Kapitel 6

Benutzeroberflächen und Dialoge gestalten

Ein wohl zentraler Teil der Entwicklung mit SAPUI5 stellt die Implementierung von Anwendungsoberflächen dar. Visuelle Aspekte in SAPUI5 können sehr unterschiedlich sein.

In den vorangegangenen Kapiteln haben Sie einen ersten Überblick über SAPUI5 erlangt. In diesem Kapitel wollen wir nun in die Tiefe der SAPUI5-Programmierung abtauchen und beginnen dabei mit der Gestaltung von Benutzeroberflächen. Wir geben Ihnen in Abschnitt 6.1 zunächst eine Einführung in die Programmierung von einfachen Benutzeroberflächen. Wir zeigen Ihnen, wie Sie SAPUI5-Controls nutzen können, um Benutzeroberflächen zu strukturieren, und besprechen in Abschnitt 6.2, wie Sie komplexe Anwendungsoberflächen erstellen. In Abschnitt 6.3 zeigen wir Ihnen, was Fragmente sind und welche Möglichkeiten sie für die Strukturierung von Anwendungsoberflächen bieten. Den Abschluss des Kapitels bildet Abschnitt 6.4 mit einer Einführung in die Implementierung von Dialogen. Wir zeigen Ihnen, wie Sie eigene Dialoge erstellen und wie Sie vorgefertigte Dialogklassen in Ihren Anwendungen nutzen können.

Wir verwenden in diesem Kapitel das in Kapitel 4 und Kapitel 5 eingeführte Projekt und erweitern es um die entsprechenden Artefakte.

6.1 Gestaltung von einfachen Benutzeroberflächen

Zur Gestaltung von Benutzeroberflächen gibt es unterschiedliche Möglichkeiten, und je nach Anwendung und Anwendungsfall bietet SAPUI5 einen reichhaltigen Schatz an UI-Controls. Wir widmen uns in diesem Abschnitt zunächst der Verwendung von einfacheren Benutzeroberflächen.

6.1.1 Verwendung von einfachen Controls

Wie Sie bereits wissen, haben einfache Controls in SAPUI5 keine Abhängigkeiten zu anderen Controls oder Elementen. Das bedeutet, sie können ein-

fach verwendet werden. Öffnen Sie zunächst die View **App.view.xml**. Fügen Sie in den `content`-Bereich der `content`-Aggregation des `Page`-Controls ein Control vom Typ `sap.m.Label`, eines vom Typ `sap.m.Input` und eines vom Typ `sap.m.Button` ein (siehe Listing 6.1).

```
<Page id="page" title="Demo-Anwendung">
  <content>
    <Label text="Sag Hallo zu"/>
    <Input/>
    <Button text="Sag Hallo" press="onPress"/>
  </content>
</Page>
```

Listing 6.1 Einfache Controls in einer SAPUI5-Anwendung

Wie Sie sehen, geben wir den Controls einige Attribute mit. Die drei Controls können einfach verwendet werden, da unsere Anwendung in der Datei **manifest.json** eine Abhängigkeit zur Bibliothek `sap.m` definiert hat. Diese Abhängigkeit führt dazu, dass die Bibliothek geladen wird und wir lediglich einen XML-Namensraum in **App.view.xml** definieren müssen, der auf `sap.m` verweist.

Schauen Sie sich das Ergebnis im Browser an, nachdem Sie die Datei gespeichert haben. Die drei Controls werden relativ einfach in die Seite eingebettet (siehe Abbildung 6.1). Wie Sie sehen, spannt sich das `Input`-Control über die gesamte Breite der Seite auf. Wie Sie mit Layouts die Darstellung beeinflussen können, erfahren Sie in Abschnitt 6.1.3.

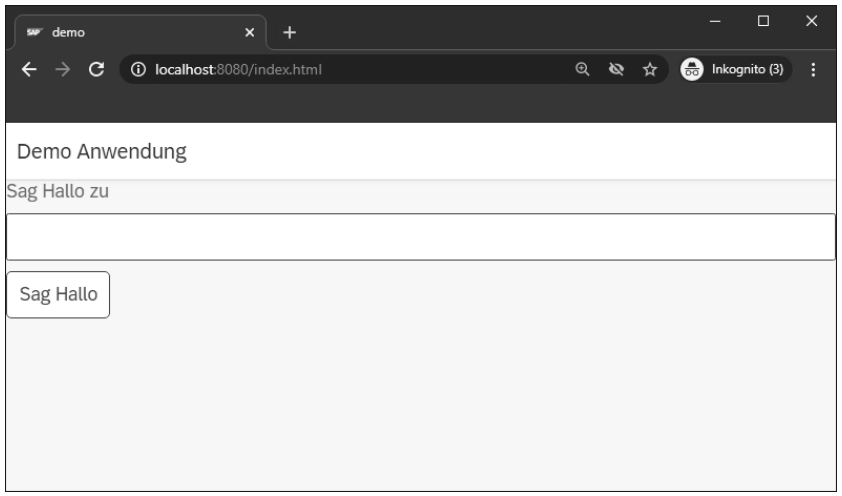

Abbildung 6.1 Einfache SAPUI5-Controls in Aktion

6.1.2 Zugriff auf UI-Controls

Arbeiten mit IDs

SAPUI5 erzeugt beim Rendern der UI-Controls und UI-Elemente eine generische ID für jedes UI-Control bzw. UI-Element, das keine feste ID als Attribut definiert. In unserem Beispiel hat das Eingabefeld beispielsweise die ID `__input0` erhalten (siehe Abbildung 6.2). Diese generischen IDs können theoretisch für den Zugriff auf die entsprechenden Controls verwendet werden. Allerdings ist die ID-Vergabe nicht Teil der öffentlichen API von SAPUI5, was bei Erscheinen einer neuen Version zu Problemen führen kann. Um dies zu vermeiden, legen Sie die ID über das Attribut `id` selbst fest.

Abbildung 6.2 Der HTML-DOM-Baum zur Laufzeit

Legen Sie bei dem Eingabefeld des Typs `sap.m.Input` aus Listing 6.1 das Feld `id` an, und setzen Sie den Wert auf `idInputField`:

```
<Input id="idInputField"/>
```

Schauen Sie sich nach der Änderung den DOM-Baum an. Das Eingabefeld hat nun die ID `container-demo---idAppControl--idInputField` (siehe Abbildung 6.3). Wie kommt diese ID zustande, da wir dem Attribut `id` des Eingabefeldes doch lediglich den Wert `idInputField` zugewiesen haben? Die Vergabe der ID erfolgt bei deklarativen View-Typen wie XML- und HTML-Views auf Basis der Container. In unserem Beispiel wurde dem Komponentencontainer in **index.html** die ID `container` zugewiesen und konfiguriert, dass alle Kind-Controls, die innerhalb des Containers angelegt werden, das Präfix `demo` erhalten sollen:

```
<div data-sap-ui-component data-name="de.sapui5buch.demo" data-id=
  "container"
    data-settings='{"id" : "demo"}'>
```

