autoren die allgemeine Nutzung von SAPUI5 sowie deren Modelltypen und erörtern die unterschiedlichen Formen des Data Bindings.

Leseprobe weiterempfehlen

Christiane Goebels, Denise Nepraunig, Thilo Seidel
SAPUI5 – Das umfassende Handbuch
699 Seiten, gebunden, November 2016
79,90 Euro, ISBN 978-3-8362-4456-5

www.sap-press.de/4303

In diesem Kapitel erörtern wir die allgemeine Nutzung von SAPUI5 und die darin enthaltenen Modelltypen sowie das Data Binding.

5 Modelle und Bindings

Im vorherigen Kapitel 4, »Aufbau von MVC-Anwendungen«, haben wir erläutert, wie Modelle als Datencontainer funktionieren, in denen sämtliche Geschäftsdaten enthalten sind, die in Ihrer Anwendung verarbeitet werden. SAPUI5 wird mit verschiedenen vordefinierten, einsatzbereiten Modellklassen bereitgestellt. In der Beispielanwendung haben Sie schon ein JSON- und ein Ressourcenmodell in Aktion gesehen. In diesem Kapitel erklären wir diese Modelltypen genauer, und Sie erfahren mehr über die Bindungsfunktionalität, die in allen Anwendungen zum Einsatz kommt.

Data Binding beinhaltet das Verbinden von Daten, die in einem bestimmten Modell in einer Anwendung verfügbar sind, mit einem bestimmten Teil der Benutzeroberfläche auf dem Bildschirm. Diese Verbindung kann entweder *unidirektional* oder *bidirektional* sein. Bei der ersten Verbindung werden die Daten aus dem Modell bereitgestellt, und die Modelldaten bleiben unberührt, wenn sie in der Benutzeroberfläche geändert werden. Bei der letzten Verbindung werden die Daten aus der Benutzeroberfläche und dem Modell laufend synchronisiert, wie es bereits im Beispiel des letzten Kapitels bei den Eingabefeldern gezeigt wurde.

Bei den meisten Modellen können Sie selbst entscheiden, ob Sie eine uni- oder eine bidirektionale Bindung verwenden möchten. Eine unidirektionale Bindung bedeutet allerdings nicht, dass die Daten in solch einem Modell gar nicht aktualisiert werden können: Sie müssen Sonderaktionen ausführen, um das Modell zu aktualisieren. Warum das so ist, wird deutlich, wenn wir einen genaueren Blick auf die Menge an Daten werfen, auf die unsere Anwendung zugreifen kann, und prüfen, woher diese Daten stammen. Zuerst betrachten wir ein JSON-Beispiel.

5.1 Verwenden von Modellen – ein JSON-Beispiel

JSON-Modelle sind clientseitige Modelle. Dies bedeutet, dass die Anwendung nach der Instanziierung des Modellobjekts und dem Laden der Daten Zugriff auf alle im Modell enthaltenen Daten hat.

Andere Modelle, wie z. B. das OData-Modell, sind serverseitige Modelle. Bei der Instanziierung eines bestimmten Modellobjekts werden Daten nicht automatisch geladen. Die Daten müssen angefordert werden und werden dann bei Bedarf geladen. Dies kann entweder über bewusste Aktionen im Anwendungscode erfolgen (z. B. Aufruf der entsprechenden Funktionen im Modell) oder automatisch geschehen, wenn die Daten aus solch einem Modell mit bestimmten Bindungstypen an ein bestimmtes Bildschirmelement gebunden sind. Das Modellobjekt selbst übernimmt dann die Anforderung der Daten aus dem Service. Weitere Details hierzu erhalten Sie im nächsten Kapitel, wenn wir genauer auf das oData-Modell eingehen.

Wie im vorangehenden Kapitel verwenden wir nun ein JSON-Modell und prüfen, wie dieses in der Anwendung eingesetzt werden kann und wie die unterschiedlichen Bindungstypen aus SAPUI5 angewendet werden können.

5.1.1 Instanziierung und Laden der Daten

Wir rekapitulieren zuerst die Modellstruktur aus der Beispielanwendung, und greifen wir nochmals auf, was wir im vorangehenden Kapitel erreicht haben. Alle Geschäftsdaten, die wir für unsere Anwendung benötigen, liegen uns in einer JSON-Struktur vor. Wie Sie gesehen haben, ist das Instanzieren eines JSON-Modells mit der Konstruktorfunktion der entsprechenden Modellklasse ziemlich leicht:

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

Sie wissen bereits, was hier geschieht: SAPUI5 erstellt ein neues Objekt mit dem Prototyp `sap.ui.model.json.JSONModel`.

Als Nächstes benötigen wir einige Daten für dieses Modell, die den Controls in unserer Anwendung zur Verfügung stehen sollen. Wie zuvor verwenden wir nicht einfach ein JSON-Objekt, sondern laden die Daten aus einer separaten Datei, um die Antwort zu simulieren, die wir von einem Service erhalten würden.

Im Hintergrund werden zwei Optionen für das JSON-Modell unterschieden: ob es 1) eine URL oder 2) ein Objekt als Parameter für den Konstruktor empfangen hat. Dieses Mal wurde eine URL übergeben, die von der Konstruktorfunktion des Modells erkannt wird. Die Funktion versucht dann automatisch, die Datei mit der Funktion `setData` des Modells aus der URL zu laden. Dies können Sie erkennen, wenn Sie einen Blick auf die Konstruktorfunktion werfen (siehe Listing 5.1).

```
var JSONModel = ClientModel.extend("sap.ui.model.json.JSONModel", {
  constructor : function(oData) {
    ClientModel.apply(this, arguments);
    if (oData && typeof oData == "object") {
      this.setData(oData);
    }
  },

  metadata : {
    publicMethods : ["setJSON", "getJSON"]
  }
});
```

Listing 5.1 sap.ui.model.JSONModel-Konstruktor

Wurden die Daten erfolgreich abgerufen, werden sie im Modellobjekt, genauer gesagt in einer Eigenschaft des Modells mit dem Namen `oData`, gespeichert. Beachten Sie, dass diese Eigenschaft nicht für den direkten Zugriff gedacht ist und hier nur erwähnt wird, um zu verdeutlichen, wo die Daten abgelegt werden.

Neben diesem automatischen Laden bei der Instanziierung gibt es eine weitere Möglichkeit, um die Anforderung von Daten auszulösen. Das JSON-Modell verfügt außerdem über eine Methode `loadData`, mit der wir das Laden von Daten, wie es in Listing 5.2 dargestellt ist, manuell auslösen können.

```
// 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 5.2 Instanziieren des Modells und manuelles Laden der Daten

Diese Komfortfunktion ist tatsächlich ein Wrapper für eine XHR-Anfrage, die wir ansonsten selbst erstellen müssten, wenn wir selbst entscheiden möchten, wann die Anforderung zum Laden von Daten ausgelöst werden soll. Wir

erhalten so eine Möglichkeit, um einen Teil des aus der übergeordneten Klasse `sap.ui.model.Model` vererbten Standardverhaltens zu beeinflussen.

Wir lassen das Standardverhalten für den Moment aber unverändert und sehen uns genauer an, was geschieht, wenn die Funktion `loadData` aufgerufen wird (siehe Listing 5.3; Codezeilen, die derzeit nicht relevant sind, haben wir ausgelassen).

```
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) {
[...]
that.setData(oData, bMerge);
 that.fireRequestCompleted({url : sURL, type : sType,
 async : bAsync,
 headers: mHeaders, info : "cache=" + bCache + ";bMerge=
" + bMerge,
 infoObject: {cache : bCache, merge : bMerge}, success: true});
  },
  error: function(XMLHttpRequest, textStatus, errorThrown){
[...]
  });
 };
```

Listing 5.3 Auszug aus der Funktion loadData des JSON-Modells

Neben der eigentlichen Anforderung gibt es auch Code, der bestimmte Ereignisse auslöst, die die Listener über den Status der Anforderung informieren. Mit diesen Ereignissen können Sie in Ihrer Anwendung ermitteln, wann Sie mit den Daten aus dem Modell sicher arbeiten können oder wann die Anforderung fehlgeschlagen ist. Möglicherweise möchten Sie auf einen solchen Fehler reagieren. Diese Ereignisse werden im Modell ausgelöst; daher können Sie problemlos Event-Handler mit ihnen verknüpfen.

Jedes Mal, wenn eine solche Anforderung gesendet wird, wird das Ereignis `requestSent` im Modell ausgelöst. Wichtiger für unsere nachfolgenden Experimente im Modell ist jedoch das Ereignis, das bei Erfolg (oder Fehler) ausgelöst wird: das Ereignis `requestCompleted`.

5.1.2 Zugriff auf Modellwerte

Sie erfahren nun, wie Sie über die Methoden `getProperty` und `setProperty` auf Werte in einem JSON-Modell zugreifen und diese manipulieren können. Um allerdings sicherzustellen, wann die Daten im Modell verfügbar sein sollen und wann wir damit arbeiten können, müssen wir das Ereignis `requestCompleted` anhängen.

Wechseln wir für die restliche Einführung in das Data Binding zu einem kleineren und isolierteren Anwendungsbeispiel. Dieses nennen wir *Databinding First Steps*. Mit diesem Beispiel werden unsere ersten zaghaften Schritte im Modell transparenter. Wir kehren gelegentlich zum größeren Anwendungsbeispiel zurück, um das neu Erlernte umzusetzen.

Die Datei *index.html* für diese neue, kleine Beispielanwendung sehen Sie in Listing 5.4.

```
<!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 id="content"></div>
 </body>
</html>
```

Listing 5.4 Erste JSON-Beispielanwendung

In dieser Beispielanwendung sind auch JSON-Daten in einer Datei mit dem Namen *data.json* enthalten, die sich im Ordner *service* befinden (wie in den vorangehenden Beispielen). Die Datei sieht derzeit wie in Listing 5.5 aus.

```
{
    "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 5.5 JSON-Modell

Wenn Sie diese Anwendung ausführen, wird ein nicht übermäßig befüllter Bildschirm angezeigt (siehe Abbildung 5.1).

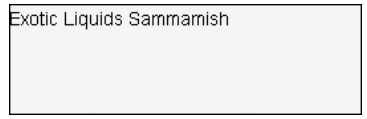


Abbildung 5.1 Erste Anzeige der Beispielanwendung

In dieser Anwendung rufen wir zunächst die Daten aus unserer Modelldatei manuell ab, indem wir die Funktion `loadData` aufrufen. Danach möchten wir auf einen der Werte im Modell zugreifen und diesen bearbeiten. Wir können die Methoden `getProperty` und `setProperty` hierzu verwenden, die für JSON-Modelle selbst implementiert sind. Mit diesen Methoden können Sie auf bestimmten Knoten im Modell operieren und auf die Werte mit der Pfadsyntax, die Sie kurz in Kapitel 4, »Aufbau von MVC-Anwendungen«, gesehen haben, zugreifen. Sie erfahren, wie Sie den bald benötigten Pfad ermitteln. Zuerst möchten wir aber auf ein Timing-Problem eingehen, das hier vorliegt. Dieses Problem wird am offensichtlichsten, wenn Sie die Anwendung in einem Browser ausführen und das Modellobjekt zur Laufzeit prüfen.

Wenn Sie den Effekt selbst sehen möchten, setzen Sie einen Breakpoint direkt nach der Zeile des Codes, in der das Modell instanziiert ist, und betrachten das Modellobjekt im Debugger Ihres Browsers. Sie können erkennen, dass es im Modell noch keine bearbeitbaren Modelle gibt (siehe Abbildung 5.2).

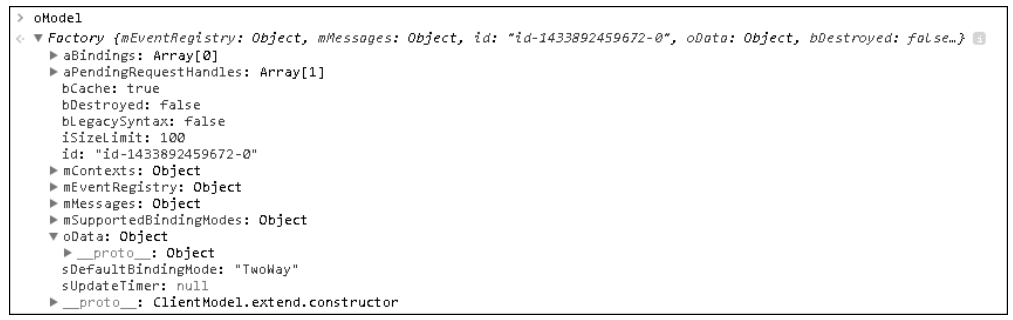


Abbildung 5.2 oModel in der Entwicklertools-Konsole von Google Chrome, bevor die Daten geladen wurden

Wenn wir den nächsten Breakpoint in der Funktion setzen, rufen wir `requestCompleted` auf. War die Anforderung erfolgreich, wird Ihnen sofort ein Ergebnis wie in Abbildung 5.3 angezeigt, wenn Sie diesen Breakpoint erreichen.

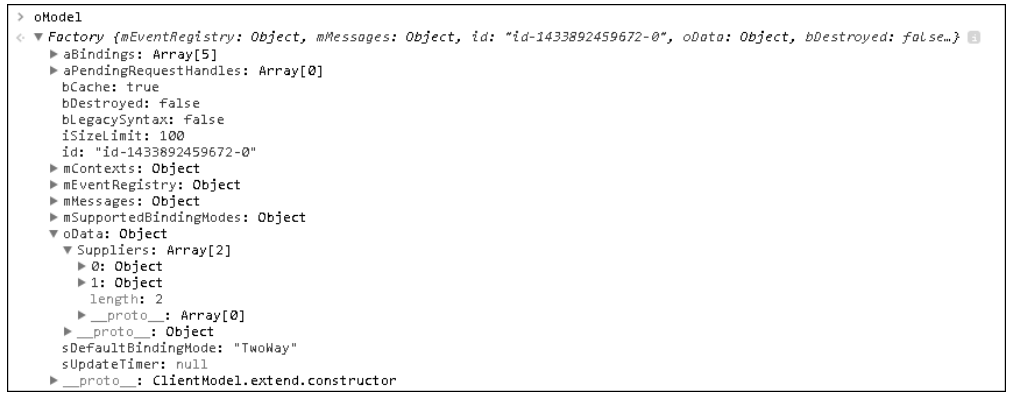


Abbildung 5.3 oModel in der Entwicklertools-Konsole von Google Chrome, nachdem die Daten geladen wurden

Sie können sehen, dass die Eigenschaft `oData` des Modells mit den entsprechenden Daten aus der Datei *data.json* befüllt wurde.

Bearbeiten wir nun die Daten mit den zuvor genannten Methoden. Die Callback-Funktion wird nur aufgerufen, wenn das Ereignis `requestCompleted` ausgelöst wurde:

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

Befüllen wir nun diese Callback-Funktion. Um Zugriff auf die Werte im Modell zu erhalten, benötigen Sie nur eine Information: den Pfad zum Knoten, mit dem Sie arbeiten möchten.