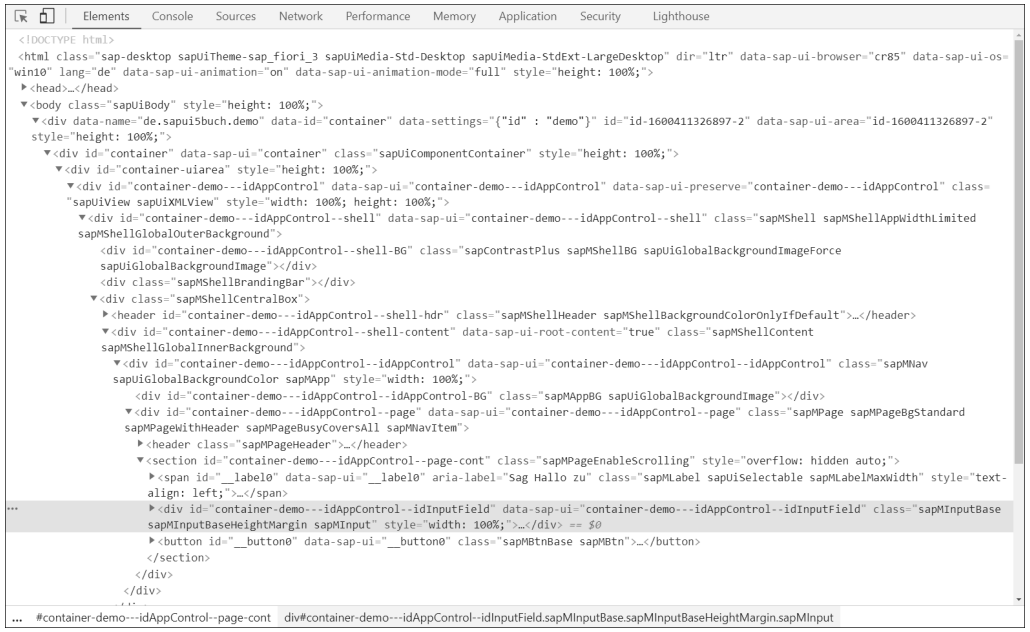

Abbildung 6.3 Statische ID-Vergabe im HTML-DOM-Baum

Stabile ID-Vergabe

Der zweite Teil der ID `idAppControl` kommt von der ID, die wir dem zentralen `sap.m.App`-Control hinzufügt haben (siehe Listing 5.3). Um die stabile ID-Vergabe zu testen, wollen wir den Wert, den der Nutzer in das Eingabefeld eingegeben hat, auslesen und in einem einfachen Nachrichtendialog anzeigen. Praktischerweise bietet SAPUI5 für diesen Anwendungsfall das Control `sap.m.MessageBox` an. Um das Control zu verwenden, müssen Sie zunächst eine Referenz für das Control anfordern. Fügen Sie dazu die Referenz in den Controller der App-View ein (siehe Listing 6.2).

```
sap.ui.define([
  "sap/ui/core/mvc/Controller",
  "sap/m/MessageBox"
], function(Controller,MessageBox) {
  "use strict";
  return Controller.extend("de.sapui5buch.demo.controller.App", {

  });
});
```

Listing 6.2 Referenz auf das Control »sap.m.MessageBox«

»onPress«

Als Nächstes implementieren Sie die Funktion `onPress` im Controller der View (siehe Listing 6.3). Die Funktion erhält als Aufrufparameter ein Event-

objekt. Über dieses Objekt haben Sie Zugriff auf alle relevanten Informationen, die Sie im Rahmen der Eventverarbeitung durch das SAPUI5-Framework erhalten.

```
return Controller.extend("de.sapui5buch.demo.controller.App", {
    onPress : function(oEvent) {
    }
});
```

Listing 6.3 Implementierung des Eventhandlers »onPress«

Prägen Sie nun die `onPress`-Funktion aus. Dazu besorgen Sie sich über die Funktion `byId` eine Referenz auf das Eingabefeld (siehe Listing 6.4). Sie kann entweder direkt auf dem Controller oder auf der Referenz der View aufgerufen werden. Im Anschluss können Sie die Funktion `getValue` aufrufen und erhalten als Rückgabewert die Benutzereingabe. Nachdem Sie den eingegeben Wert in einer lokalen Variablen gespeichert haben, rufen Sie auf dem Objekt `MessageBox` die Funktion `show` auf. Die `show`-Funktion parametrisieren Sie mit zwei Argumenten: zum einen mit dem Wert, den Sie im Dialog anzeigen wollen, und zum anderen mit einem literalen JavaScript-Objekt, mit dem Sie das Control konfigurieren.

```
onPress : function(oEvent) {
      var oInputField = this.byId("idInputField");
// var oInputField = this.getView().byId("idInputField");
      var sValue = oInputField.getValue();
      MessageBox.show(
        "Hallo " + sValue, {
          icon: MessageBox.Icon.INFORMATION,
          title: "Hallo Nachricht",
          }
      );
    }
```

Listing 6.4 Implementierung der Funktion »onPress«

Nachdem Sie die Änderungen gespeichert haben, können Sie die Implementierung testen. Geben Sie einen Wert in das Eingabefeld ein, und klicken Sie auf die Schaltfläche. Es erscheint eine `MessageBox`, die den über JavaScript erzeugten Dialog zeigt (siehe Abbildung 6.4).

Das »width«-Attribut

Wir wollen jedoch nicht, dass das Eingabefeld die gesamte Breite der Seite einnimmt. Um die Breite zu begrenzen, legen Sie im `Input`-Control über das Attribut `width` eine Breite fest. Setzen Sie den Wert auf `20rem`:

```
<Input width="20rem"/>
```

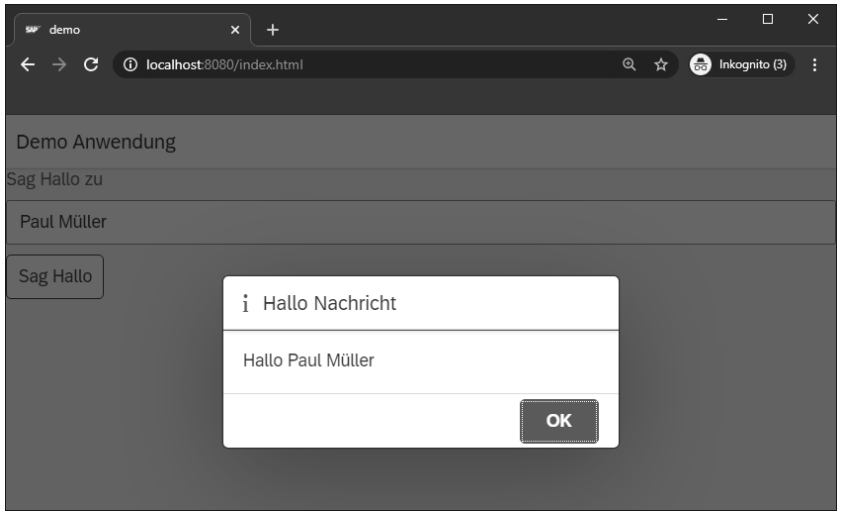

Abbildung 6.4 Die »MessageBox« in Aktion

Wir verwenden die Maßeinheit `rem`, um die Gesamtbreite in Relation zur Breite des Wurzelcontainers des `Input`-Controls zu setzen. Diese flexible Breitenangabe sorgt dafür, dass das `Input`-Feld auch bei einer Änderung der Fensterbreite sauber dargestellt wird.

Nachdem Sie gespeichert haben, sehen Sie nun zwar ein kleineres Eingabefeld, allerdings ist die Darstellung immer noch relativ unschön (siehe Abbildung 6.5). Dies wollen wir nun korrigieren.

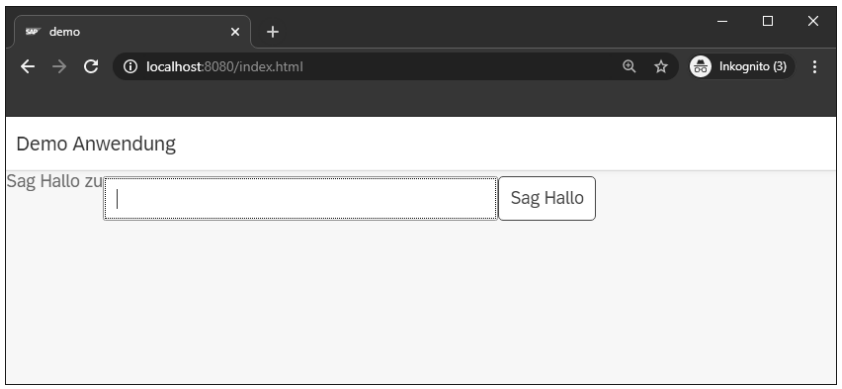

Abbildung 6.5 Verwendung des »width«-Attributs

6.1.3 Einführung in die Verwendung einfacher Layouts

Wir haben im vorangegangenen Abschnitt das Hinzufügen von einfachen Controls in SAPUI5 betrachtet. Das Hinzufügen von neuen Controls ist einfach, allerdings besteht die Herausforderung, Benutzeroberflächenaspekte optisch ansprechend auf der Benutzeroberfläche zu positionieren und zu strukturieren. SAPUI5 stellt einige Layout-Controls bereit, die wir in Kapitel 7, »Arbeiten mit Layouts«, näher beleuchten. An dieser Stelle wollen wir nur die bis jetzt implementierte Benutzeroberfläche etwas schöner machen. Die meisten Layout-Controls sind im Namensraum `sap.ui.layout` implementiert. Aus diesem Grund fügen wir in unsere Implementierung zunächst einen XML-Alias `layout` ein, das auf den Namensraum `sap.ui.layout` verweist (siehe Listing 6.5).

```
<mvc:View controllerName="de.sapui5buch.demo.controller.App"
  displayBlock="true"
  xmlns="sap.m" xmlns:layout="sap.ui.layout"
  xmlns:mvc="sap.ui.core.mvc">
```

Listing 6.5 Der Namensraum »sap.ui.layout« in einer XML-View

Fügen Sie nun das SAPUI5-Control `sap.ui.layout.VerticalLayout` ein, so dass die drei Controls in dem Layout-Control gekapselt sind (siehe Listing 6.6).

```
<layout:VerticalLayout>
   <Label text="Sag Hallo zu"/>
   <Input id="idInputField" width="20rem"/>
   <Button text="Sag Hallo" press="onPress"/>
</layout:VerticalLayout>
```

Listing 6.6 Einfügen des Controls »sap.ui.layout.VerticalLayout«

Im Anschluss speichern Sie die Änderungen und betrachten das Ergebnis (siehe Abbildung 6.6). Die Controls werden nun sauber positioniert.

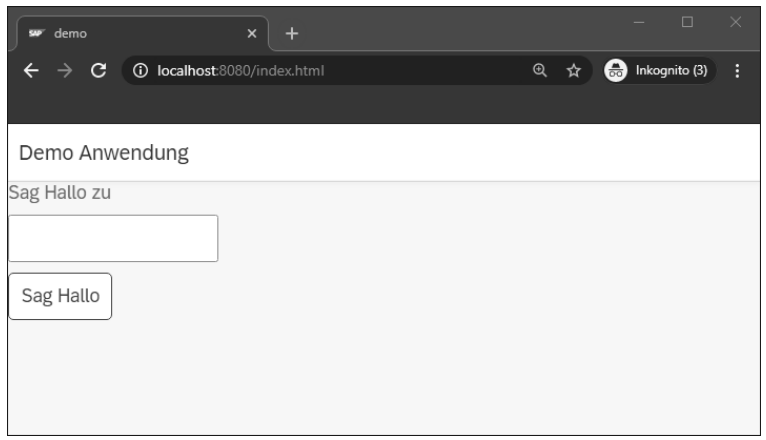

Abbildung 6.6 Das Layout-Control als Klammer

6.2 Gestaltung von komplexen Benutzeroberflächen

Nachdem Sie im vorangegangenen Abschnitt ein paar einfache Controls kennengelernt haben, wollen wir uns in diesem Abschnitt komplexeren Controls zuwenden und auch betrachten, wie ein einfaches Control einen gewissen Funktionsumfang erhält.

6.2.1 Implementieren einer Wertehilfe

»sap.m.Input«

Das Eingabefeld aus dem vorangegangenen Abschnitt ist zwar ganz nett anzusehen, sein Funktionsumfang ist jedoch eher gering. Das Feld `sap.m.Input` ist jedoch recht mächtig. Um das zu demonstrieren, machen wir aus dem Eingabefeld eine Wertehilfe unter Verwendung einer Vorschlagsliste. Diese Vorschlagliste zeigt, so wie Sie es von modernen Suchhilfen gewohnt sind, auf Basis der Eingabe gefilterte Daten an. Aktivieren Sie zunächst die grundsätzliche Funktionalität der Vorschlagsliste, und setzen Sie den Parameter `showSuggestion` auf `true`. Damit das Control nach der Eingabe auch Daten in einer Vorschlagliste anzeigen kann, geben Sie ihm über den Parameter `suggestionItems` an, welches Datenset aus dem Modell die anzuzeigenden Daten enthält. In unserem Beispiel übergeben wir hierzu den Pfad **/d/results** im Rahmen des Aggregation-Bindings. Zum Abschluss teilen wir dem Attribut `suggest` den Namen der Funktion mit, die aufgerufen wird, sobald der Nutzer beginnt, Werte in das Feld einzugeben (siehe Listing 6.7).

```
<Input id="idInputField" width="20rem"
    placeholder="BusinessPartner ID..."
    showSuggestion="true" suggest="onSuggest"
    suggestionItems="{/d/results}">
    <suggestionItems>
      <core:Item text="{BusinessPartnerID} {CompanyName}" />
    </suggestionItems>
</Input>
```

Listing 6.7 »sap.m.Input«-Control als Wertehilfe

Nachdem Sie das Eingabefeld entsprechend konfiguriert haben, wenden wir uns dem Eventhandler für das `suggest`-Event zu. Das `suggest`-Event wird nach jedem eingegebenen Buchstaben ausgelöst. Die Aufgabe des Eventhandlers ist es, die Daten unter Zuhilfenahme der bis dahin eingegebenen Zeichen zu filtern. SAPUI5 bietet für das Filtern eine entsprechende Filterklasse `sap.ui.model.Filter` an. Die Klasse verwendet für die Filterung sogenannte *Filteroperatoren*. Diese werden in der Klasse `sap.ui.model.FilterOperator` als Enumeration zur Verfügung gestellt. Um die beiden Klassen zu

verwenden, fügen wir zunächst die Referenzen in den Controller ein (siehe Listing 6.8).

```
sap.ui.define([
  "sap/ui/core/mvc/Controller",
  "sap/m/MessageBox",
  "sap/ui/model/Filter",
  "sap/ui/model/FilterOperator"
], function(Controller, MessageBox, Filter, FilterOperator) {
```

Listing 6.8 Erweiterung der Abhängigkeiten des Controllers

Implementierung der Funktion »onSuggest«

Implementieren Sie danach die Funktion `onSuggest` (siehe Listing 6.9). Diese Funktion liest den Parameter `suggestValue` aus dem Eventobjekt aus. Dieser Parameter enthält den Wert, den der Anwender in das Eingabefeld eingegeben hat. Auf Basis dieses Wertes wird ein Filterobjekt erzeugt, das den eingegebenen Wert mit den zur Auswahl stehenden Werten abgleicht und diese auf zur Eingabe passende Werte einschränkt. Dem Anwender werden so nur die Werte in der Wertehilfe angeboten, die zu seiner Eingabe passen. Falls ein Wert eingegeben wurde, erzeugt man ein `Filter`-Objekt und parametrisiert dieses.