Die meisten Modelle in SAPUI5 besitzen eine hierarchische Struktur, über die Sie auf die Knoten und Werte zugreifen können, wenn Sie den jeweiligen Pfad angeben. Mit der entsprechenden Syntax werden einfach die Modellknotenebenen getrennt, was zu dem gewünschten Wert mit Schrägstrichen führt:

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

Über diesen Pfad erfolgt der Zugriff auf den Wert der Eigenschaft `Name`, die dem ersten Lieferanten im Modell der Beispielanwendung zugewiesen ist. Der erste Teil des Pfads weist auf die Lieferantensammlung im Modell, im nächsten Teil können wir die Reihenfolge der Lieferanten im Modell auswählen, und der letzte Teil verweist auf die bestimmte Eigenschaft, mit der

wir unsere Control-Eigenschaft binden möchten. Daher greift dieser Pfad auf die hervorgehobenen Werte im nachstehenden Modellcode zu:

```
{
  "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 5.6 Pfad zu einem bestimmten Knoten im Modell

Mit dem führenden Schrägstrich wird angegeben, dass dieser Pfad am Modellstamm beginnt. Ist ein führender Schrägstrich vorhanden, bedeutet das, dass ein absoluter Pfad vorliegt. Wir können auch relative Pfade verwenden, die praktischer sind, wenn wir in Abschnitt 5.2, »Property Binding«, die Bindungskontexte besprechen.

Nachdem wir nun den Pfad kennen, können wir ihn als Parameter in den Methoden `getProperty` und `setProperty` in der Anwendung innerhalb der Callback-Funktion verwenden. Wir geben den Pfad zum Wert, auf den wir als Parameter zugreifen möchten, an die Methode `getProperty` im Modell weiter.

Anschließend ändern wir den Wert nach Bedarf und geben diesen geänderten Wert über die Methode `setProperty` wieder an das Modell zurück. Diese Methode verwendet den Pfad als Parameter sowie den Wert, den Sie zuweisen möchten. Schließlich sieht der Code wie in Listing 5.7 aus.

```
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);
});
```

Listing 5.7 Finale Callback-Funktion für das Ereignis requestCompleted

Wenn Sie den Code speichern und diese Seite anzeigen lassen, sehen Sie, dass der Wert für den ersten Lieferanten tatsächlich im Modell und in der Benutzeroberfläche geändert wurde. Letzteres geschieht, da das Modell selbst erkennt, dass die Daten geändert wurden, und die Benutzeroberfläche darüber benachrichtigt, dass ein Wert aktualisiert wurde.

getProperty

Die Methode `getProperty` gibt nicht zwangsläufig einen einfachen Wert zurück, sondern je nach Art der im entsprechenden Pfad gefundenen Daten ein Objekt oder mehrere Objekte. Wenn die Funktion `getProperty` auf den Pfad `/Suppliers/0` verweist, wird ein JavaScript-Objekt zurückgegeben, das mit Eigenschaften aus dem JSON-Modell befüllt ist.

Betrachten wir als Nächstes, wie diese bestimmte Verbindung hergestellt wird, da sie die Controls innerhalb unserer Anwendung bereits verwenden.

5.2 Property Binding

Da das Modell nun vorliegt, stehen uns die benötigten Informationen über dessen Struktur und Inhalte sowie die Werte aus dem Modell zur Verfügung, die wir in der Anwendung anzeigen möchten. Wie in Kapitel 4, »Aufbau von MVC-Anwendungen«, erläutert, ist *Property Binding* eine Möglichkeit, um Werte aus einem Modell auf dem Bildschirm anzuzeigen. Property Bindings ermöglichen uns, die Eigenschaften der Controls mit bestimmten Werten in einem Modell zu verbinden. Sobald solch eine Verbindung aufgebaut wurde, wird jede Änderung am Wert in einem Modell in der Control-Eigenschaft

widergespiegelt. Da wir derzeit eine bidirektionale Bindung verwenden, gilt dies auch in umgekehrter Reihenfolge.

Sie benötigen zwei Parameter, um eine Bindung zwischen einer Control-Elementeigenschaft und einem Wert im Modell herzustellen:

- den Pfad zum Wert im Modell
- den Namen der Eigenschaft, die Sie binden möchten

Als Nächstes betrachten wir die Methoden zum Binden der Eigenschaft eines Controls. Anschließend gehen wir auf den Einsatz von Datentypen und die Definition eigener Datentypen ein.

5.2.1 Methoden zum Binden der Eigenschaft eines Controls

Es gibt zwei grundsätzliche Methoden, um die Eigenschaft eines Controls zu binden:

- **Control-Einstellungen**
  Sie können das Binding in dem Objekt für initiale Settings im Konstruktoraufruf eines Controls erzeugen, wenn Sie nicht deklarative View-Typen nutzen. Oder Sie können die Binding-Informationen einfach statt eines fixen Werts in das gewünschte Attribut schreiben wenn Sie z. B. einen XML-View verwenden.
- `bindProperty`
  Diese Methode kann für das Control jederzeit im Code verwendet werden, um das System darüber zu informieren, an welchen Wert im Modell eine Eigenschaft gebunden werden soll.

In den nächsten beiden Unterabschnitten gehen wir auf diese beiden Methoden ein und geben Beispiele für deren Einsatz. Dann erläutern wir die Verwendung von Datentypen und die Definition eigener Datentypen.

Control-Einstellungen

Wenn zum Erzeugen des Bindings die Control-Settings verwendet werden, weisen geschweifte Klammern um den Eigenschaftswert das Framework an, dass das normale Parsen des Strings nicht stattfinden soll. Stattdessen soll ein Binding erstellt werden.

In Abbildung 5.4 haben wir ein einfaches Text-Controls mit der Konstruktorfunktion im JavaScript-Code instanziiert:

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

Wir geben einen Wert an den Konstruktor weiter, der der Eigenschaft `text` des Controls zugewiesen ist. Dieser Wert enthält nun den Pfad zu dem Wert im JSON-Modell. Sobald der Wert im JSON-Modell aktualisiert wird, wird auch der Wert der Control-Eigenschaft aktualisiert.

In einem Szenario mit bidirektionaler Bindung funktioniert das auch in die andere Richtung. Wenn Sie das Control `text` durch eine Eingabe ersetzen, erhalten Sie sofort die Möglichkeit, die Modellwerte anzupassen. Zum Beweis können wir ein weiteres Control zu unserer Beispielanwendung hinzufügen: `sap.m.Input`.

In Listing 5.8 instanziieren wir ein Control `sap.m.Input` und platzieren es in *content div*.

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

Listing 5.8 sap.m.Input-Instanziierung

Wenn Sie diesen Code zum Beispiel hinzufügen, sollte die Anwendung wie in Abbildung 5.4 aussehen.

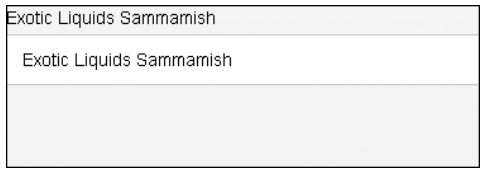


Abbildung 5.4 Beispielanwendung mit sap.m.Input

Dem Eingabefeld wurde derselbe Wert wie dem Text darüber zugewiesen. Bei einer bidirektionalen Bindung steckt aber mehr dahinter, als hier direkt zu sehen ist. Wie zuvor erwähnt, sind beide Controls nun mit demselben Wert aus dem Modell verbunden. Wenn Sie ein Control ändern, wird auch das Modell aktualisiert. Das Modell wiederum informiert alle damit verbundenen Controls, dass der Wert geändert wurde. Diese Benachrichtigung löst ein erneutes Rendern der jeweiligen Controls aus und wird somit in allen Elementen widergespiegelt.

Geben Sie einen anderen Wert in das Textfeld ein, und sehen Sie, was geschieht, wenn Sie die ↵-Taste drücken oder aus dem Feld heraus navigieren. Der eingegebene Wert wird sofort für den Text des Controls `text` übernommen und über dem Eingabefeld angezeigt.

Live-Wert

Der Live-Wert des Eingabefelds wird nur im Wertattribut des Controls reflektiert, wenn ein Änderungsereignis ausgelöst wurde. Dieses Änderungsereignis wird normalerweise ausgelöst, wenn ein Anwender eine Taste wie z. B. ↵ drückt oder das Feld anderweitig durch Klicken oder Navigieren verlässt.

In XML-Views können die spezifischen Eigenschaften eines Controls so gebunden werden wie in den Einstellungen einer Control-Instanziierung in JavaScript. Sie müssen nur geschweifte Klammern in das Attribut am entsprechenden Tag für das Control einfügen:

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

bindProperty

Wenn Sie die Eigenschaft eines Controls nicht bei der Instanziierung, sondern zu einem späteren Zeitpunkt im Code an einen Modellwert binden möchten, können Sie die Methode `bindProperty` verwenden, die in jedem Control in SAPUI5 bereitgestellt wird:

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

Möchten Sie den automatischen Datenfluss unterdrücken und den bidirektionalen Bindungsmodus für ein bestimmtes Control deaktivieren, können Sie dies am Control wie folgt festlegen:

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

In letzterem Beispiel verwenden wir die Objektliteral-Syntax zum Definieren der Bindung, wie wir es für die Bindung im Konstruktoraufruf gezeigt haben.

In der API-Dokumentation wird angegeben, dass die Methode `bindProperty` für jede Klasse verfügbar ist, die von der übergeordneten Klasse `sap.ui.base.ManagedObject` erbt. Für zahlreiche Control-Eigenschaften, Aggregationen und Assoziationen stehen auch typisierte Methoden zur Verfügung. Ob

ein Control eine typisierte Methode für eine Eigenschaft bereitstellt, wird durch das Kennzeichen `bindable` in den Metadaten des Controls festgelegt.

Für unsere Eingabe hat das Control die Werteigenschaft aus der Klasse `InputBase` geerbt. Wenn Sie die Datei *InputBase.js* in der Bibliothek `sap.m` öffnen, können Sie diese Zeilen im Konstruktor sehen, die die Eigenschaft `value` als `bindable` definieren (siehe Listing 5.9).

```
var InputBase = Control.extend("sap.m.InputBase", /
** @lends sap.m.InputBase.prototype */ { metadata: {
  library: "sap.m",
  properties: {
  value: { type: "string", group: "Data", defaultValue: null,
      bindable: "bindable" },
[...]
```

Listing 5.9 Auszug aus sap.m.InputBase

Für jede mit dem Kennzeichen `bindable` definierte Eigenschaft werden folgende Methoden automatisch erzeugt und zum Prototyp der Klasse hinzugefügt:

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

Für unsere Eingabe werden dadurch folgende Methoden verfügbar:

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

Mit diesen Methoden werden dieselben Parameter wie für die untypisierten Varianten ermöglicht. Dies bedeutet, dass wir in den vorangehenden Beispielen auch die gezeigten Codezeilen hätten verwenden können:

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

Die typisierten Funktionen sind praktische Methoden, mit denen Code einfacher geschrieben und gelesen werden kann. Nicht mehr und nicht weniger!

Beachten Sie, dass es ergänzend zur Methode `bindProperty` auch eine Methode `unbindProperty` gibt, mit der die Verbindung der Control-Eigenschaft zum Modell wieder aufgehoben wird. Wenn Sie diese Methode verwenden, wird der letzte in der Eigenschaft am Control festgelegte Wert beibehalten. Er wird nicht mehr aktualisiert, wenn der Wert im Modell geändert wird.

Die für das Property Binding in einem JSON-Modell verwendete Logik wird in der Klasse `sap.ui.model.json.JSONPropertyBinding` implementiert. Ein Großteil ihrer Funktionalität ist für alle Property Bindings in SAPUI5 verfügbar, da sie aus einer allgemeinen Klasse `sap.ui.model.ListBinding.js` vererbt wird. Alle Property Bindings teilen sich deshalb die Getter und Setter für Wert, Typ, Formatierung, externen Wert und Bindungsmodus (siehe Listing 5.10).

```
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 5.10 Vererbte Getter und Setter für Property Bindings

5.2.2 Verwenden von Datentypen

Wenn Sie das Property Binding für bestimmte Modelleigenschaften, z. B. Float-Werte, verwenden, kann es manchmal nützlich sein, den Datentyp nicht nur dem Modell, sondern auch der Benutzeroberfläche bekannt zu machen. Dies kann nützlich sein, wenn Sie z. B. sicherstellen möchten, dass in ein Textfeld nur eine Zahl für einen Preis, eine Menge oder Ähnliches bzw. nur ein Datum für ein Geburtsdatum eingefügt wird.

Um dies und ein gewisses Maß an automatischer Datenformatierung, Parsen und Validierung zu erreichen, können Sie Datentypen für die Bindungen in SAPUI5 verwenden.

In SAPUI5 sind bereits einige einfache Typen verfügbar, die sofort einsatzbereit sind:

- `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`

Kehren wir zum Eingabebeispiel zurück. Wir können einen bestimmten Datentyp, den wir als Parameter verwenden möchten, an den Konstruktor weitergeben (siehe Listing 5.11).

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

Listing 5.11 Festlegen eines Typs im Control-Konstruktoraufruf

Wir können auch den Ansatz `bindProperty` beibehalten und einfach den Typ, wie in Listing 5.12 gezeigt, als einen dritten Parameter an diese Methode weitergeben.

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

Listing 5.12 Festlegen des Typs im Aufruf bindProperty am Control

Betrachten wir nun die Details und Funktionen, wenn ein Datentyp zusammen mit einer Bindung verwendet wird. Jeder einfache Datentyp in SAPUI5 hat drei Funktionen:

- **`formatValue`**
  Wenn eine Control-Eigenschaft an einen Wert im Modell gebunden wird, wird die Funktion `formatValue` ausgeführt, sobald eine Änderung an den Modelldaten im Control wiedergegeben werden soll.
- **`parseValue`**
  Die Funktion `parseValue` agiert als Formatierung in die andere Richtung. Sie wird ausgeführt, wenn ein Eigenschaftswert seitens der Benutzeroberfläche geändert wird und diese Änderung in das Modell zurückgegeben werden soll.
- **`validateValue`**
  Die Funktion `validateValue` wird ausgeführt, um zu ermitteln, ob Werteinschränkungen verletzt wurden.

Alle Funktionen lösen entsprechende Ausnahmen aus, wenn die Anforderungen nicht erfüllt werden, nämlich `FormatException`, `ParseException` und `ValidateException`.

Um auf diese Ausnahmen bei der Auslösung zu reagieren, müssen Sie die Ausnahme nicht selbst erfassen. Stattdessen sollen Sie in solchen Fällen über folgende Controls an die von SAPUI5 ausgelösten Ereignisse verbinden:

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

Sie können problemlos auf eine falsche Benutzereingabe durch Verbinden mit `validationError` reagieren, das z. B. ausgelöst wird, wenn die Eingabe eines Anwenders die durch den Typ definierten Einschränkungen verletzt.

Betrachten wir, was geschieht, wenn wir einen Typ `integer` innerhalb unserer Beispielanwendung verwenden. Der Konstruktor von `sap.ui.model.type.Integer` nutzt zwei optionale Parameter des Typobjekts – `oFormatOptions` und `oConstraints`. Eine neue Instanz kann somit wie folgt instanziiert werden:

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

Wird einer der beiden Parameter festgelegt, werden Standardwerte für den Typ verwendet.

Unter der folgenden Eingabe

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

fügen wir eine weitere Eingabe ein, die die Postleitzahl aus der Adresse des Lieferanten als Wert nutzt (siehe Listing 5.13). Wir erwarten eine Eingabe vom Typ `integer`.

```
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 5.13 Verwenden eines Typs mit Einschränkungen für eine Eingabe

Beachten Sie, dass wir die Formatoptionsparameter für den Konstruktor `Integer` vorläufig als leeres Objekt belassen haben. In Kürze erfahren Sie, wofür der Parameter am besten verwendet wird.

Jetzt verbinden wir die entsprechenden Ereignisse, um den Anwendern Feedback geben zu können, wenn sie eine Falscheingabe machen (siehe Listing 5.14).