In unserem Beispiel prüfen wir die Eingabe gegen die Daten im Attribut `BusinessPartnerID` der Entität `BusinessPartner`. Wir gleichen daher ab, ob es einen Wert für diese Eigenschaft gibt, der mit der eingegebenen Zeichenkette beginnt. Das so erzeugte `Filter`-Objekt fügen wir in ein Array ein. Den Filter wenden wir auf die in der Aggregation `suggestionItems` befindlichen Datensätze zu den Business-Partnern an. Hierzu beziehen wir die entsprechende Binding-Referenz und rufen auf dem Ergebnis (einem Array aus Bindings) die Funktion `filter` auf (siehe Listing 6.9).

```
onSuggest: function(oEvent) {
   var sTerm = oEvent.getParameter("suggestValue");
   var aFilters = [];
   if (sTerm) {
    aFilters.push(new Filter("BusinessPartnerID",
      FilterOperator.StartsWith, sTerm));
}
   oEvent.getSource().getBinding("suggestionItems")
     .filter(aFilters);
}
```

Listing 6.9 Implementierung der Funktion »onSuggest«

Speichern Sie die Änderungen, und sehen Sie sich das Ergebnis an. Geben Sie testweise den Beginn einer gültigen Business-Partner-ID in das Eingabefeld ein, beispielsweise »01«. Sie erhalten eine Liste von Geschäftspartnern mit passender ID aus dem zugrundeliegenden Datenmodell (siehe Abbildung 6.7). Wenn Sie einen Eintrag aus dieser Liste auswählen, wird er in das Eingabefeld übernommen.

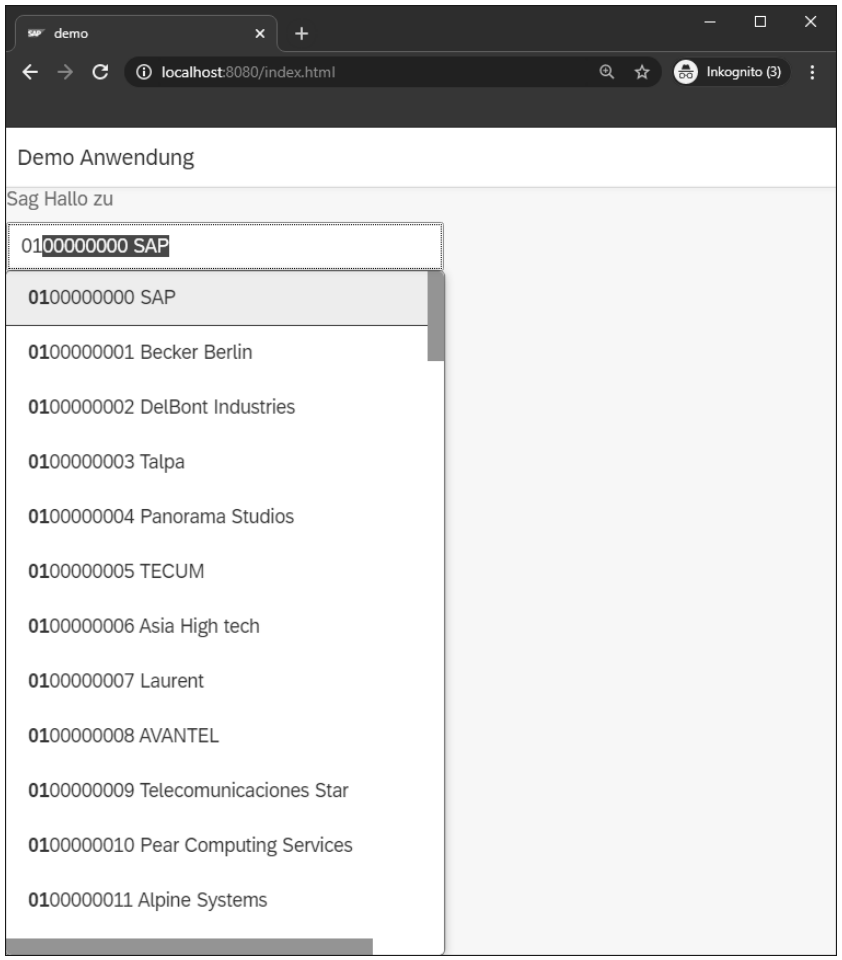

Abbildung 6.7 Darstellung der Vorschlagswerte für »BusinessPartnerID«

6.2.2 Komplexe Controls zur Seitengestaltung

Als Nächstes wollen wir für den ausgewählten Geschäftspartner mehr Details anzeigen lassen. SAPUI5 bietet hierfür vorgefertigte Masken an. Eine dieser Darstellungsformen ist die *Object Page*. Die Objektseite und ihre verwandten Controls werden im Namensraum `sap.uxap` implementiert. Das

zentrale Control ist `sap.uxap.ObjectPageLayout`, das wir nun verwenden wollen, um die Objektdetails für unser `BusinessPartner`-Objekt darzustellen. Damit das Control in der View verwendet werden kann, fügen wir den Namensraum über einen XML-Alias mit dem Bezeichner `uxap` ein (siehe Listing 6.10).

```
<mvc:View controllerName="de.sapui5buch.demo.controller.App"
  displayBlock="true"
  xmlns="sap.m" xmlns:layout="sap.ui.layout"
  xmlns:mvc="sap.ui.core.mvc" xmlns:core="sap.ui.core"
      xmlns:uxap="sap.uxap">
```

Listing 6.10 Einführung des Namensraums »sap.uxap« als XML-Alias

Einfügen des Controls »sap.uxap.ObjectPageLayout«

Anschließend kommentieren wir das gesamte `Page`-Control aus, um der Seite eine andere Struktur zu geben, und fügen dann das zentrale Control `sap.uxap.ObjectPageLayout` in das `App`-Control ein. Wir geben dem Control die ID `ObjectPageLayout` und setzen die Eigenschaften `showHeaderContent` und `toggleHeaderOnTitleClick` auf `false` (siehe Listing 6.11). Hierdurch wird der Kopf der Objektseite nicht angezeigt.

```
<uxap:ObjectPageLayout id="ObjectPageLayout" showHeaderContent=
"false"   toggleHeaderOnTitleClick="false">
</uxap:ObjectPageLayout>
```

Listing 6.11 Implementierung des Controls »sap.uxap.ObjectPageLayout«

Implementierung der Aggregation »headerTitle«

Implementieren Sie nun den Kopf der Objektseite `headertitle` fest. Wir wollen, dass im Kopfbereich des Controls die Wertehilfe angezeigt wird, die wir in Abschnitt 6.2.1 angelegt hatten. Wir fügen daher die Implementierung des `Input`-Feldes in ein Control vom Typ `sap.uxap.ObjectPageDynamicHeaderTitle` ein (siehe Listing 6.12).

```
<uxap:headerTitle>
  <uxap:ObjectPageDynamicHeaderTitle>
   <uxap:heading>
     <Input id="idInputField" width="20rem"
        placeholder="BusinessPartner ID..."
        suggestionItemSelected="onSuggestionItemSelected"
        showSuggestion="true" suggest="handleSuggest"
        suggestionItems="{/d/results}">
       <suggestionItems>
        <core:Item text="{BusinessPartnerID} {CompanyName}"
         />
         </suggestionItems>
       </Input>
```

```
    </uxap:heading>
    </uxap:ObjectPageDynamicHeaderTitle>
  </uxap:headerTitle>
```