```
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 5.14 Verbinden mit den Validierungsereignissen aus dem Typ

Wenn Sie die Anwendung neu laden und eine Zahl eingeben möchten, die größer ist, als es in den Einschränkungen festgelegt wurde, wird ein Alert angezeigt, dass Sie die Mindest- bzw. Maximallänge verletzt haben. Wenn Sie nichtnumerische Zeichen eingeben, wird stattdessen ein Alert aus `parseError` angezeigt.

Bei jedem Fehler, der auftritt, übernimmt der verbundene Event-Handler. Wir können auf den fehlerhaften Wert zugreifen, indem wir einen Parameter aus dem Objekt `oControlEvent` verwenden, das der Event-Handler erhält. Bevor wir den ungültigen Wert aus der Eingabe entfernen, können wir dem Anwender einige Informationen dazu zur Verfügung stellen, was falsch gelaufen ist.

Der neue Wert, den der Anwender festlegen wollte, ist als Parameter `newValue` verfügbar. Ebenso können Sie auf den alten Wert, den Datentyp und die Elemente zugreifen, für die eine Eigenschaft versucht hat, das Modell zu aktualisieren.

Beachten Sie, dass die zusätzlichen Einschränkungen (für `sap.ui.model.type.Integer` gibt es nur die beiden Einschränkungen, die wir verwendet haben) nur berücksichtigt werden, wenn eine Validierung stattfindet. Diese erfolgt nur, wenn versucht wird, das Modell zu aktualisieren.

Es gibt weitere Parameter, die Sie an den Typ `Integer` weitergeben können. Diese Parameter sind allerdings Teil des Musters, das verwendet wird, wenn der Quellwert eines Controls in einen String umgewandelt werden soll, da eine Control-Eigenschaft von diesem Typ ist. Dies bedeutet, dass bei jeder Ausgabe eines Quellwerts des Typs `Integer` mit einer bestimmten Anzahl an Ziffern, z. B. in einer Werteigenschaft `sap.m.Input`, ein Muster angewendet wird, wenn der Wert in diese Control-Eigenschaft geschrieben wird, da die Eigenschaft selbst vom Typ `string` ist.

Für `sap.ui.model.type.Integer` legen Sie z. B. die folgenden Werte fest:

```
minIntegerDigits: 1, // minimal number of non-fraction digits
maxIntegerDigits: 99 // maximal number of non-fraction digits
```

Diese Parameter sind Teil des Objekts `formatOptions`, das wir beim ersten Versuch, diesen Datentyp zu verwenden, leer gelassen haben. Wenn Sie einige dieser Formatoptionen zum Beispiel hinzufügen, können Sie direkt sehen, was geschieht, wenn Sie einen Wert einfügen möchten, der nicht dem Muster folgt.

Legen wir z. B. einen Mindestzahl an Ziffern fest, und ändern wir das Eingabefeld für die Postleitzahl, wie es in Listing 5.15 dargestellt ist.

```
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 5.15 Festlegen der Formatoptionen für einen Typ

Wenn Sie die Anwendung nun erneut laden und z. B. eine einstellige Zahl einfügen möchten, wird der von Ihnen eingegebene Wert gemäß dem festgelegten Muster umgewandelt, sobald das Änderungsereignis zur Eingabe ausgelöst wurde. Der einstellige Eintrag wird dann in 00001 geändert (siehe Abbildung 5.5).

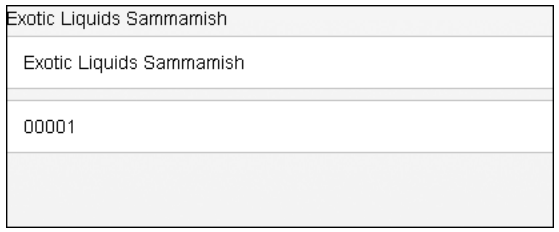


Abbildung 5.5 Anwendung mit Eingabe für die Postleitzahl

Es gibt auch mehrere Optionen für jeden der anderen Typen, und für Float-Werte können Sie definieren, wie viele Dezimalstellen für eine Zahl zulässig sind. Eine vollständige Liste der Optionen erhalten Sie im SAPUI5-Demokit auf der Registerkarte API Reference unter `sap.ui.model.type`.

5.2.3 Definieren eines eigenen Datentyps

Sie haben erfahren, welche wichtigen Methoden in den Datentypen in SAPUI5 enthalten sind. Nun gehen wir darauf ein, wie Sie Ihre eigenen Datentypen definieren können.

Wenn Sie z. B. einen Datentyp an mehr als einer Stelle in Ihrer Anwendung anwenden möchten, können Sie problemlos Ihren eigenen Typ schreiben und hierzu die Klasse `sap.ui.model.SimpleType` erweitern. Sie können dann Ihre spezifischen Anforderungen umsetzen, indem Sie die drei in Abschnitt 5.2.2, »Verwenden von Datentypen«, erläuterten Methoden implementieren.

Sie benötigen z. B. einen speziellen Datentyp für Telefonnummern, der dem Muster +01 1234 456789 folgt, das Sie über die Anwendung hinweg wieder-

verwenden möchten. In diesem Fall ist es sinnvoll, Ihren eigenen Typ zu definieren, da Sie ein bestimmtes Format für Telefonnummern erwarten. Da in der Beispielanwendung auch Anwendereingaben möglich sind, ist es sinnvoll, die Funktion `validateValue` des Typs zu nutzen, um sicherzustellen, dass die eingegebenen Informationen die Formatanforderungen erfüllen. Ansonsten wird ein Fehler ausgegeben und das bereits bekannte Ereignis `validationFailed` ausgelöst.

Erweitern wir zuerst das Modell mit der Telefonnummer für den Lieferanten wie in Listing 5.16 dargestellt.

```
{
  "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": {
        "Street": "NE 40th",
        "City": "Redmond",
        "State": "WA",
        "ZipCode": "98052",
        "Country": "USA",
        "PhoneNumber": "+1-123-123-1235"
      }
    }
  ]
}
```

Listing 5.16 Erweiterte JSON-Modelldaten

Im nächsten Schritt erstellen wir einen neuen Typ, indem wir die Klasse `sap.ui.model.SimpleType` erweitern und die drei zuvor erwähnten Methoden bereitstellen (siehe Listing 5.17).

```
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*)([2-9]\d{2})(\D*)(\d{4})
\D/.test(oValue)) {
      throw new sap.ui.model.ValidateException("phone number must
follow the pattern +1 234-567-890!");
            }
        }
    });
```

Listing 5.17 Definieren eines neuen Typs

Lassen Sie den Wert für die ersten beiden Methoden unverändert, und geben Sie diesen in der im Modell vorliegenden Form zurück. Verwenden Sie bei der dritten Methode einen regulären Ausdruck, um zu sehen, ob die Anwendereingabe mit dem Muster übereinstimmt, das wir für Telefonnummern vordefiniert haben.

Wir können nun den Typ einer neuen Eingabe für die Telefonnummer in der Beispielanwendung auf den eigenen Typ festlegen (siehe Listing 5.18).

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

Listing 5.18 Verwenden eines eigenen Typs

Schließlich verbinden wir noch mit dem Ereignis `validationError`. Dieses Mal verwenden wir die Mitteilung aus der von uns definierten Validierungsausnahme, um den Anwender über den Fehler zu informieren (siehe Listing 5.19).

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



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

Listing 5.19 Verwenden der Validierungsausnahme, um einen Fehler für den Anwender auszugeben

Die Beispielanwendung sieht nun wie in Abbildung 5.6 aus.

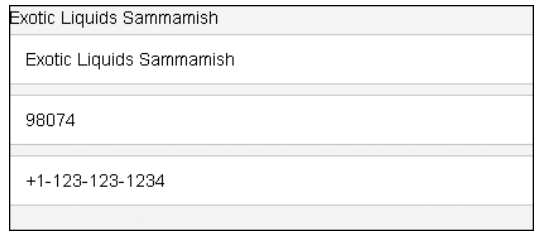


Abbildung 5.6 Beispielanwendung mit hinzugefügter Telefonnummer

Wenn der Anwender jetzt etwas eingibt, das nicht dem regulären Ausdruck aus dem eigenen Typ entspricht, wird dem Anwender über ein Alert mitgeteilt, dass das vordefinierte Muster nicht eingehalten wurde.

5.3 Verwenden von Formatierungen

Neben dem Formatieren von Werten über Datentypen können Sie auch eigene Formatierungen definieren, wenn Sie nicht den Wert im Modell, sondern die Ausgabe in der Benutzeroberfläche beeinflussen möchten. Im Gegensatz zu einem Typ, der wesentlich leistungsstärker ist, ist eine Formatierung generell dort sinnvoll, wo die Formatierung eines Modells unidirektional ist.

Wir können eine Formatierung, wie folgt, als einen Parameter an den Konstruktor des Controls übergeben:

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

Wir können die Funktion auch weitergeben, wenn wir die Methode `bindProperty` für die entsprechende Eigenschaft aufrufen (siehe Listing 5.20).

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

```
        }
    }
})
```

Listing 5.20 Einsatz einer Formatierung für sap.m.Text Control

Beachten Sie, dass in beiden Fällen die Formatierungsfunktion eine anonyme Callback-Funktion ist, die ausgeführt wird, wenn der View instanziiert wird.

Sie können eine Formatierungsfunktion auch nicht anonym in einem View-Controller definieren. Bei einer Wiederverwendung über die Anwendung hinweg kann es sogar sinnvoll sein, eine separate Datei *formatter.js* in der Anwendung verfügbar zu haben, die in mehreren Views referenziert werden kann.

Im ersten Fall müssen Sie die Funktion nur im Controller, wie in Listing 5.21 dargestellt, implementieren.

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

Listing 5.21 Code der Formatierung im Controller

Anstelle der anonymen Funktion aus dem vorangehenden Beispiel können Sie dann diese Formatierung wie folgt verwenden:

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

Zurück in unserer MVC-Anwendung können wir auch die Formatierung nutzen, indem wir sie von Listing 5.21 in den Controller des Detail-Views übertragen. Fügen Sie die Zeilen aus Listing 5.21 in die Datei *Detail.controller.js* ein. Der Detail-Controller sieht nun wie in Listing 5.22 aus.

```
sap.ui.define([
  "sapui5/demo/mvcapp/controller/BaseController"
], function(BaseController) {
  "use strict";

  return BaseController.extend("sapui5.demo.mvcapp.controller.Detai
l", {
```



```
    /* ======================================================= */
    /* lifecycle methods                                       */
    /* ======================================================= */

    /**
     * Called when the worklist controller is instantiated.
     * @public
     */
    onInit: function() {
      this.getRouter().getRoute("detail").attachPatternMatched(this
._onObjectMatched, this);
    },

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

    /**
     * Navigates back to the Master
     * @function
     */
    onNavPress: function() {
      this.myNavBack("master");
    },

    /* ======================================================= */
    /* formatters                                              */
    /* ======================================================= */

    /**
     * Formats a given string to uppercase.
     *
     * @function
     * @param {string} sName string to be formatted
     * @public
     */
    toUpperCase: function(sName) {
      return sName && sName.toUpperCase();
    },

    /* ======================================================= */
    /* internal methods                                        */
    /* ======================================================= */

    /**
     * Binds the view to the object path.
```

```
     *
     * @function
     * @param {sap.ui.base.Event} oEvent pattern match event
     * in route 'object'
     * @private
     */
      _onObjectMatched: function(oEvent) {
      var sObjectPath = "/Suppliers/
" + oEvent.getParameter("arguments").ID;
      this._bindView(sObjectPath);
    },

    /**
     * Binds the view to the object path.
     *
     * @function
     * @param {string} sObjectPath path to the object to be bound
     * @private
     */
    _bindView: function(sObjectPath) {
      var oView = this.getView();
      oView.bindElement(sObjectPath);
    }
  });
});
```

Listing 5.22 Detail-Controller in der MVC-Anwendung

Im Detail-View können wir nun die Formatierung, wie in Listing 5.23 gezeigt, nutzen.

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



```
        <ObjectAttribute
          text="{Address/Country}">
        </ObjectAttribute>
      </ObjectHeader>
    </content>
  </Page>
</mvc:View>
```

Listing 5.23 Detail-View in der MVC-Anwendung

Mit dem Punkt (.) vor dem Namen der Formatierungsfunktion wird dem Framework mitgeteilt, dass es innerhalb des aktuellen Controllers nach einer Formatierung suchen soll.

Durch das Ausführen der Anwendung und Laden der Details eines Lieferanten wird ein Bildschirm wie in Abbildung 5.7 erzeugt. Dort sehen Sie, dass der Lieferantenname jetzt in Großbuchstaben angezeigt wird. Der Wert des Modells bleibt von dieser Änderung unberührt.

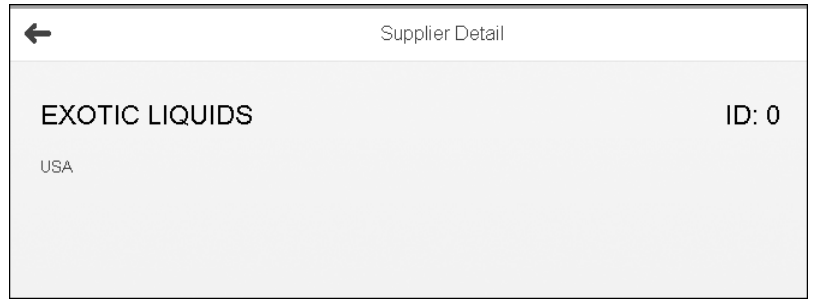


Abbildung 5.7 Detail-View mit der Formatierung in Aktion

Die beste Wiederverwendbarkeit erzielen Sie jedoch, indem Sie die Formatierungsfunktionen in eine separate Datei übertragen, die über die Anwendung hinweg zur Verfügung steht. Zum Schreiben großer Anwendungen hat es sich bewährt, alle Formatierungen zu sammeln, die Sie in mehr als einem View verwenden möchten. Definieren wir nun unsere eigene Datei *formatter.js* für unsere Anwendungen. Wir legen diese Datei in einem neuen Ordner *model* ab, da sie zu einer Funktionalität gehört, die sich auf das Modell und dessen Werte bezieht. Wenn Sie die Funktion `toUpperCase` in diese Datei übertragen, sieht der gesamte Code wie in Listing 5.24 aus.

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

  return {
```

```
    /**
     * Formats a given string to uppercase.
     *
     * @function
     * @param {string} sName string to be formatted
     * @public
     */
    toUpperCase: function(sName) {
      return sName && sName.toUpperCase();
    }
  };

});
```

Listing 5.24 formatter.js für die MVC-Anwendung

Wird diese Formatierung aus einem Controller geladen, gibt sie ein Objekt mit der Funktion `toUpperCase` zurück, die als Member-Funktion verfügbar ist.