Listing 6.12 Implementierung des Objektkopfes

Implementierung der Aggregation »headerContent«

Nun wollen wir im Bereich `headerContent` des Controls `ObjectPageLayout` weitere Details zu unserem Geschäftspartner anzeigen. Wir implementieren hierfür die Aggregation `headerContent` entsprechend (siehe Listing 6.13).

```
<uxap:headerContent>
  <FlexBox wrap="Wrap" fitContainer="true"
     id="headerContent">
    <VBox class="sapUiLargeMarginEnd">
       <HBox>
         <Title text="Partnerdata"/>
            </HBox>
            <HBox>
              <Label class="sapUiTinyMarginEnd"
                   text="Firmenname:" />
              <Text text="{CompanyName}" />
            </HBox>
              <HBox>
              <Label class="sapUiTinyMarginEnd"
                   text="Partner ID:" />
              <Text text="{BusinessPartnerID}" />
          </HBox>
            <HBox>
            <Label class="sapUiTinyMarginEnd"
                text="Legal Form:" />
            <Text text="{LegalForm}" />
          </HBox>
      </VBox>
      <VBox>
        <HBox>
         <Title text="Kontaktdaten"/>
        </HBox>
        <HBox>
         <Label class="sapUiTinyMarginEnd"
                text="URL:" />
         <Text text="{WebAddress}" />
        </HBox>
        <HBox>
         <Label class="sapUiTinyMarginEnd"
```



```
                text="Email-Addresse:" />
            <Text text="{EmailAddress}" />
          </HBox>
          <HBox>
           <Label class="sapUiTinyMarginEnd"
              text="Telefon:" />
           <Text text="{PhoneNumber}" />
          </HBox>
          <HBox>
           <Label class="sapUiTinyMarginEnd"
               text="Fax:" />
           <Text text="{FaxNumber}" />
         </HBox>
       </VBox>
     </FlexBox>
</uxap:headerContent>
```

Listing 6.13 Implementierung der »headerContent«-Aggregation

Implementierung von Sections

Das Control `sap.uxap.ObjectPageLayout` verfügt über sogenannte *Sections*. Diese werden in der Aggregation `sections` des Controls implementiert. Sections werden verwendet, um den `content`-Bereich der Object Page mit Inhalten zu füllen. Eine Section enthält dabei eine zusammengehörende Informationseinheit, so dass Sie Informationen durch die Verwendung von Sections clustern können. Eine Section wird über die Klasse `sap.uxap.ObjectPageSection` implementiert. Eine Section besitzt einen Titel. Die Titel der Sections werden über den eigentlichen Inhalt nebeneinander dargestellt. Eine Section kann wiederum *Untersections* haben. Prinzipiell gibt es für die Verwendung von Sections keine Beschränkung. Ab einer gewissen Anzahl kann die Anwendung jedoch nicht mehr sinnvoll vom Anwender genutzt werden. Grund dafür ist, dass die Titel der Sections grundsätzlich nebeneinander dargestellt werden. Je nach Bildschirmgröße kann es allerdings passieren, dass nicht alle Titel nebeneinander in eine Zeile passen. In diesem Fall werden die nicht mehr darzustellenden Titel in einer Dropdown-Liste visualisiert. Nutzer können dadurch leicht den Überblick verlieren. Achten Sie immer auf einen sauberen Informationsschnitt, werden Sie aber nicht zu feingranular. In unserem Falle wollen wir nur eine Section implementieren, die die Adresse des ausgewählten Geschäftspartners darstellen soll. Die Implementierung können Sie dem Programmbeispiel aus Listing 6.14 entnehmen.

```
<uxap:sections>
  <uxap:ObjectPageSection title="Adresse"
                visible="false" id="addressSection">
    <uxap:subSections>
      <uxap:ObjectPageSubSection>
        <VBox>
         <HBox>
          <Label class="sapUiTinyMarginEnd"
                    text="Strasse:" />
          <Text text="{Street} {Building}" />
         </HBox>
         <HBox>
          <Label class="sapUiTinyMarginEnd"
                    text="Land PLZ Ort:" />
          <Text text="{Country} {PostalCode} {City}" />
         </HBox>
        </VBox>
      </uxap:ObjectPageSubSection>
    </uxap:subSections>
  </uxap:ObjectPageSection>
</uxap:sections>
```

Listing 6.14 Implementierung der Aggregation »sections« des Controls »sap.uxap.ObjectPageLayout«

Im Anschluss müssen Sie sich um die Verbindung zwischen der Auswahl eines `BusinessPartner`-Elements aus der Vorschlagsliste und dem Darstellen der Daten des durch den Nutzer ausgewählten Geschäftspartners kümmern. Hierzu implementieren Sie den Eventhandler `onSuggestionItemSelected` (siehe Listing 6.15). Die Funktion wird aufgerufen, sobald der Anwender einen Eintrag aus der Treffermenge ausgewählt hat (siehe hierzu auch die Implementierung der Wertehilfe in Listing 6.12). Hierzu liest der Eventhandler aus dem Eventobjekt den Parameter `selectedItem` aus und liest den Bindungspfad aus. Dieser Pfad wird der Funktion `_updateUI` übergeben, die wir im nächsten Schritt implementieren.

```
onSuggestionItemSelected : function(oEvent) {
  var oSelectedItem = oEvent.getParameter("selectedItem");
  var sBindingPath =
      oSelectedItem.getBindingContext().getPath();
  this._updateUI(sBindingPath);
}
```

Listing 6.15 Implementierung des Eventhandlers »onSuggestionItemSelected«

»_updateUI«

Den Abschluss unserer Implementierung bildet die Funktion `_updateUI`. Die Funktion erhält als Aufrufparameter den Binding-Pfad des ausgewählten Listeneintrags. Er wird den beiden Controls `headerContent` und `addressSection` übergeben (siehe Listing 6.16). Zum Schluss werden die Controls auf sichtbar gestellt.

```
_updateUI : function(sBindingPath) {
   var oHeaderContent = this.byId("headerContent");
  oHeaderContent.bindElement(sBindingPath);
  var oAdrSection = this.byId("addressSection");
  oAdrSection.setVisible(true);
  oAdrSection.bindElement(sBindingPath + "/Address")
  var oObjLayout = this.byId("ObjectPageLayout");
  oObjLayout.setShowHeaderContent(true);
  oObjLayout.setToggleHeaderOnTitleClick(true);
}
```

Listing 6.16 Die Funktion »_updateUI«

Nachdem Sie alle Änderungen gespeichert haben, testen Sie das Ergebnis. Wenn Sie einen Geschäftspartner ausgewählt haben, wird er entsprechend dargestellt (siehe Abbildung 6.8).

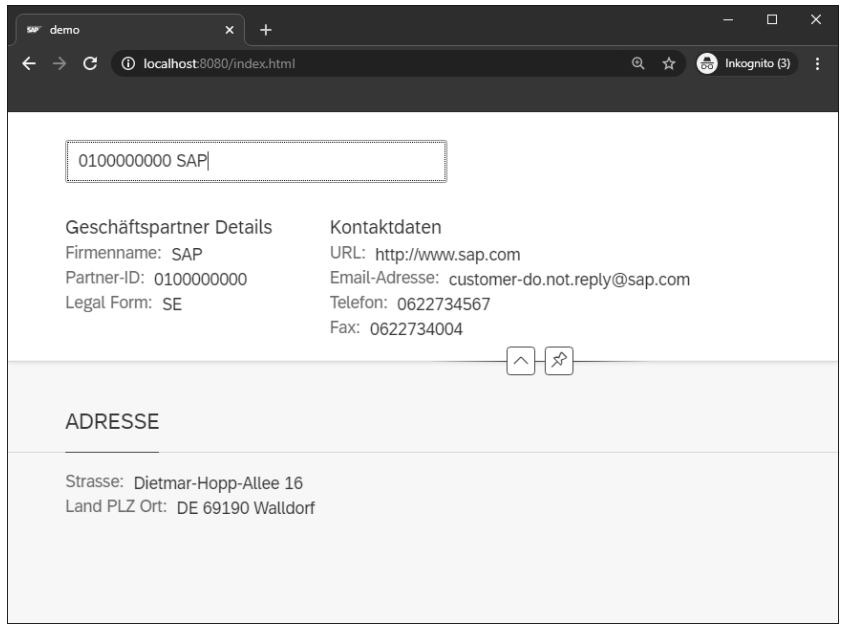

Abbildung 6.8 Das »ObjectPageLayout« im Einsatz

6.3 Arbeiten mit Fragmenten

Wie in den vorangegangenen Abschnitten beschrieben, besitzen Views eine gewisse Grundkomplexität. Views sind dadurch nicht gut für das Modularisieren von Oberflächenteilen geeignet. Besonders dann, wenn die Funktionalität an einer zentralen Stelle, beispielsweise in einem zentralen Controller, gehalten werden soll. Als Ausweg bliebe nur die Verwendung einer gemeinsamen Basisklasse, dies führt aber gegebenenfalls zur Vermengung von Zuständigkeiten und erhöht die Komplexität. Für eine leichtgewichtige Strukturierung bietet SAPUI5 sogenannte *Fragmente* an. Zur Laufzeit werden Fragmente als Unterknoten der View (*UI sub-tree*) bzw. anderer Fragmente im DOM-Baum angelegt und sind somit direkte Kinder des nutzenden Benutzeroberflächen-Artefakts (sprich der View oder eines anderen Fragments). Fragmente können in separaten Dateien, innerhalb einer View oder innerhalb eines anderen Fragments (über die sogenannte *Inlinedefinition*) implementiert werden. Die Inlinedefinition ist zwar möglich, wird aber sehr selten verwendet. Im Gegensatz zu einer View kann ein Fragment den Controller der Eltern-View, einen eigenen Controller oder gar keinen Controller besitzen.

6.3.1 Typen von Fragmenten

Fragmenttypen

In SAPUI5 können Fragmente als HTML-, JavaScript- und XML-Fragmente angelegt werden. Die Namen der Dateien, in denen Fragmente implementiert werden, folgen dem Muster `<NameDesFragments>.fragment.<Typ>`. Als Typ können Sie `html`, `js` oder `xml` angeben.

HTML-Fragmente

HTML

HTML-Fragmente werden in Dateien mit der Endung **.fragment.html** implementiert und besitzen einen einfachen Aufbau (siehe Listing 6.17). Die UI-Controls werden über `div`-Elemente eingebettet. Im Gegensatz zu HTML-Views besitzen HTML-Fragmente kein `template`-Tag.

```
<div
   data-sap-ui-type="sap.m.Button"
   data-press="onPress"
   data-text="Hallo SAPUI5">
</div>
```

Listing 6.17 Einfaches HTML-Fragment

JavaScript-Fragmente

JavaScript

Die Implementierung eines JavaScript-Fragments folgt im Großen und Ganzen der Implementierung einer JavaScript-View. JavaScript-basierte Fragmente werden über die Funktion `sap.ui.jsfragment` implementiert (siehe Listing 6.18). Die Funktion erhält zwei Argumente: den vollqualifizierten Fragmentnamen und die Implementierung des Fragments. Die Implementierung erfolgt über die Funktion `createContent`. Die Funktion erhält durch den Aufrufer optional eine Instanz eines Controllers, der daher als Parameter angegeben wird. Ob er tatsächlich durch den Fragmentkonsumenten mitgegeben wird, ist nicht sicher, da der Konsument nicht gezwungen wird, ein Controller-Objekt zu übergeben. Möchten Sie mehr als ein Control auf gleicher Ebene verwenden, können Sie ein Array von Controls zurückliefern. Listing 6.18 zeigt ein einfaches JavaScript-Fragment.

```
sap.ui.define([
    "sap/m/Button"
], function(Button) {
  "use strict";
  sap.ui.jsfragment("de.sapui5buch.demo.view.JSFragment", {
         createContent : function(oController) {
             var oButton = new Button({
                    text: "Hallo SAPUI5",
                    press: oController.onPress
             });

             return oButton;
         }
      });
});
```

Listing 6.18 Implementierung eines JavaScript-Fragments

XML-Fragmente

XML

XML-Fragmente werden ähnlich wie XML-Views implementiert, mit dem Unterschied, dass sie, solange sie nur ein Control enthalten, kein besonderes umschließendes Control nutzen müssen. Möchten Sie mehr als ein Control auf gleicher Ebene verwenden, müssen Sie die Control-Sammlung einem Wurzel-Control zuordnen. SAPUI5 stellt hierfür das Control `sap.ui.core.FragmentDefinition` zur Verfügung (siehe Listing 6.19).

```
<core: FragmentDefinition xmlns="sap.m" xmlns:core="sap.ui.core">
    <Label text="Label in einem Fragment"/>
    <Button text="Hallo SAPUI5" press="onPress"/>
</core: FragmentDefinition >
```

Listing 6.19 Implementierung eines XML-Fragments

6.3.2 Instanziierung von Fragmenten

Einbettung in Views

Nachdem Sie die Implementierung der drei Fragmentarten kennengelernt haben, wenden wir uns der Instanziierung und Verwendung zu. Fragmente können über zwei verschiedene Wege instanziiert werden. Sie können ein Fragment über JavaScript erzeugen oder es in HTML- oder XML-Views einbetten.

Fragmente mittels JavaScript laden und instanziieren

Das Instanziieren von Fragmenten über JavaScript kann entweder im Rahmen einer JavaScript-View-Implementierung oder im Kontext eines Controllers erfolgen. Die Implementierung im Controller wird in aller Regel dann gewählt, wenn beispielsweise abhängig von einer Aktion in der Benutzeroberflächen entschieden wird, ob Fragment A oder B geladen und instanziiert werden soll.

Dabei ist es egal, zu welchem View-Typ der Controller gehört. Für das Laden und Instanziieren eines Fragments bietet die Klasse `sap.ui.core.Fragment` die Methode `load` an. Die Methode erhält als Aufrufargument eine Map aus Key-Value-Paaren übergeben. Das Programmbeispiel aus Listing 6.20 zeigt Ihnen die Verwendung der Funktion `load` im Rahmen einer JavaScript-View. `load` enthält eine Map aus drei Key-Value-Paaren. Über den Parameter `name` geben Sie den vollqualifizierten Namen des Fragments an, das geladen werden soll. Als zweiten Parameter geben Sie das Attribut `type` mit, das die Werte `XML`, `JS` oder `HTML` enthalten kann. Der dritte Parameter, `controller`, ist optional und wird verwendet, um eine Referenz auf den Controller zu übergeben.

```
sap.ui.define([
    "sap/ui/core/Fragment",
    "sap/ui/layout/HorizonalLayout"
], function(Fragment, HorizontalLayout) {
   "use strict";
 sap.ui.jsview("de.sapui5buch.demo.view.JavaScriptView", {

  getControllerName:function() {
   return "de.sapui5buch.demo.controller.JavaScriptView";
```



```
  },

  createContent : function(oController) {
         var oLayout = new HorizontalLayout();
         Fragment.load({
             name:
                "de.sapui5buch.demo.view.JSFragment",
             type: "JS",
             controller : oController
         }).then(function (oFragment) {
             oLayout.addContent(oFragment);
         });
         return oLayout;
     }
  });
});
```