Wir stellen nun sicher, dass den Controllern diese zusätzliche Funktion bekannt ist, indem wir diese in die Controller-Definition des Detail-Views einfügen. Der Controller sieht jetzt wie in Listing 5.25 aus.

```
sap.ui.define([
  "sapui5/demo/mvcapp/controller/BaseController",
  "sapui5/demo/mvcapp/model/formatter"
], function(BaseController, formatter) {
  "use strict";
  return BaseController.extend("sapui5.demo.mvcapp.controller.Detai
l", {

    formatter: formatter,
    /* =========================================================== */
    /* lifecycle methods                                           */
    /* =========================================================== */

    /**
     * Called when the worklist controller is instantiated.
     * @public
     */
    onInit: function() {
      this.getRouter().getRoute("detail").attachPatternMatched
      (this._onObjectMatched, this);
    },
```



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

    /**
     * Navigates back to the Master
     * @function
     */
    onNavPress: function() {
      this.myNavBack("master");
    },

    /* =========================================================== */
    /* internal methods                                            */
    /* =========================================================== */

    /**
     * Binds the view to the object path.
     *
     * @function
     * @param {sap.ui.base.Event} oEvent pattern match event in
     * route 'object'
     * @private
     */
    _onObjectMatched: function(oEvent) {
      var sObjectPath = "/Suppliers/
" + oEvent.getParameter("arguments").ID;
      this._bindView(sObjectPath);
    },

    /**
     * Binds the view to the object path.
     *
     * @function
     * @param {string} sObjectPath path to the object to be bound
     * @private
     */
    _bindView: function(sObjectPath) {
      var oView = this.getView();
      oView.bindElement(sObjectPath);
    }
  });
});
```

Listing 5.25 Einfügen der Formatierung in den Controller

Beachten Sie, dass innerhalb des Controllers die entsprechende Formatierungsfunktion gelöscht wurde.

Im View muss der Zugriff auf die Formatierung nun etwas anders erfolgen. Wie erwähnt, laden wir die Formatierungsdatei und übergeben das zurückgegebene Objekt als Parameter an den Controller. Die Formatierung ist dann als Eigenschaft `formatter` im Controller verfügbar. Wir müssen deshalb `ObjectHeader` in der Datei *Detail.view.xml* ändern, in der wir die Formatierung, wie in Listing 5.26 dargestellt, verwenden möchten.

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

Listing 5.26 Auszug aus Detail.view.xml

Wenn alles ordnungsgemäß ausgeführt wurde, wird nach dem erneuten Laden der Anwendung dasselbe Ergebnis im Browser angezeigt. Wir haben nichts an der Art der Formatierung geändert, sondern nur die Wiederverwendbarkeit verbessert, indem wir den Formatter an einem anderen Ort gespeichert haben. Wir könnten diese jetzt in mehrere Controller in unserer Anwendung einfügen.

Wir zeigen nun weitere Informationen über unseren Lieferanten an und fügen hierzu einige zusätzliche Felder zum Objekt-Header im Detail-View hinzu. Wir möchten die Adresse und die Telefonnummer eines Lieferanten mit dem Datentyp aus dem vorangehenden Beispiel für die Telefonnummer einfügen.

Erstellen Sie eine Datei *types.js*, die der Datei *formatters.js* ähnlich ist, innerhalb des Verzeichnisses *model* der MVC-Anwendung, kopieren Sie den Dateityp für die Telefonnummer aus Abschnitt 5.2.3, »Definieren eines eigenen Datentyps«, und fügen Sie diesen ein. Wir verwenden wieder die Syntax `define` und sorgen dafür, dass das Modul ein Objekt mit spezifischen Typen als Eigenschaften zurückgibt.

Der vollständige Code innerhalb der Datei *types.js* sieht wie in Listing 5.27 aus.


```
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 +1 234-567-890!");
        }
      }
    })
  };
});
```

Listing 5.27 Separate Datei types.js, die in der Anwendung verwendet werden soll

Als Nächstes müssen wir das Modell in dieser Anwendung ein wenig erweitern, wie wir es bereits in der kleineren Beispielanwendung umgesetzt haben. Dies ermöglicht, dass die Informationen angezeigt werden können. Die Datei *model.json* sieht jetzt wie in Listing 5.28 aus.

```
{
  "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":{
        "Street":"NE 40th",
        "City":"Redmond",
        "State":"WA",
        "ZipCode":"98052",
        "Country":"USA"
        "PhoneNumber": "+1-123-123-1235"
      },
[...]
}
```

Listing 5.28 Modellauszug – Telefonnummer zu den Lieferanten hinzugefügt

Nun müssen wir dem Controller erneut mitteilen, wo der im View zu verwendende Typ vorliegt. Wie bei der Formatierung fügen Sie die Typen einfach den Modulen hinzu, die im Abschnitt `define` des Controllers geladen wurden (siehe Listing 5.29).

```
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.Det
ail", {

      formatter: formatter,
      types: types,
[...]
```

Listing 5.29 Auszug aus Detail.controller.js

Schließlich können wir die neuen Modellwerte mit dem Datentyp und der Formatierung kombinieren und so den Detail-View erweitern. Fügen Sie das zusätzliche Objektattribut, wie gezeigt, zum View hinzu:

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


Führen Sie die Anwendung aus, und wählen Sie einen Lieferanten aus der Liste aus. Der Bildschirm sollte wie in Abbildung 5.8 aussehen.

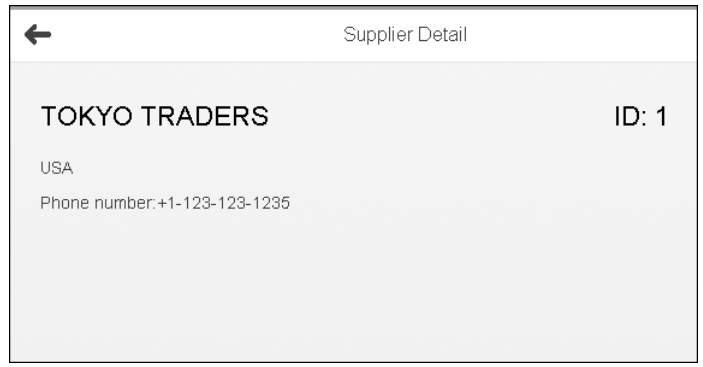


Abbildung 5.8 Detail-View mit Typ für die Telefonnummer

5.4 Aggregation Binding

Bisher haben wir die Property Bindings in der Anwendung betrachtet. Es gibt aber einen weiteren Binding-Typ namens *Aggregation Binding*, mit dem Elemente für eine Control-Aggregation aus den Modelldaten automatisch erzeugt werden.

Ein Beispiel hierzu finden wir in *Master.view.xml* (siehe Listing 5.30).

```
<Table
  id="table"
  width="auto"
  class="sapUiResponsiveMargin"
  items="{/Suppliers}"
  noDataText="No data"
  growing="true"
  growingScrollToLoad="true">
[...]
<items>
  <ColumnListItem
    type="Navigation"
    press="onListPress">
    <cells>
    <ObjectIdentifier
     text="{ID}"/>
    <ObjectIdentifier
     text="{Name}"/>
</cells>
```

```
</ColumnListItem>
</items>
```

Listing 5.30 Auszug aus dem Master-View mit Aggregation Binding

Die Elementaggregation für die Tabelle wird automatisch mit einem Element pro Modellknoten befüllt, der durch den im Elementattribut in Listing 5.30 vergebenen Pfad referenziert wurde. Für jedes Element muss eine Vorlage angewendet werden. Diese Vorlage wird dann für jeden Wert im Modell geklont, und der Bindungspfad des aggregierten Elements wird auf den entsprechenden Modellknoten festgelegt.

Verwenden Sie, wie in Kapitel 4, »Aufbau von MVC-Anwendungen«, erwähnt, den in Listing 5.31 dargestellten Code, wenn Sie das übergeordnete Control innerhalb eines JavaScript-Views instanziieren und die Aggregation-Binding-Parameter direkt an den Control-Konstruktor übergeben möchten.

```
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 5.31 Aggregation Binding in JavaScript

Innerhalb des XML-Views ist diese Vorlage deklarativ innerhalb der Aggregationsgrenzen `<items></items>` festgelegt.

Einige Controls bieten eine Standardaggregation, die Sie aus den Metadaten des Controls in der entsprechenden Control-Klasse identifizieren können. Wenn Sie `sap.m.ObjectHeader` in der Beispielanwendung betrachten, sind die folgenden Zeilen in den Metadaten fett markiert (siehe Listing 5.32):

```
[Property definitions here...]
  },
  defaultAggregation : "attributes",
  aggregations : {
[Aggregation definitions follow here...]
```

Listing 5.32 In den Metadaten eines Controls definierte Standardaggregation

Diese Standardaggregationen werden automatisch eingesetzt, wenn Sie die Aggregation nicht innerhalb eines XML-Views festlegen (siehe Listing 5.33).

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

Listing 5.33 Auszug aus Detail.view.xml ohne explizite Nutzung der Standardaggregation

Über den Code aus Listing 5.33 und dem folgenden Listing 5.34 wird dasselbe DOM erzeugt, selbst wenn Letzteres zusätzliche Tags zum Öffnen und Schließen der Standardaggregation enthält:

```
<ObjectHeader id="objectHeader"
  title="{
    path: 'Name',
   formatter: '.formatter.formatUpperCase'
```

```
  }"
  number="ID: {ID}">
  <attributes>
    <ObjectAttribute  text="{Address/Country}">
      </ObjectAttribute>
        <ObjectAttribute text="{
          path: 'Address/PhoneNumber',
          type: '.types.PhoneNumber'
         }" />
    </attributes>
</ObjectHeader>
```

Listing 5.34 Auszug aus Detail.view.xml mit expliziter Nutzung der Standardaggregation

Üblicherweise wird der Inhalt der Standardaggregationen im XML-View ohne die umgebenden Tags eingefügt, die normalerweise den Beginn und das Ende einer Aggregation anzeigen. Die »Begrenzungs-Tags« müssen nur für diejenigen Aggregationen explizit angegeben werden, die nicht als Standard für das Eltern-Control definiert sind.

Beachten Sie, dass innerhalb eines Aggregation Bindings erstellte Controls am Lebenszyklus der übergeordneten Elemente beteiligt sind. Sie werden instanziiert, wenn die übergeordneten Controls instanziiert werden, und gemeinsam mit den übergeordneten Elementen zerstört.

Wir erweitern unsere Beispielanwendung um ein weiteres Aggregation Binding im Detail-View, indem wir ein weiteres Control `sap.m.Table` hinzufügen, das alle von einem bestimmten Lieferanten angebotenen Produkte enthält.

Diese Tabelle fügen wir zur Datei *Detail.view.xml* direkt unter dem schließenden Tag `</ObjectHeader>` hinzu (siehe Listing 5.35).

```
<Table id="table"
  width="auto"
  class="sapUiResponsiveMargin"
  items="{Products}"
  noDataText="No Data"
  growing="true"
  growingScrollToLoad="true">
  <headerToolbar>
    <Toolbar>
      <Title id="tableHeader" text="Suppliers Products" />
    </Toolbar>
  </headerToolbar>
  <columns>
    <Column id="idColumn">
      <header>
        <Text text="ID:" id="IDColumnTitle" />
      </header>
    </Column>
    <Column id="nameColumn">
      <header>
        <Text text="Name:" id="nameColumnTitle" />
      </header>
    </Column>
    <Column id="priceColumn" hAlign="Right">
      <header>
        <Text text="Price:" id="priceColumnTitle" />
      </header>
    </Column>
  </columns>
  <items>
    <ColumnListItem>
      <cells>
        <ObjectIdentifier text="{ID}" />
        <ObjectIdentifier text="{Name}" />
        <ObjectNumber number="{Price}" unit="USD" />
      </cells>
    </ColumnListItem>
  </items>
</Table>
```

Listing 5.35 Produkttabelle im Detail-View

Wenn Sie diese Aggregation hinzufügen, wird ein Element für jedes mit dem Lieferanten im Modell assoziierte Produkt instanziiert. Die Anwendung sollte jetzt wie in Abbildung 5.9 aussehen.

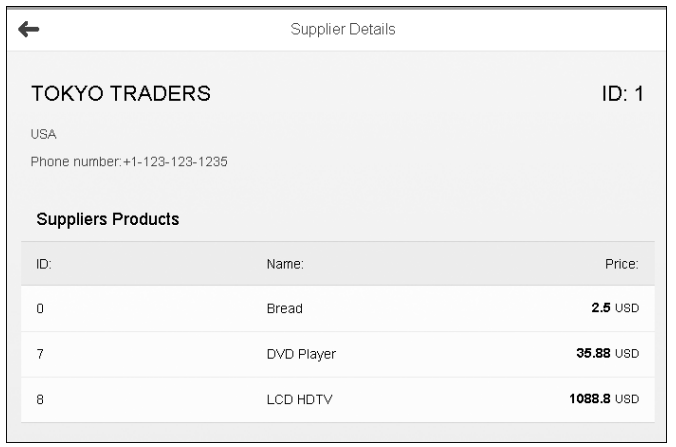


Abbildung 5.9 Detail-View mit Produkten

5.4.1 bindAggregation

Anstatt die Aggregationsvorlage zu deklarieren und innerhalb des XML-Views zu binden, können wir die entsprechende Methode auf dem übergeordneten Control, d. h. der Funktion `bindAggregation`, verwenden. Diese Methode kann wie die Methode `bindProperty` eingesetzt werden, wenn Sie die gewünschte Aggregation nicht bei der Instanziierung des übergeordneten Controls, sondern zu einem späteren Zeitpunkt im Code verwenden möchten.

Diese Funktion enthält zwei Parameter: den Namen der Aggregation und ein Objekt `oBindingInfo`. Dieses Objekt selbst kann mehrere Parameter umfassen. Einige davon betrachten wir in den folgenden Unterabschnitten.

Hinsichtlich der Methode `bindProperty` bieten einige Controls typisierte Methoden für das Aggregation Binding an. In den Metadaten des Controls werden diese Methoden durch das Kennzeichen `bindable` bei der Aggregationsdefinition bestimmt. Beachten Sie, dass die Methoden durch die übergeordnete Klasse `sap.ui.base.ManagedObject` automatisch erzeugt werden, jedoch beim spezifischen Control übersteuert werden können.

Im Fall von `sap.m.Table` aus dem vorangehenden Beispiel wäre folgender Aufruf möglich gewesen:

```
oTable.bindItems("Products", oBindingInfo);
```

Ergänzend zur Methode `bindAggregation` gibt es auch eine Methode `unbindAggregation`, die analog zur bereits bekannten Methode `unbindProperty` ist.

5.4.2 Verwenden einer Factory

Ein großer Vorteil des Programmieransatzes ist, dass Sie eine Factory zusammen mit der Funktion `bindAggregation` verwenden können. Ein häufiger Anwendungsfall ist die dynamische Erstellung der Benutzeroberfläche, z. B. wenn Sie zusätzliche oder andere untergeordnete Controls anzeigen lassen möchten, falls bestimmte Kriterien erfüllt werden.

Angenommen, wir möchten in unserer Anwendung nicht alle Produkte gleich anzeigen lassen und die Anzeige für die Produkte mit bestimmten Zutaten anpassen. Dieser Teil der Übung weicht ein wenig vom Pfad ab, den die Beispielanwendung für das restliche Kapitel nehmen wird. Stellen Sie also sicher, dass Sie eine Kopie Ihrer Arbeit speichern, bevor Sie die nächsten Schritte durchführen. Wir kehren nach diesem Abschnitt zum eigentlichen Beispiel zurück.

Zuerst erweitern wir das JSON-Modell, um diese Informationen wiederzugeben, entfernen dann das deklarative Aggregation Binding innerhalb unseres Views und setzen stattdessen die Produktaggregation programmatisch um, die wir im Controller anzeigen lassen möchten (siehe Listing 5.36).

```
{
  "Suppliers": [{
    "ID": 0,
    "Name": "Exotic Liquids",
    "Address": {
      "Street": "NE 228th",
      "City": "Sammamish",
      "State": "WA",
      "ZipCode": "98074",
      "Country": "USA",
   "Phone Number":"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": 20.9
    }, {
      "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
    }, {
      "ID": 4,
      "Name": "Fruit Punch",
      "Description": "Mango flavor, 8.3 Ounce Cans (Pack of 24)",
      "ReleaseDate": "2003-01-05T00:00:00",
      "DiscontinuedDate": null,
      "Rating": 3,
      "Price": 22.99
    }, {
      "ID": 5,
      "Name": "Cranberry Juice",
      "Description": "16-Ounce Plastic Bottles (Pack of 12)",
      "ReleaseDate": "2006-08-04T00:00:00",
      "DiscontinuedDate": null,
      "Rating": 3,
      "Price": 22.8
    }, {
      "ID": 6,
      "Name": "Pink Lemonade",
      "Description": "36 Ounce Cans (Pack of 3)",
      "ReleaseDate": "2006-11-05T00:00:00",
      "DiscontinuedDate": null,
      "Allergens": "Milk, Soy, Edible Nuts, Gluten",
      "Rating": 3,
      "Price": 18.8
    }]
  }, {
    "ID": 7,
    "Name": "Green Lemonade",
    "Description": "36 Ounce Cans (Pack of 3)",
    "ReleaseDate": "2006-11-05T00:00:00",
    "DiscontinuedDate": null,
    "Allergens": "Milk, Soy, Edible Nuts, Gluten",
    "Rating": 3,
    "Price": 18.8
  }]
},

[...]
]
}
```

Listing 5.36 Geändertes JSON-Modell, um zusätzliche Produktinformationen anzuzeigen

Einige Produkte wurden um Informationen über enthaltene Allergene erweitert. Um diese Informationen deutlich in der Produktliste hervorzuheben,

fügen wir ein Control zur Elementliste hinzu, wenn diese Information im Modell enthalten ist.

Wir entfernen zunächst einen Teil der Tabellendefinition aus dem vorangehenden Beispiel, sodass der Detail-View jetzt wie in Listing 5.37 aussieht.

```
<mvc:View controllerName=
"sapui5.demo.mvcapp.controller.Detail" xmlns:mvc=
"sap.ui.core.mvc" xmlns="sap.m">
  <Page id="page" navButtonPress="onNavPress" showNavButton=
"true" title="Supplier Details">
    <content>
      <ObjectHeader id="objectHeader" title="{
       path: 'Name',
    formatter: '.formatter.formatUpperCase'
  }" number="ID: {ID}">
        <attributes>
          <ObjectAttribute text="{Address/Country}">
          </ObjectAttribute>
          <ObjectAttribute text="{
              path: 'Address/PhoneNumber',
              type: '.types.PhoneNumber'
           }" />
        </attributes>
      </ObjectHeader>
      <Table id="table" width="auto" class=
"sapUiResponsiveMargin" noDataText="No Data" growing=
"true" growingScrollToLoad="true">
        <headerToolbar>
          <Toolbar>
            <Title id="tableHeader" text="Suppliers Products" />
          </Toolbar>
        </headerToolbar>
        <columns>
          <Column id="idColumn">
            <header>
              <Text text="ID:" id="IDColumnTitle" />
            </header>
          </Column>
          <Column id="nameColumn">
            <header>
              <Text text="Name:" id="nameColumnTitle" />
            </header>
          </Column>
          <Column id="priceColumn" hAlign="Right">
            <header>
              <Text text="Price:" id="priceColumnTitle" />
```

```
            </header>
          </Column>
        </columns>
      </Table>
    </content>
  </Page>
</mvc:View>
```

Listing 5.37 Detail-View ohne Elementaggregation für die Tabelle

Wenn Sie die Anwendung nun im Browser neu laden, wird die Tabelle weiterhin angezeigt. Allerdings werden keine Einzelposten angezeigt, sondern nur `noDataText`.

In der Funktion `_onObjectMatched` im Detail-Controller rufen wir nun wie folgt eine weitere private Funktion auf:

```
createProductsAggregation(sObjectPath);
```

Danach müssen wir diese Funktion implementieren, mit der die Elementaggregation in der Tabelle erstellt wird. In dieser Funktion möchten wir ein bestimmtes Control verwenden, nämlich `VerticalLayout` aus der Bibliothek `sap.ui.layout`. Wir müssen sicherstellen, dass dieses Control dem Controller bekannt ist. Daher fügen wir es in den `define`-Teil des Controllers ein (siehe Listing 5.38).

```
_createProductsAggregation: function() {
  var oTable = this.getView().byId("table");

  oTable.bindAggregation("items", "Products", function(sId,
    oContext) {
    var sAllergens = oContext.getProperty("Allergens");
    var oColumnListItem = new sap.m.ColumnListItem(sId);
    oColumnListItem.addCell(new sap.m.ObjectIdentifier({
      text: "{ID}"
    }));

    if (sAllergens) {
      // we have found allergens, so we provide a VerticalLayout
      // instead of just displaying the product name. The
      // VerticalLayout 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 {
      // 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 5.38 Binden der Elementaggregation mit einer Factory

Sie können im Code erkennen, dass wir nun die Controls innerhalb der Aggregation instanziieren, indem wir eine Factory-Funktion an die Methode `bindAggregation` weitergeben, die wir im übergeordneten Control aufrufen. Wird das übergeordnete Control nun instanziiert und die Aggregation erstellt, wird die Factory für jeden Eintrag im Modell ausgeführt. Abhängig davon, ob ein Produkt ein Allergen enthält, wird entweder nur der Name des Produkts angezeigt oder eine Spalte mit dem Control-Typ `sap.ui.layout.VerticalLayout` erstellt, mit der wir mehr als ein Inhalts-Control für das vertikale Layout festlegen können.

Wir verwenden dann einfach zwei Text-Control, die beide in die Content-Aggregation von `VerticalLayout` einfließen. So wird die Leistungsfähigkeit beim Einsatz von Factorys im Aggregation Binding deutlich, da Sie als Entwickler entscheiden können, ob Sie dieselben Informationen, abhängig von den Daten, in unterschiedlicher Form anzeigen lassen oder sogar ein vollständiges Control durch ein anderes Control ersetzen möchten.

Wenn Sie die Seite neu laden, können Sie sehen, dass es Produkte mit zusätzlichen Informationen innerhalb eines anderen Controls in der Zelle mit der Beschreibung gibt, während andere Produkte ohne zusätzliche Informationen angezeigt werden.

Die über die Factory für die Aggregation erzeugten Controls verhalten sich wie jedes andere Aggregationselement. Dies bedeutet, dass sie Teil des Lebenszyklus des übergeordneten Elements sind. Mit den neu hinzugefügten Elementen sollte der Detail-View der Anwendung wie in Abbildung 5.10 aussehen.

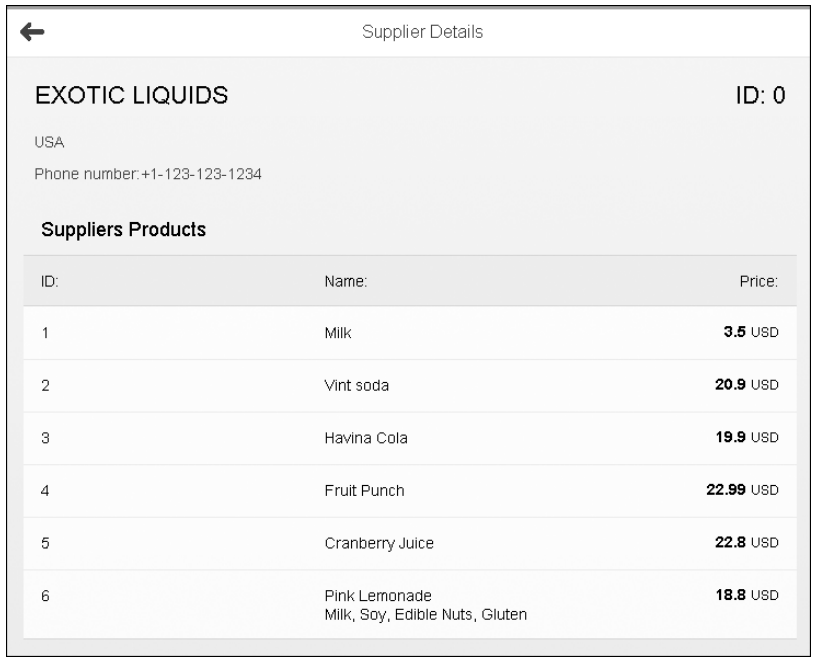


Abbildung 5.10 Produkte des Lieferanten mit Informationen über Allergene

5.5 Element Binding

Sie haben gesehen, dass die Informationen, die beim Umschalten vom Master- auf den Detail-View angezeigt werden, davon abhängen, welchen Lieferanten wir aus der Tabelle ausgewählt haben. Im Aggregation Binding haben wir den relativen Bindungspfad zu den Produkten verwendet. Um diesen relativen Bindungspfad an den ausgewählten Lieferanten anzupassen, muss dem übergeordneten Element des Controls `List`, das die Aggregation enthält, mitgeteilt werden, wie dieser Pfad korrekt aufgelöst wird. Hierzu verwenden wir einen dritten Bindungstyp: *Element Binding*.

Ein Element Binding ermöglicht es uns, einen Kontext für alle relativen Bindungen innerhalb eines Elements zu erstellen. Wir möchten dies anhand eines Beispiels verdeutlichen.

Innerhalb des Detail-Views gibt es einige Property Bindings, die immer die Informationen über den ausgewählten Lieferanten anzeigen. Dies gilt auch für `sap.m.List`, für die wir unsere Elementaggregation definiert haben, damit die Produkte eines bestimmten Lieferanten angezeigt werden. Keiner dieser Pfade beginnt mit einem Schrägstrich, was darauf hinweist, dass uns ein relativer Bindungspfad vorliegt.

Ermitteln wir nun, woher die Informationen darüber kommen, wie diese Pfade aufgelöst werden: Wenn Sie den Code, der den Detail-Controller instanziiert, etwas genauer betrachten, sehen Sie ein in der Methode verwendetes Element Binding, in dem das Ereignis `patternMatched` verarbeitet wird (siehe Listing 5.39).

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

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

Listing 5.39 Binden eines Views an ein Element

Wird ein Lieferant im Master-View angeklickt, werden die Informationen zur Lieferanten-ID über ein Routing an den Detail-View weitergegeben. Der Parameter ist dann in dem Ereignis verfügbar, das an den entsprechenden Event-Handler übergeben wird.

Wir können somit durch Zusammenbringen der Pfadinformationen bestimmen, welcher Lieferant im Detail-View angezeigt werden soll. Hierzu wird innerhalb des Ereignisses `onObjectMatched` der Name der Sammlung im Modell mit der ID aus dem Ereignisparameter kombiniert.

Als Nächstes nutzen wir diese Informationen innerhalb unserer privaten Methode `_bindView`, in der wir `bindElement` explizit im View aufrufen. Über die Funktion `bindElement` wird dann ein Bindungskontext für ein SAPUI5-Control erstellt und der Modellname sowie der Pfad zu einem Element in einem Konfigurationsobjekt empfangen. Dies löst eine Aktualisierung der UI-Controls aus, die wir mit den Modellfeldern verbunden haben.

Wird das Element Binding in einen anderen Bindungskontext gesetzt, z. B. wenn wir zu einem anderen Lieferanten wechseln, werden alle Controls innerhalb des Views mit den entsprechenden Werten aktualisiert.

Die Funktion `bindElement` wird aus der Klasse `sap.ui.base.Element` vererbt und kann in jedem untergeordneten Element aufgerufen werden. Wie bei den anderen Bindungstypen gibt es auch eine Funktion `unbindElement`, mit der ein bestimmter, für ein Element erstellter Bindungskontext entfernt werden kann.

5.6 Expression Binding und berechnete Felder

Sie haben erfahren, wie Werte aus den Modellen für unterschiedliche Bindungstypen verwendet und wie diese entweder über Formatierungsfunktionen oder einen Datentyp formatiert werden. Bisher haben wir allerdings mit jeweils nur einem Wert gearbeitet.

Wie kombinieren wir aber mehrere Werte aus dem Modell in einer Control-Eigenschaft? Hier kommen dann *Expression Bindings* oder *berechnete Felder* ins Spiel. Wir kombinieren nun einige der Lieferanteninformationen im Detail-View an einem Ort. Hierzu verwenden wir die Stadt und den Bundesstaat als Beispiel.

5.6.1 Berechnete Felder

Berechnete Felder können an die Eigenschaft weitergegeben werden, die Sie, ähnlich wie die herkömmliche Bindung, binden möchten. Der einzige Unterschied ist, dass Sie nun mehr als einen Teil des Werts übergeben müssen, weshalb Sie diese bei der Implementierung als genau solche festlegen können: als Teile.

Im Detail-View ändern wir `ObjectHeader` mit den in Listing 5.40 gezeigten Lieferanteninformationen.

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

Listing 5.40 Objekt-Header im Detail-View

Wir geben einfach zwei Werte an den Detail-View weiter, woraufhin in SAPUI5 eine automatische Verkettung der beiden Werte stattfindet. Im Framework wird automatisch ein Composite Binding erstellt, und mit diesem Composite Binding werden die Daten kombiniert und in der Benutzeroberfläche bereitgestellt.

Wenn Sie die Seite jetzt neu laden, wird dadurch das berechnete Feld, wie in Abbildung 5.11 dargestellt, angezeigt.

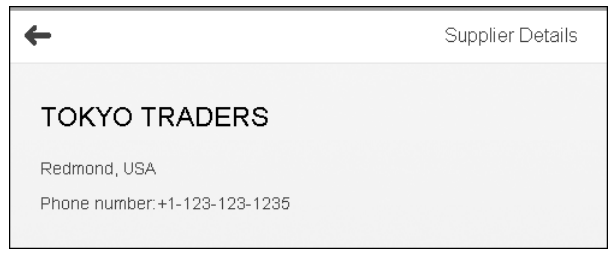


Abbildung 5.11 Objekt-Header – berechnetes Feld

Möchten Sie diese Art der Bindung innerhalb des JavaScript-Codes verwenden, müssen Sie die Teile der Bindung, wie in Listing 5.41 gezeigt, explizit festlegen.

```
var oObjectHeaderAttribute = new sap.m.ObjectAttribute({
    text: {
        parts: [
                {path: "Address/
City", type: new sap.ui.model.type.String()},
                {path: "Address/Country"}
               ]
          }
});
```

Listing 5.41 Erstellen eines Objekt-Headers

Sie können für jeden Pfad einen anderen Typ bestimmen. Dies ist jedoch nicht obligatorisch. In diesem Fall müssen Sie sich allerdings selbst um die Formatierung kümmern, weshalb Sie hier auch eine eigene Formatierung einreichen können. Ansonsten wird vom Framework der standardmäßige Verkettungsmechanismus verwendet, bei dem die beiden Strings einfach miteinander kombiniert werden.

Die Übergabe einer Formatierung ist einfach. Sie können sie einfach innerhalb eines Objektliterals wie in Listing 5.42 oder als Parameter in der (typisierten oder untypisierten) Methode `bindProperty` am Control übergeben.