Listing 6.20 Instanziierung eines Fragments

Wie eingangs erwähnt, muss der übergebene Controller nicht der Controller der entsprechenden View sein. Sie können auch einen im Code implizit angelegten Controller oder eine ausgelagerte Implementierung verwenden. Listing 6.21 zeigt Ihnen die Nutzung eines in der View angelegten Controller-Objekts. Es ist nicht notwendig, das übergebene Objekt von einer Controller-Basisklasse abzuleiten.

```
sap.ui.define([
    "sap/ui/core/Fragment",
    "sap/ui/layout/HorizonalLayout"
], function(Fragment, HorizontalLayout) {
   "use strict";
 sap.ui.jsview("de.sapui5buch.demo.view.JavaScriptView", {
 // getControllerName
  createContent : function(oController) {
    var oMyController = {
      onPress : function(oEvent) {
        //handle press-event
      }
    }
    var oLayout = new HorizontalLayout();
      Fragment.load({
       name: "de.sapui5buch.demo.view.JSFragment",
       type: "JS",
       controller : oMyController
      }).then(function (oFragment) {
```

```
        oLayout.addContent(oFragment);
      });
    return oLayout;
    }
 });
});
```

Listing 6.21 Verwendung eines anderen Objekts als Controller

Fragmente in einer XML-View verwenden

Einbetten mit »Fragment«

XML-Views bieten ebenfalls die Möglichkeit, Fragmente einzubetten. Hierzu nutzen Sie das `Fragment`-Control. Das `Fragment`-Control erhält zwei Attribute (siehe Listing 6.22): zum einen das Attribut `fragmentName`, über das der vollqualifizierte Name des einzubettenden Fragments angegeben wird, zum anderen das Attribut `type`. Es gibt den Typ des Fragments an und kann die Werte `JS`, `XML` oder `HTML` enthalten.

```
<mvc:View controllerName="de.sapui5buch.demo.controller.XMLView"
  xmlns:mvc="sap.ui.core.mvc" xmlns="sap.m"
  xmlns:core="sap.ui.core" xmlns:l="sap.ui.layout">
    <l:HorizontalLayout>
    <l:content>
       <core:Fragment fragmentName=
         "de.sapui5buch.demo.view.XMLFragment" type="XML"/>
       <core:Fragment fragmentName=
         "de.sapui5buch.demo.view.JSFragment" type="JS"/>
       <core:Fragment fragmentName=
        "de.sapui5buch.demo.view.HTMLFragment" type="HTML"/>
    </l:content>
    </l:HorizontalLayout>
</mvc:View>
```

Listing 6.22 Instanziierung von Fragmenten in einer XML-View

6.3.3 IDs in Fragmenten

ID-Präfix-Verarbeitung bei Fragmenten in XML- und HTML-Views

Bei der Arbeit mit IDs in Fragmenten gibt es einiges zu beachten. Wenn Sie in einem Fragment statische IDs verwenden und das Fragment in eine XML-View einbetten, werden die IDs nicht mit einem Präfix versehen. Wenn Sie das Fragment nun mehr als einmal zu einem Zeitpunkt verwenden, führt dies dazu, dass aufgrund der mehrfach vorhandenen ID ein Fehler auftritt. Sie vermeiden diesen Fehler, indem Sie dem Fragment beim Einbetten in eine View bzw. in ein anderes Fragment eine (eindeutige) ID mitgeben. Betrachten wir das gerade Erläuterte an einem Beispiel und nehmen an dieser Stelle an, dass Sie ein Fragment wie in Listing 6.23 gezeigt implementieren.


```
<core:FragmentDefinition xmlns="sap.m"
  xmlns:core="sap.ui.core" xmlns:l="sap.ui.layout">
   <l:HorizontalLayout>
    <Input id="idInput" value="Hello"/>
    <Input value="SAPUI5"/>
   </l:HorizontalLayout>
</core:FragmentDefinition>
```

Listing 6.23 »Input«-Control mit ID in einem Fragment

Betten Sie das Fragment danach entsprechend in eine View ein (siehe Listing 6.24).

```
<mvc:View controllerName="de.sapui5buch.demo.controller.App"
   xmlns:mvc="sap.ui.core.mvc" xmlns="sap.m"
   xmlns:core="sap.ui.core">
 <core:Fragment
   fragmentName="de.sapui5buch.demo.view.IDInFragment"
   type="XML"
 />
</mvc:View>
```

Listing 6.24 Einbetten des Fragments in eine XML-View

Wechseln Sie nach dem Neustart der Anwendung in den Entwicklungswerkzeugen Ihres Browsers in die Elementansicht des DOM-Baums (siehe Abbildung 6.9). Das erste `Input`-Feld enthält kein spezifisches Präfix für das Fragment. Lediglich der View, in die das Fragment eingebettet ist, wurde ein Präfix vorangestellt.

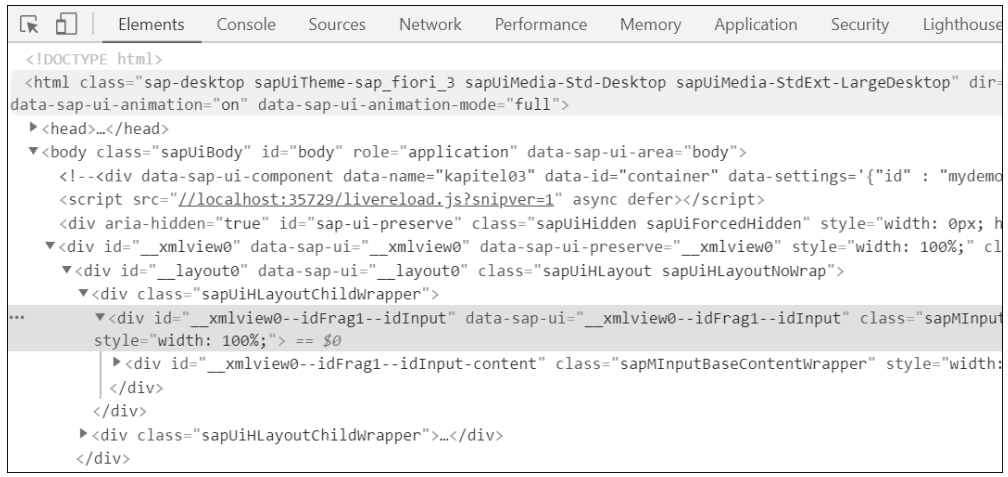

Abbildung 6.9 ID-Verarbeitung in deklarativen Views

Möchten Sie das Fragment beispielsweise mehr als einmal in der View verwenden (siehe Listing 6.25), würden Sie unweigerlich eine Duplicate ID Exception zur Laufzeit im Browser erhalten.

```
<mvc:View controllerName="de.sapui5buch.demo.controller.App"
  xmlns:mvc="sap.ui.core.mvc" xmlns="sap.m"
  xmlns:core="sap.ui.core">
  <core:Fragment
      fragmentName="de.sapui5buch.demo.view.IDInFragment"
      type="XML"/>
   <core:Fragment
      fragmentName="de.sapui5buch.demo.view.IDInFragment"
      type="XML"/>
</mvc:View>
```

Listing 6.25 Wiederverwendung eines Fragments

Starten Sie nun die Anwendung erneut, erhalten Sie wie in Abbildung 6.10 gezeigt eine Duplicate ID Exception.