```
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 5.42 Übergeben der Formatierung an die Bindungsmethode

Sie können beliebig viele Werte in einer berechneten Bindung kombinieren, und Sie müssen sich nur über die Art der Verkettung kümmern, die innerhalb dieser Funktion zum Einsatz kommen soll. Wird das Layout jedoch komplexer, möchten Sie möglicherweise auf mehr als ein Control zurückgreifen, meist in Kombination mit einem SAPUI5-Layout-Control. Versuchen Sie nicht, hier HTML-Code auszugeben. Dieser ist schwierig zu formatieren und kann sogar die Struktur Ihrer Seite zerstören.

[»] Unidirektionale Verwendung berechneter Felder

Wenn Sie berechnete Felder verwenden, müssen Sie sich bewusst sein, dass dies nur unidirektional funktioniert. Aktuell gibt es keinen Mechanismus, der eine bidirektionale Bindung für solche Felder erlaubt, da der Algorithmus, mit dem die potenziellen Anwendungsfälle für die Rückwärtsberechnung berechnet würde, so komplex wäre, dass das Framework diesen nicht mehr implementieren könnte.

Wenn Sie den Anwendern die Möglichkeit geben möchten, den Wert einer Eingabe mit einem berechneten Feld zu ändern, müssen Sie die Logik implementieren, die diese Änderungen im Modell selbst wiedergeben kann.

5.6.2 Expression Binding

Expression Binding ist eine weitere Data-Binding-Funktion in SAPUI5, die anstelle einer Formatierung verwendet werden kann, wenn die Bindung auf bestimmten Bedingungen basieren muss. Wenn Sie eine ziemlich einfache Formatierung programmieren, mit der lediglich zwei Werte verglichen werden, sollten Sie darüber nachdenken, ob ein Expression Binding nicht auch ausreichend wäre.

Sie könnten den Code aus der Factory in Abschnitt 5.4.2, »Verwenden einer Factory«, umschreiben und einen Ausdruck verwenden, um zu bestimmen, ob das Text-Control mit den Allergenen für ein Produkt angezeigt werden soll oder nicht. In diesem Fall müssten Sie nur den View ein wenig ändern, statt den Controller zu erweitern. Der Elementteil im Detail-View würde dann wie in Listing 5.43 aussehen.

```
<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 5.43 Verwenden eines Expression Bindings anstelle einer Formatierung

Das Expression Binding wird nur empfohlen, wenn die Ausdrücke nicht übermäßig komplex sind. Der Nachteil dieses Ansatzes ist, dass das vertikale Layout weiterhin als solches gerendert würde, selbst wenn eines der darin enthaltenen Controls nicht sichtbar wäre und somit das DOM mit unnötigen Elementen belasten würde. Bei komplexeren Anwendungsfällen sind die anderen von uns erläuterten Implementierungsoptionen – Formatierungen oder Factorys für Aggregation Bindings – möglicherweise besser geeignet.

Sie können verschiedene Operatoren innerhalb von Ausdrücken einsetzen – u. a. Equality-Operatoren, Multiplication-Operatoren und additive Operatoren. Im SAPUI5-Demokit auf der Registerkarte API Reference finden Sie weitere Informationen.

5.7 Ressourcenmodelle und Internationalisierung

Ein *Ressourcenmodell* ist eine Sonderform des clientseitigen Modells, das für die Internationalisierung in SAPUI5 verwendet wird. Es agiert als Wrapper für Ressourcenbündel. Ein *Ressourcenbündel* ist eine Datei zum Speichern übersetzbarer Texte, die in der Anwendung zum Einsatz kommen. Diese Dateien haben eine einfache interne Struktur, die eine Reihe von Name-Wert-Paaren enthält, die durch ein Gleichheitszeichen getrennt sind. Beispiel:

```
appTitle=Suppliers And Products
```

Es ist wichtig zu verstehen, dass es innerhalb einer Datei *.properties* kein Konzept für eine Objekthierarchie gibt. Die Struktur ist vollkommen flach.

In diesem Abschnitt betrachten wir die Dateispeicherorte, Dateinamenskonventionen und die Codepage für Ressourcenbündel. Danach setzen wir ein Ressourcenmodell in unserer Beispielanwendung ein.

5.7.1 Dateispeicherort

Ressourcenbündel-Dateien werden normalerweise im Ordner *i18n* innerhalb der Verzeichnisstruktur der Anwendung abgelegt. Diese Dateien enden alle mit dem Suffix *.properties*, weshalb sie auch als »dot properties«-Dateien bezeichnet werden.

Wenn wir unsere Beispielanwendung genau betrachten, erkennen wir, dass es einige Texte gibt, die nicht Teil der im JSON-Modell bereitgestellten Geschäftsdaten sind. Vielmehr gibt es Texte, die als Bezeichnungen oder Titel dienen und übersetzt werden sollten, falls der Anwender die Anwendung in einer anderen Sprache ausführen möchte.

5.7.2 Dateinamenskonvention

SAPUI5-Ressourcenbündel-Dateien folgen derselben Namenskonvention, die auch für Java-Ressourcenbündel verwendet werden. Es wird angenommen, dass jede Datei, die mit *.properties* endet, einfache Name-Wert-Paare

enthält, deren Sprache durch Gebietsschemabezeichner im Dateinamen bestimmt werden. In Tabelle 5.1 werden einige Beispiele für Ressourcenbündel-Dateien aufgeführt.

Dateiname	Beschreibung
i18n.properties	Es ist keine Sprache definiert. Daher wird die Standardsprache vorausgesetzt. Welche Sprache genau als Standard gilt, wird durch den größeren Kontext bestimmt, in dem Ihre Anwendung ausgeführt wird. Wenn z. B. die Standardsprache im Unternehmen Englisch ist, sind die in dieser Datei enthaltenen Texte englischsprachig – dies ist aber ein übernommener Standard und kann nicht aus dem Dateinamen selbst abgeleitet werden.
i18n_en.properties	Ein Ressourcenbündel mit englischem Text.
i18n_en_GB.properties	Ein Ressourcenbündel mit Text in britischem Englisch.
i18n_de.properties	Ein Ressourcenbündel mit deutschem Text.
i18n_de_AT.properties	Ein Ressourcenbündel mit Text in österreichischem Deutsch.

Tabelle 5.1 Namenskonventionen für Ressourcenbündel

Bei der Ermittlung, welche Sprachressourcendatei geöffnet werden soll, wird vom Browser immer die spezifischste Datei zuerst angefordert. Wenn Ihr Browser auf das Gebietsschema UK eingestellt ist, wird *i18_en_GB.properties* angefordert. Wird diese Datei nicht gefunden, wird vom Browser die nächste spezifischste Datei angefordert, die in diesem Fall *i18n_en.properties* wäre. Wird diese Datei nicht gefunden, werden alle Gebietsschemawerte aus dem Dateinamen entfernt und einfach das Ressourcenbündel mit dem Text in der Standardsprache *i18n.properties* angefordert. Dies ist als *Fallback-Prozess zur Ermittlung der Sprache* bekannt.

5.7.3 Codepage

Es ist wichtig zu verstehen, dass in der Spezifikation für Ressourcenbündel-Dateien festgelegt ist, dass alle Texte zur Codepage ISO-8859-1 (LATIN-1) gehören. Dies hat zwei wichtige Konsequenzen:

- Bei der Verwendung der LATIN-1-Codepage wird davon ausgegangen, dass der Text zu einer von links nach rechts verlaufenden Sprache gehört.
- Texte, die zu Sprachen gehören, die den Einsatz einer anderen Codepage erfordern, z. B. fernöstliche Sprachen, müssen mit Unicode-Programmie-

rung statt mit dem nativen Skript der Sprache in die Ressourcenbündel-Datei eingegeben werden.

Wenn Sie z. B. hebräischen Text in eine Ressourcenbündel-Datei eingeben möchten, müssen Sie berücksichtigen, dass Hebräisch zum einen eine Nicht-LATIN-1-Sprache und zum anderen eine von rechts nach links verlaufende Sprache ist. Die hebräische Übersetzung für »Hello World« ist »Shalom Olam«.

Um dieses Problem zu beheben, müssen Sie zuerst eine Ressourcenbündel-Datei mit dem Namen *i18n_he.properties* erstellen. Allerdings ist es nicht korrekt, das hebräische Skript direkt in diese Datei einzugeben. Wenngleich das hebräische Skript absolut korrekt ist, ist das folgende Beispiel inkorrekt:

```
helloWorld=!שלום עולם
```

Stattdessen sollten Sie Folgendes eingeben:

```
helloWorld=\u05E9\u05DC\u05D5\u05DD \u05E2\u05D5\u05DC\u05DD!
```

Zum einen wurden die hebräischen Zeichen in die hexadezimalen Unicode-Werte umgewandelt, und zum anderen musste die Zeichenreihenfolge des hebräischen Satzes umgekehrt werden. Dies ist wichtig, da bei allen Texten in einer *.properties*-Datei davon ausgegangen wird, dass sie im Links-nach-rechts-Format festgelegt sind – selbst wenn der Text zu einer von rechts nach links verlaufenden Sprache gehört!

Im Browser wird die Reihenfolge der Zeichen in diesem String automatisch umgekehrt, damit er korrekt gerendert wird. Weitere Informationen finden Sie im Attribut `dir="RTL"` für UI-Controls.

5.7.4 Verwenden eines Ressourcenmodells

Setzen wir nun unsere Beispielanwendung fort und ersetzen alle Texte, die übersetzbar sein sollten, durch Werte aus einer *.properties*-Datei.

Zuerst müssen Sie das Verzeichnis *i18n* innerhalb Ihres Anwendungsstammordners erstellen. Danach müssen Sie die *.properties*-Datei innerhalb des *i18n*-Verzeichnisses erzeugen. Innerhalb dieses Verzeichnisses können *.properties*-Dateien für jede Sprache abgelegt werden, in die Ihre Anwendung übersetzbar sein soll. Eine typische *.properties*-Datei für unsere Anwendung könnte wie in Listing 5.44 aussehen.

```
notFoundTitle=Not Found
notFoundText=The resource was not found
backToMaster=Back to Master List

masterViewTitle=Supplier Overview
tableNoDataText=No data
masterTableTitle=Supplier
tableIDColumnTitle=ID
tableNameColumnTitle=Name

detailTitle=Supplier Detail
ID=ID
```

Listing 5.44 Standardmäßige .properties-Datei für die Beispielanwendung

Im nächsten Schritt müssen wir das Ressourcenmodell instanziieren. Dies erfolgt in der Komponente, da dieses Modell über die Anwendung hinweg verfügbar sein soll.

Es gibt zwei Bereiche im Component-Code, die wir ändern. Der erste Bereich ist Teil der Metadaten der Komponente, d. h. die Konfiguration. Definieren Sie den Pfad zum Modell, wie in Listing 5.45 dargestellt.

```
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 5.45 Auszug aus den Metadaten der Komponente

Der zweite Bereich ist die Methode `init` der Komponente. Greifen Sie hier auf den Konfigurationswert zu, instanziieren Sie das Ressourcenmodell über den von Ihnen abgerufenen Pfad, und legen Sie es für die Komponente fest, die dieses wiederum an die untergeordneten Elemente (Views) propagiert (siehe Listing 5.46).

```
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 5.46 init-Funktion in der Komponente

Wir können nun die hartcodierten Strings in der Anwendung durch Werte aus dem Modell ersetzen. Da es sich hier um ein benanntes Modell handelt, können wir auf die Werte innerhalb der Bindungen mit einem Präfix wie z. B. `i18n>detailTitle` zugreifen (siehe Listing 5.47).

```
<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 5.47 Auszug aus Detail-View mit Ressourcenmodell

Im Objekt-Header mit dem ID-Wert des Lieferanten sehen Sie, dass es auch möglich ist, Werte aus zwei verschiedenen Modellen, z. B. aus dem Ressourcen- und dem Geschäftsdatenmodell, in einer Bindung zu kombinieren.

Sie sollten die Anwendung jetzt prüfen und die restlichen hartcodierten Strings ersetzen, damit die Anwendung vollständig übersetzbar wird. Auf Wunsch können Sie danach eine weitere *.properties-Datei* zum Ordner *i18n* mit dem Suffix für die gewünschte Zielsprache hinzufügen und die Eigenschaften aus Ihrer Quelldatei übersetzen.

Wenn Sie die Anwendungssprache im Browser ändern, können Sie sehen, wie bestimmte Strings durch die entsprechende Sprache ersetzt werden. Als Beispiel haben wir eine zusätzliche deutsche *.properties*-Datei mit dem Namen *i18n_de.properties* bereitgestellt (siehe Listing 5.48).

```
notFoundTitle=Nicht gefunden
notFoundText=Die Ressource wurde nicht gefunden
```



```
backToMaster=Zurück zur Master-Liste

masterViewTitle=Anbieter-Überblick
tableNoDataText=Keine Daten
masterTableTitle=Anbieter
tableIDColumnTitle=ID
tableNameColumnTitle=Name
tablePriceColumnTitle=Preis

detailTitle=Anbieter-Detail
detailTableHeader=Anbieter-Produkte
ID=ID
detailFilterLabel=Produkte Filtern
```

Listing 5.48 i18n.properties-Datei mit übersetzbaren Texten für die Anwendung

Sie können zwischen den verschiedenen Sprachen wechseln, indem Sie den Sprachparameter für Ihre Anwendung in der Adressleiste des Browsers wie folgt festlegen:

http://yourpathtoyourapplication?sap-ui-language=fr

Im Allgemeinen verwendet das Framework einen Sprachcode des Typs `string`, um eine Sprache zu identifizieren. Hierzu gehören typischerweise Strings wie der BCP-47-Standard, de, en-US, zh-Hans-CN usw. Die Java-Gebietsschemasyntax und die SAP-eigenen Sprachcodes werden aber auch akzeptiert. Weitere Informationen finden Sie im Entwicklerhandbuch.

Die aktuelle Sprache wird aus verschiedenen Quellen ermittelt. In der folgenden Liste finden Sie alle Quellen in umgekehrter Reihenfolge ihrer Priorität. Dies bedeutet, dass der letzte Eintrag in der Liste Vorrang vor den anderen hat:

- Hartcodierte SAPUI5-Standardgebietsschema 'en'.
- Potenziell konfigurierte Browsersprache (`window.navigator.browserLanguage`); für den Internet Explorer ist dies die Sprache des Betriebssystems.
- Potenziell konfigurierte Benutzersprache `window.navigator.userLanguage`; für den Internet Explorer ist dies die Sprache in den Einstellungen Region.
- Allgemeine Sprachinformationen vom Browser `window.navigator.language`.
- Android: In der Zeichenfolge des Benutzeragenten enthaltene Sprache `window.navigator.userAgent`.

- Im Anwendungscode konfiguriertes Gebietsschema
  `jsdoc:symbols/sap.ui.core.Configuration.`
- Über die URL-Parameter konfiguriertes Gebietsschema.

Wenn Sie über die API auf die aktuelle Sprache zugreifen möchten, ist dies über die entsprechende Methode im Konfigurationsobjekt möglich:

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

Wenn Sie direkt aus dem Code auf einen Wert im Ressourcenmodell zugreifen müssen, z. B. wenn Sie einen Wert innerhalb einer Formatierung verwenden müssen, können Sie als Modul `jQuery.sap.resources` nutzen. Dieses Modul enthält eine API, mit der Sie abhängig von der URL und dem Gebietsschema, das Sie an das Modul übergeben, ein Ressourcenbündel abrufen können.

Sie müssen hier allerdings sicherstellen, dass das Modul tatsächlich vorhanden und in Ihrer Anwendung geladen ist. Hierzu können Sie das Modul explizit wie folgt anfordern:

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

Anschließend können Sie mit dem Modul das gewünschte Ressourcenbündel abrufen:

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

Weitere Informationen erhalten Sie im SAPUI5-Demokit auf der Registerkarte API Reference unter `jQuery.sap.resources`. Nachdem Sie das Ressourcenbündel in Ihre Anwendung geladen haben, können Sie die Methode `getText` auf die Gruppe anwenden, um über den Schlüssel auf bestimmten Text zuzugreifen:

```
var sText = oBundle.getText(sKey);
```

5.8 View-Modelle und das Gerätemodell

Die letzten beiden in diesem Kapitel besprochenen Modelle sind eigentlich keine unterschiedlichen Modelltypen, sondern es kann sich dabei um beliebige Modelle handeln. Allerdings verwenden wir wieder JSON-Modelle in der Beispielanwendung.

5.8.1 Verwenden von View-Modellen

Mit einem *View-Modell* können Sie die Status der verschiedenen Elemente in Ihrem View nachverfolgen. Damit können Sie eine Verbindung zwischen Werten, die durch die Anwendungslogik innerhalb des Controllers berechnet werden, und den UI-Elementen herstellen, die z. B. aktiviert/deaktiviert, ein-/ausgeblendet oder aktiv/inaktiv sein müssen.

Immer wenn Sie einen Wert aus Ihrem Geschäftsdatenmodell nicht über die typischen Bindungsmethoden an den View übergeben können, sondern die Werte auf Basis verschiedener Bedingungen berechnen müssen, sollten Sie ein View-Modell verwenden. So können Sie die Controller-Logik und die Deklaration des Views voneinander trennen, da somit verhindert wird, dass der Controller etwas über die bestimmten Controls innerhalb der Views und deren jeweiligen Status erfahren muss.

Wir möchten dies anhand eines Beispiels verdeutlichen. Wir implementieren eine Funktion, mit der wir die Lieferanten innerhalb der Anwendung durchblättern können. Wenn ein Anwender den Detail-View für einen Lieferanten öffnet, wird ihm rechts oben im View ein Button zum Vor- und/oder Zurückblättern angezeigt. Wenn er auf eine dieser Buttons klickt, wird ein Routingereignis ausgelöst, und der Anwender gelangt zum nächsten bzw. vorherigen Lieferanten.

Der Detail-View in der Anwendung sieht nun wie in Listing 5.49 aus.

```
<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>
[...]
```

Listing 5.49 Geänderte Kopfzeile im Detail-View

Wir verwenden hierzu spezielle Symbole für die beiden Buttons, die zum Blättern entwickelt wurden. (Mit `ToolbarSpacer` wird sichergestellt, dass die Buttons rechtsbündig sind.) Die Herausforderung ist es jetzt, diese Buttons abhängig davon zu aktivieren bzw. zu deaktivieren, ob es einen nächsten bzw. vorherigen Lieferanten überhaupt gibt. Wird der erste Lieferant in der Liste angezeigt, ergibt es keinen Sinn, einen aktiven Back-Button bereitzustellen.

Wir könnten fortfahren und die Methoden verwenden, mit denen wir zuvor Informationen über die Position des Lieferanten in der Liste abgerufen haben. Um jedoch die Buttons aus dem Controller dynamisch zu aktivieren und zu deaktivieren, müssten wir den Code in den Controller schreiben, der den Status des Buttons aktualisiert, wenn der Bindungskontext im View auf einen anderen Lieferanten wechselt. Zudem müsste dem Controller genau bekannt sein, welche Buttons im View sind, indem auf diese z. B. über ihre IDs zugegriffen wird. Um dies zu entkoppeln und so den Code möglichst wiederverwendbar und verwaltbar zu gestalten, sollte die Controller-Logik unabhängig von den im View sichtbaren Buttons sein.

Glücklicherweise gibt es eine einfache Lösung für dieses Dilemma: Erstellen Sie ein weiteres Modell im Ereignis `onInit` des Detail-Views. Dieses Modell ist ein normales JSON-Modell, das wir als benanntes Modell im View festlegen. Es enthält nur wenige Daten (in diesem Fall sind nur zwei Werte erforderlich), weshalb wir die Werte des Modells direkt dort bestimmen können, wo das Modell instanziiert ist.

Der Handler `onInit` im Detail-View sieht nun wie in Listing 5.50 aus.

```
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 5.50 Instanziieren eines View-Modells

Die beiden Eigenschaften enthalten nun boolesche Werte, die wir auch zum Binden der aktivierten Eigenschaften aus den Buttons verwenden. Hierzu verwenden wir die normale Property-Binding-Syntax im View. Fügen wir

deshalb eine weitere Eigenschaft zu jedem der beiden Buttons, und zwar die Eigenschaft `enabled` hinzu, und binden wir diese an das View-Modell (siehe Listing 5.51).

```
<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 5.51 Binden an das View-Modell

Wenn Sie die Seite neu laden, sehen Sie, dass nun beide Buttons deaktiviert sind. Egal welchen Lieferanten Sie jetzt auswählen, Sie können keinen anklicken (siehe Abbildung 5.12).

Abbildung 5.12 Lieferantendetails mit Objekt-Header

Es fehlt noch die Logik, mit der das View-Modell aktualisiert wird, wenn das Element `bindingContext` für den View und die Funktionalität für die Navigation geändert wird.

Wir müssen diese Logik so implementieren, dass sie ausgelöst wird, wenn die Funktion `bindElement` im View aufgerufen wird, was in der Methode `onObjectMatched` des Detail-Views geschieht.

Um einen klaren Code zu erhalten, schreiben wir unsere eigene Funktion, mit der wir den korrekten Zustand ermitteln können. Diese Funktion können wir dann aus der Methode `onObjectMatched` aufrufen.

Die neue Funktion muss die aktuelle Position eines Lieferanten im Modell abrufen und anhand dieser Informationen den Status, in dem sich die Buttons befinden sollten, berechnen. Außerdem speichern wir die Informatio-

nen über die Position im View-Modell, da diese im Navigations-Event-Handler wiederverwendet werden.

Wir können die Informationen abrufen und das View-Modell aktualisieren, indem wir die Codezeilen aus Listing 5.52 zum Detail-Controller hinzufügen.

```
/**
 * 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 5.52 Binden des Views an den richtigen Kontext und Berechnen der Button-Aktivierung

Hier versuchen wir, die nächsten und die vorherigen Lieferanten abzurufen, indem wir 1 zur aktuellen ID addieren bzw. diesen Wert davon subtrahieren. Hierbei gehen wir davon aus, dass das ID-Feld ein Feld ist, mit dem wir die Reihenfolge sicher bestimmen können (was in den meisten Fällen nicht funktionieren würde). Normalerweise sollte der Service dazu in der Lage sein, Informationen über das nächste bzw. vorherige Objekt einer Sammlung bereitstellen zu können.

Als Nächstes müssen wir die Codezeile hinzufügen, mit der diese Funktion ausgelöst wird (siehe Listing 5.53).

```
[...]
/**
 * 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 5.53 Aktualisieren des gebundenen Lieferanten und des View-Modells aus der Funktion _onObjectMatched

Beachten Sie, wie wir gleichzeitig die Dokumentation für diese Methode aktualisiert haben. Wenn Sie nun die Seite neu laden, wird der korrekte Button-Status abhängig davon gepflegt, ob wir uns beim ersten oder letzten Lieferanten befinden. Allerdings können wir mit diesen Buttons immer noch nicht navigieren, da die entsprechenden Event-Handler für die `press`-Ereignisse noch fehlen.

Wir fügen die Methode `onPagingButtonPress` sowohl zum XML-View als Event-Handler für die Buttons als auch zum Controller hinzu, in dem die eigentliche Navigation implementiert wird. Da wir innerhalb der Anwendung ein entsprechendes Routing verwenden, können wir jetzt einfach dieses Routing durch Aufruf der entsprechenden Funktion `navTo` mit den Informationen über den nächsten bzw. vorherigen Lieferanten nutzen.

Wie bereits erwähnt, gehen wir in unserem Szenario davon aus, dass wir den Index des Lieferanten um 1 erhöhen bzw. um 1 verringern können, um den nächsten bzw. vorherigen Lieferanten zu erhalten. Daher müssen wir nur die beiden Funktionen in Listing 5.54 schreiben.

```
onPageUp : function(oEvent) {
  var sID = oEvent.getSource().getBindingContext().getPath();
  sID = parseInt(sID.substr(sID.lastIndexOf("/")+1));
  sID = sID-1;
  this.getRouter().navTo("detail", {ID: sID});
},

onPageDown : function(oEvent) {
  var sID = oEvent.getSource().getBindingContext().getPath();
  sID = parseInt(sID.substr(sID.lastIndexOf("/")+1));
```

```
  sID += 1;
  this.getRouter().navTo("detail", {ID: sID});
},
```

Listing 5.54 Navigationsfunktionen zum Blättern durch die Lieferanten über die Pfeil-Buttons

Wir verwenden den Pfad aus dem Ereignisobjekt, das an den Event-Handler übergeben wurde, und extrahieren die aktuelle Lieferantennummer aus diesem Pfad. Wir wandeln diese in eine ganze Zahl um, da das Hinzufügen von +1 eine Zeichenfolgenverkettung auslösen würde. Dann subtrahieren wir entweder 1, wenn ein Anwender auf den Pfeil nach oben klickt, oder addieren 1, wenn der Anwender nach unten navigiert. Wir übergeben das Ergebnis als Parameter `ID` an die Funktion `navTo`, damit die Navigation zum nächsten oder vorherigen Lieferanten ausgelöst wird und diese entsprechend angezeigt werden.

5.8.2 Verwenden des Gerätemodells

Das *Gerätemodell* ist ein normales JSON-Modell, das wir separat behandeln, da es eine besondere Bedeutung innerhalb der Anwendung hat. Dieses Modell wird jedoch nicht für einen bestimmten View implementiert, sondern ganz allgemein in der Anwendung eingesetzt. Damit werden die Control-Eigenschaften nachverfolgt, die abhängig davon angepasst werden müssen, ob die Anwendung auf einem Desktop-PC oder einem Mobilgerät (Touchscreen) angezeigt wird.

Dies ist insbesondere für einige Eigenschaften sinnvoll, die sich auf das Aussehen auswirken, z. B. die aktiven Zustände eines Controls, auf das Sie getippt oder geklickt haben. Da dieses Modell in mehr als einem View verwendet werden kann, implementieren wir dieses separat und instanziieren es innerhalb der Komponente.

Die Implementierung für das Modell selbst erfolgt in einer separaten Datei, die wir im vorhandenen Verzeichnis *model* der Anwendung ablegen und *models.js* nennen.

Der Code für dieses Modell ist in Listing 5.55 dargestellt.

```
sap.ui.define([
 "sap/ui/model/json/JSONModel",
 "sap/ui/Device"
], function (JSONModel, Device) {
  "use strict";
```



```
  return {
createDeviceModel: function () {
var oModel = new JSONModel(Device);
oModel.setDefaultBindingMode("OneWay");
    return oModel;
    }
  };
});
```

Listing 5.55 Implementieren eines Gerätemodells für die Anwendung

Die in den Aufruf `sap.ui.define` übergebenen Parameter sind zwei Klassen, die wir innerhalb dieses Modells benötigen, um alle Informationen über den aktuellen Client aus der API `sap.ui.Device` zu erhalten und diese innerhalb eines JSON-Modells zu speichern.

Wenn dieses Modell aufgerufen wird, wird eine neue Funktion `createDeviceModel` zurückgegeben. Innerhalb dieser Funktion erstellen wir ein neues JSON-Modell aus den Daten, die wir aus der Geräte-API abgerufen haben. Da dieses Modell Informationen in Schlüssel-Wert-Paaren bereitstellt, können wir das Objekt an den JSON-Modell-Konstruktor übergeben.

Anschließend setzen wir `bindingMode` auf `OneWay`, da es nicht sinnvoll ist, die Werte innerhalb des Modells aus dem Anwendungscode zu manipulieren. Schließlich geben wir das neu erstellte Modell zurück. Wenn wir nun zur Anwendungskomponente zurückgehen, müssen wir das neue Modell nur in die Komponente einfügen.

Wir müssen die entsprechende Ressource zum Abschnitt `define` der Komponente hinzufügen (siehe Listing 5.56).

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

Listing 5.56 Hinzufügen von Ressourcen für das Gerätemodell

Innerhalb der Funktion `init` der Komponente fügen Sie den Code für die zusätzliche Modellinstanziierung, wie in Listing 5.57 dargestellt, ein.

```
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 5.57 In der Funktion init instanziiertes Gerätemodell

Nun können wir über unsere Views auf das Gerätemodell zugreifen. Mithilfe der Informationen darüber, welches Gerät auf die Anwendung zugreift, können wir bestimmen, ob wir Controls anzeigen oder ausblenden, deren Verhalten ändern oder Informationen in einem anderen Control anzeigen möchten. Wir könnten z. B. die Blätter-Buttons anzeigen lassen, die wir nur für Smartphone-Bildschirme für die Lieferanten erstellt haben.

In diesem Fall mussten wir nur die Zeilen im Detail-View ändern, die die entsprechenden Text-Controls, wie in Listing 5.58 dargestellt, instanziieren.

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

Listing 5.58 Verwenden von Eigenschaften aus dem Gerätemodell für Bindungen

Die Anwendung sieht nun im Desktop-Modus ein wenig anders aus, bei dem die Pfeil-Buttons nicht mehr im Detail-View angezeigt werden.

Wir können diese aber wieder aktivieren, indem wir in Google Chrome in den Emulationsmodus wechseln. So können wir ermitteln, wie die Anwendung auf einem Smartphone aussieht.

5.9 Fazit

In diesem Kapitel haben Sie die clientseitigen Modelle und deren Einsatz in SAPUI5 kennengelernt. Sie haben erfahren, wie Modelle instanziiert werden, und Sie haben die typischen Data-Binding-Methoden in der Beispielanwendung verwendet.

Mit diesem Wissen haben wir die Anwendung soweit erweitert, dass die gängigsten Funktionen in den Anwendungen der Enterprise-Klasse angesprochen wurden. Wir haben die Anwendung mithilfe eines Ressourcenmodells multilingual gestaltet und eine plattformübergreifende Kompatibilität erzielt, indem wir ein Gerätemodell für jene Funktionen umgesetzt haben, die auf Desktop-PCs und typischen Mobilgeräten unterschiedlich gehandhabt werden müssen.

In Kapitel 6, »CRUD-Operationen«, fahren wir mit der Erweiterung der Anwendung fort, indem wir Mittel zur Manipulation der Daten in einem Modell aus der Anwendung heraus hinzufügen. Dieses Mal verwenden wir aber einen echten Service anstelle von lokalen Beispieldaten.

Einleitung

SAPUI5 ist das neue Toolkit von SAP zur Programmierung von Benutzeroberflächen als wichtiger Bestandteil in der aktuellen Technologiestrategie von SAP. Insbesondere handelt es sich dabei um die Technologie, die zur Entwicklung von SAP-Fiori-Anwendungen genutzt wird, die den neuen Standard für das moderne Aussehen von SAP-Anwendungen verkörpern.

Mit SAPUI5 erstellte Anwendungen sind mit verschiedensten Browsern und Geräten einsetzbar und bieten eine integrierte Erweiterbarkeit, funktionsreiche UI-Controls für komplexe Muster und Layouts, die Übersetzung in mehrere Sprachen und Barrierefreiheitsfunktionen. Die Steuerung all dieser Funktionen kann wegen ihrer Komplexität einschüchternd wirken. Und genau hier kommt dieses Buch ins Spiel! Mit diesem umfassenden Handbuch können Sie Ihre Codefähigkeiten verfeinern und die SAPUI5-Sprache sicher erlernen. Egal ob Sie mit den Grundbausteinen beginnen oder erweiterte Funktionen in Ihrem Code umsetzen möchten – dieses Buch begleitet Sie auf Ihrer gesamten SAPUI5-Reise.

An wen richtet sich das Buch?

Dieses Buch wurde für alle Entwickler konzipiert und geschrieben, die bereits über grundlegende Kenntnisse in der Webentwicklung und in JavaScript verfügen und ihr Wissen über den Aufbau von Anwendungen mit SAPUI5 vertiefen möchten. Entwickler mit einem anderen Hintergrund sollten sich von dieser Aussage aber nicht abschrecken lassen! Wenn Sie mit diesem Buch arbeiten, empfehlen wir Ihnen, dass Sie entweder parallel ein gutes JavaScript-Buch verwenden oder vor der Buchlektüre eine der zahlreichen Onlinequellen zu den JavaScript-Grundlagen zurate ziehen. Für alle ABAP-Entwickler nochmals der Hinweis: Denken Sie daran, dass in JavaScript die Groß-/Kleinschreibung berücksichtigt wird.

Ziel dieses Buches

Ziel dieses Buches ist es, Ihnen das nötige Wissen zu vermitteln, um voll einsatzfähige SAPUI5-Anwendungen entwickeln zu können. Am Ende der Lektüre sollten Sie umfassende Kenntnisse über die hierzu notwendigen Kon-

zepte sowie ein praktisches Verständnis dafür entwickelt haben, wie diese Konzepte beim Aufbau echter Anwendungen angewendet werden.

Wie das Buch zu lesen ist

Das Buch und die einzelnen Kapitel bauen aufeinander auf. Zu Beginn werden die Grundlagen für den Aufbau von SAPUI5-Anwendungen erläutert, und im weiteren Buchverlauf werden nach und nach komplexere Konzepte vorgestellt.

Teil I, »Einführung«, bietet eine Einführung in SAPUI5 und dessen Architektur. Teil I enthält die folgenden Kapitel:

- In **Kapitel 1**, »SAPUI5 auf einen Blick«, geben wir eine Einführung in SAPUI5 im Allgemeinen. Hierzu gehören Informationen über die Geschichte des Produkts, seine Entwicklung, Funktionen, Anwendungsfälle, Konkurrenzprodukte und die Open-Source-Variante OpenUI5.
- In **Kapitel 2**, »Architektur«, gehen wir näher auf die Architektur von SAPUI5 sowie auf die Bibliotheken, Elemente und die Strukturhierarchie ein.

In **Teil II**, »SAPUI5 in Aktion – Entwicklung von Anwendungen«, widmen wir uns den einzelnen Schritten für den Aufbau von Anwendungen unter der Verwendung von SAPUI5 – von »Hello, World« bis hin zu erweiterten Konzepten wie dem Schreiben von eigenen Controls. Teil II beinhaltet die folgenden Kapitel:

- **Kapitel 3**, »Hello, SAPUI5 World«, startet mit einem einfachen »Hello, World«-Anwendungsbeispiel. Hier werden Sie durch die ersten und wichtigsten Schritte bei der Entwicklung von Anwendungen mit SAPUI5 geführt. In diesem Kapitel betrachten wir die Struktur einer einseitigen Anwendung sowie die wichtigsten Bausteine von SAPUI5: die Controls, deren Funktionalität und deren API.
- **Kapitel 4**, »Aufbau von MVC-Anwendungen«, baut auf den in Kapitel 3 vorgestellten Konzepten auf und geht näher auf Anwendungsmodelle, Views und Controller ein.
- In **Kapitel 5**, »Modelle und Bindings«, erläutern wir die allgemeine Nutzung und die Modelltypen in SAPUI5. Wir erörtern auch die unterschiedlichen Arten des Data Bindings, die in SAPUI5 implementiert werden können. Hierzu gehören Property, Element, Aggregation und Expression Binding.

- In **Kapitel 6**, »CRUD-Operationen«, beginnen wir, unsere Anwendungen mit der »echten Welt«, d. h. mit echten Services zu verknüpfen. Außerdem erfahren Sie, wie typische Anfragen an das Backend erstellt werden. Darüber hinaus erläutern wir wichtige Funktionen wie das Filtern, Sortieren, Gruppieren und Erweitern.
- In **Kapitel 7**, »Einsatz von OData«, stellen wir eines der wichtigsten REST-basierten Protokolle vor: OData. Wir zeigen Ihnen, wie Sie eine Anwendung mit einem OData-Modell verknüpfen können, und gehen auf die weiteren Funktionen von OData ein.
- In **Kapitel 8**, »Anwendungsmuster und -beispiele«, erkunden wir die am häufigsten verwendeten Anwendungsmuster in SAPUI5. Wir betrachten auch die Bereitstellung unserer Anwendungen im Portal SAP Fiori Launchpad, das für SAP-Fiori- und SAPUI5-Anwendungen verwendet wird.
- In **Kapitel 9**, »Weiterführende Konzepte«, zeigen wir Ihnen, wie Sie Ihre Anwendungen noch weiter verbessern können! Sie erfahren, wie Sie OData-Annotations zur Anreicherung der intelligenten Controls einsetzen oder die SAPUI5-Funktionalität durch das Schreiben von eigenen Controls erweitern können.

Teil III, »Der letzte Schliff«, gibt unserer Anwendungsentwicklung den letzten Schliff. Hier erhalten Sie Informationen dazu, wie Sie SAPUI5-Anwendungen definieren, debuggen und bereitstellen können. Teil III beschäftig sich mit den folgenden Kapiteln:

- In **Kapitel 10**, »Enterprisetaugliche Anwendungen erzeugen«, stellen wir Ihnen alle wichtigen Enterprise-Funktionen vor, die Sie zur Erweiterung Ihrer Anwendungen einsetzen können. Hierzu gehören der Einsatz von Unternehmensprofilen sowie die Implementierung von Sicherheitsmaßnahmen und Leistungsoptimierungen. Außerdem erhalten Sie Richtlinien, damit Sie Ihre Anwendungen Menschen mit Behinderungen zugänglich machen können.
- In **Kapitel 11**, »Debuggen und Testen von Code«, werden die Schritte erläutert, mit denen Sie die Einsatzbereitschaft Ihrer Anwendungen sicherstellen können. Hierzu gehört das manuelle Debuggen und das Schreiben automatischer Tests, die Stabilität und fehlerfreien Anwendungscode gewährleisten.

- In **Kapitel 12**, »Was Sie nicht tun sollten«, zeigen wir Ihnen, wie Sie Ihre Anwendungen ordentlich durcheinanderbringen können. In diesem Kapitel finden Sie die »Worst Practices« und Möglichkeiten, wie Sie Ihre Anwendungen ruinieren können.

Das Buch schließt mit einigen hilfreichen Anhängen zur Abrundung des SAPUI5-Themas ab:

- In **Anhang A**, »IDE-Einrichtung«, erhalten Sie die grundlegenden Konfigurationsrichtlinien für den Einsatz gängiger IDEs innerhalb von SAP einschließlich der SAP Web IDE und WebStorm.
- In **Anhang B**, »Zugriff auf das Backend«, geben wir Ihnen Tipps und Tricks für den Zugriff auf das Backend, einschließlich des Zugriffs auf die SAP HANA Cloud Platform (SAP HCP) und Sicherheits-Plug-ins für Google Chrome.
- In **Anhang C**, »Deployment von Anwendungen«, erläutern wir die Schritte für die Bereitstellung Ihrer Anwendungen in unterschiedlichen Plattformen wie z. B. der SAP HCP, der SAP Web IDE und dem ABAP-Server.
- In **Anhang D**, »Cheat Sheets«, erhalten Sie Kurzanleitungen für wichtige Codekonzepte, z. B. zum Starten einer Anwendung, zum Durchführen von Data Bindings usw.
- In **Anhang E**, »Weitere Ressourcen«, finden Sie eine umfangreiche Liste mit Ressourcen zum Weiterlernen. Diese reichen von Links zu openSAP-Kursen und Dokumentationen bis hin zu Webseiten und GitHub-Repositorys.

Begleitmedien

Im gesamten Buch verwenden wir Beispiele, Schritt-für-Schritt-Anleitungen und Beispielcode, um die relevanten Konzepte praktisch darzustellen. Den Beispielcode und andere zusätzliche Inhalte zu diesem Buch finden Sie auf der Webseite des Buches unter *https://www.sap-press.de/4303/*.

Wir hoffen, dass Sie beim Lesen und Umsetzen dieses Buches so viel Spaß haben, wie wir es beim Schreiben hatten. Lassen Sie uns anfangen!

Danksagung

Unser besonderer Dank geht an unsere Kollegen des zentralen SAPUI5-Teams in Walldorf und in Sofia, die fortlaufend an der Verbesserung dieses Frameworks arbeiten. Ohne deren Beiträge wäre dieses Buch nicht möglich gewesen.

Wir danken Chris Whealy für seine Hilfe und Unterstützung – beim Schreiben von Büchern im Allgemeinen und für sein Lektorat und sein nützliches Feedback!

Auch die Autoren möchten ihre persönlichen Danksagungen aussprechen:

Danke an meinen Partner, meine Familie und meine Freunde, die viel Geduld mit mir hatten und sich nicht beschwert haben, als ich zum 100. Mal eine Einladung mit den Worten »Ich muss noch Kapitel XY abschließen.« ausgeschlagen habe.

– Christiane Goebels

Ich möchte meinen Freunden, Kollegen und meiner Familie danken, die mich auf dieser Reise und beim Schreiben dieses Buches motiviert und ermutigt haben. Ein großes Dankeschön geht an meine großartigen Mitautoren Christiane und Thilo. Es war mir eine Freude, mit Euch zusammenzuarbeiten! Ein weiteres großes Dankeschön geht an meinen Vorgesetzten Christian Paulus, der dieses Buchprojekt von Anfang an unterstützt hat und viel Verständnis hatte, wenn ich kurzfristig Urlaub nehmen musste, wenn ein weiterer Abgabetermin anstand.

Ein besonderer Dank geht an unsere Lektorin Sarah Frazier, die großartige Arbeit geleistet hat und deren ermutigenden Worte mir beim Abschluss dieses Projekts geholfen haben.

– Denise Nepraunig

Ich möchte all den großartigen Menschen danken, die SAPUI5 entwickeln, unterstützen, nutzen und fördern. Insbesondere danke ich den internen SAPUI5-Entwicklungsteams und den Kollegen, die es nie leid waren, all mei-

ne Fragen in den letzten Jahren und insbesondere bei der Entstehung dieses Buches zu beantworten.

Außerdem möchte ich den Mitarbeitern von OpenUI5 danken, die wichtige Teile von SAPUI5 als Open Source freigegeben haben. Ihr seid großartig!

Persönlich möchte ich Sonia dafür danken, dass sie stolz auf mich ist; Mattis danke ich, dass es ihn gibt; Denise und Christiane danke ich, dass ich bei diesem Buch Mitautor sein durfte!

– Thilo Seidel

Auf einen Blick

Inhalt

TEIL II SAPUI5 in Aktion – Entwicklung von Anwendungen

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

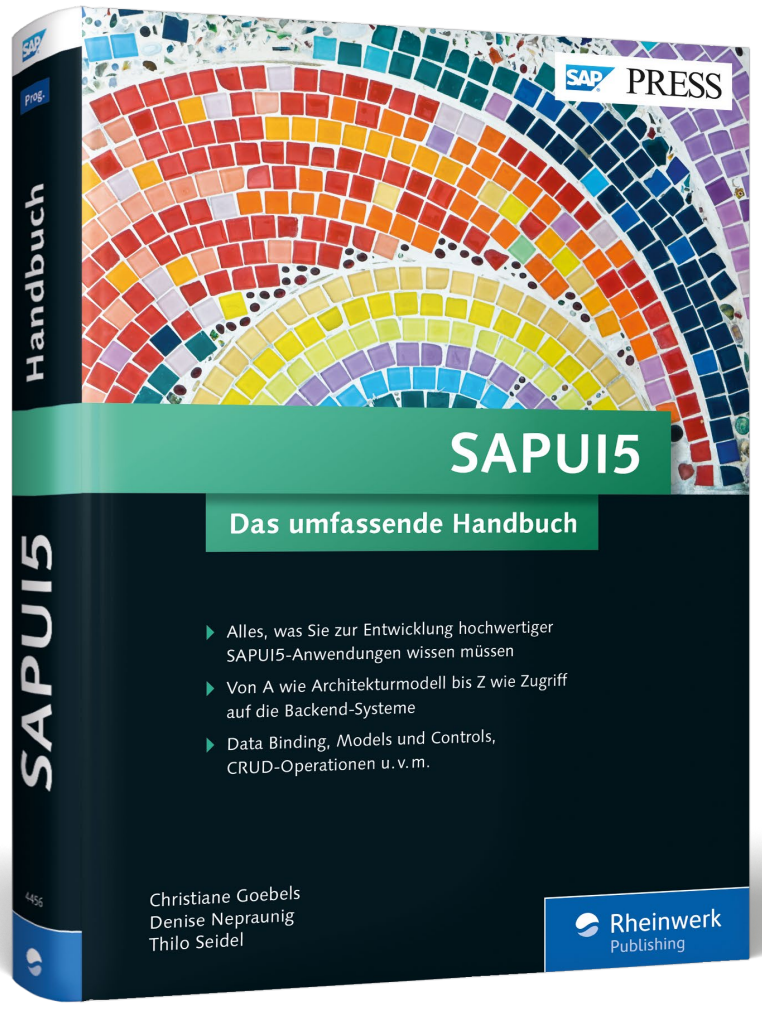
U

V

W

X

Z



Christiane Goebels, Denise Nepraunig, Thilo Seidel

SAPUI5 – Das umfassende Handbuch

699 Seiten, gebunden, November 2016
79,90 Euro, ISBN 978-3-8362-4456-5

www.sap-press.de/4303

Christiane Goebels ist Senior-Software-Entwicklerin bei SAP. Sie hatte ihr erstes Webentwicklungsprojekt 2001. Seitdem hat sie an Dutzenden großen und kleinen Webanwendungen für SAP, SAP-Kunden und – in ihrem eigenen Internet-Startup von 2005 bis 2010 – für Kunden ohne Bezug zu SAP gearbeitet. Sie ist eine erfahrene Dozentin und Trainerin für SAPUI5 und OpenUI5 sowie andere webbezogene Themen.

Denise Nepraunig ist Software-Entwicklerin bei SAP in Walldorf, wo sie SAPUI5-Apps entwickelt und an der Entwicklung von SAP Web IDE beteiligt war. Denise ist eine erfahrene Sprecherin und SAPUI5-Coach und seit kurzem SAP-Mentorin. Sie liebt es, neue Technologien zu erforschen und experimentiert in ihrer Freizeit mit SAP HCP und SAP HANA.

Thilo Seidel baute seine erste Webseite 2002 und verliebte sich direkt. Seitdem hatte er verschiedene Aufgaben, unter anderem im Vertrieb, als Designer, Denkender, Reisender, Student und Projektmanager. Thilo ist an Werktagen der Product Owner des SAP Fiori Launchpad, am Wochenende ist er manchmal als Hacker unterwegs.