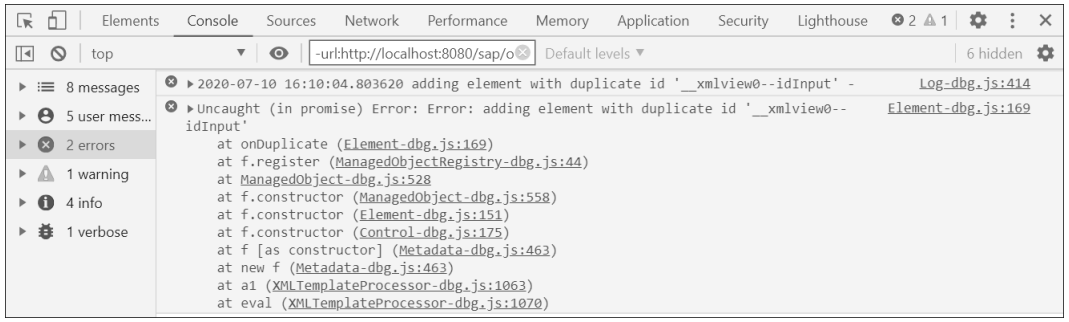

Abbildung 6.10 Duplicate ID Exception

Lösen können Sie dies nur, indem Sie in unserem Beispiel mindestens einem der Fragmente eine ID übergeben (siehe Listing 6.26).

```
<mvc:View controllerName="de.sapui5buch.demo.controller.App"
   xmlns:mvc="sap.ui.core.mvc" xmlns="sap.m"
   xmlns:core="sap.ui.core">
    <core:Fragment
      fragmentName="de.sapui5buch.demo.view.IDInFragment"
      type="XML" id="idFrag1"/>
    <core:Fragment
      fragmentName="de.sapui5buch.demo.view.IDInFragment"
      type="XML"/>
</mvc:View>
```

Listing 6.26 Übergabe einer eindeutigen ID beim Einbetten der Fragmente

Starten Sie nun die Anwendung erneut, ergibt sich der in Abbildung 6.11 gezeigte DOM-Baum.

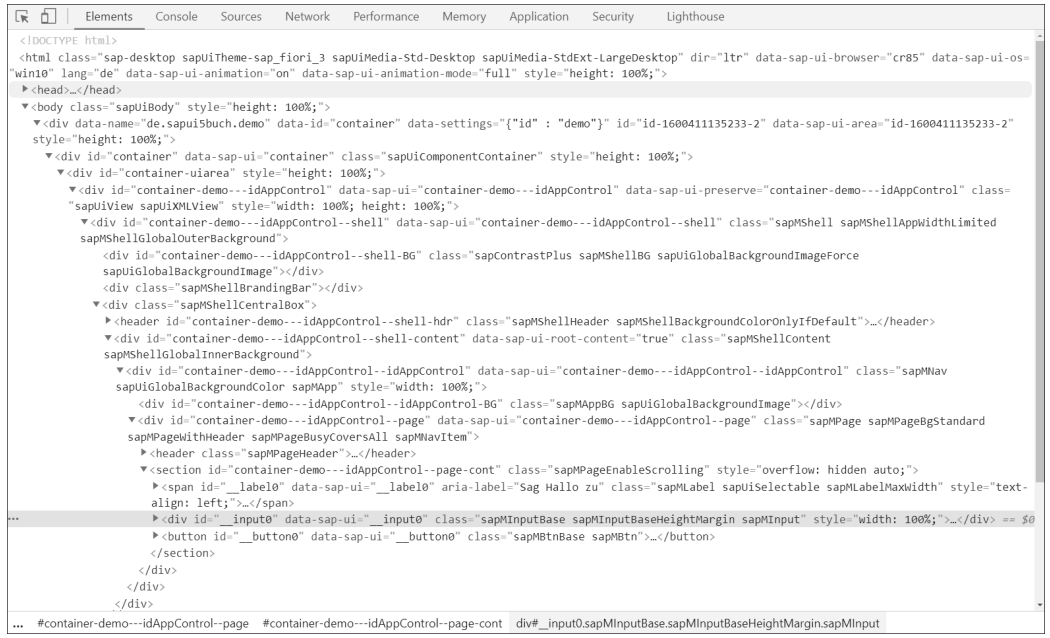

Abbildung 6.11 Verwendung von IDs bei Fragmenten

Bei der Arbeit mit IDs in Fragmenten sollten Sie also die folgenden Regeln beachten:

- Haben SAPUI5-Controls in einem Fragment eine ID, wird diese ID zur Laufzeit nur mit dem Präfix der View versehen, in die das SAPUI5-Control eingebettet ist. Dies gilt nur, wenn das Fragment keine eigene ID besitzt.
- Haben SAPUI5-Controls in einem Fragment mit ID eine ID, erhält die ID des Controls zur Laufzeit ein Präfix, das sich aus der ID der View sowie der ID des Fragments zusammensetzt.

Zugriff auf UI-Controls in Fragmenten mit ID

Um die Regeln an einem Beispiel zu testen, implementieren wir in der View den in Listing 6.27 gezeigten Code. Wir legen zwei `Fragment`-Elemente an. Beide Elemente werden mit der in Listing 6.23 gezeigten Implementierung verbunden. Dem ersten `Fragment`-Element wird jedoch eine ID zugeordnet. Zusätzlich legen wir zwei `Button`-Controls an. In den dazugehörenden Eventhandlers greifen wir zur Laufzeit auf das `Input`-Control des Fragments zu.

```
<mvc:View controllerName="de.sapui5buch.demo.controller.App"
 xmlns:mvc="sap.ui.core.mvc" xmlns="sap.m"
 xmlns:core="sap.ui.core">
  <core:Fragment
```

```
    fragmentName="de.sapui5buch.demo.view.IDInFragment"
    type="XML" id="idFrag1"/>
  <core:Fragment
    fragmentName="de.sapui5buch.demo.view.IDInFragment"
    type="XML"/>
  <Button text="Setze Input 1" press="onPress1"/>
  <Button text="Setze Input 2" press="onPress2"/>
</mvc:View>
```

Listing 6.27 ID-Verarbeitung

Bevor wir mit der Implementierung der Eventhandler fortfahren, testen wir unsere Anwendung. Wie in Abbildung 6.12 zeigt die Anwendung die Fragmentimplementierung zweimal an.

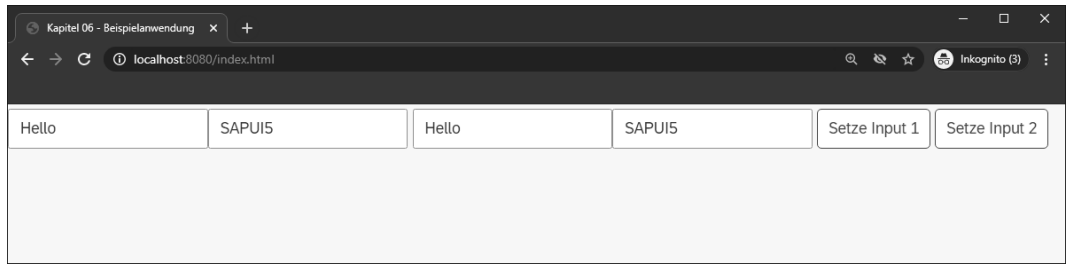

Abbildung 6.12 Darstellung des zweifach eingebetteten Fragments

Eventhandler »onPress1«

Als Nächstes wollen wir den Eventhandler `onPress1` ausprägen, und zwar so, dass bei einem Klick zunächst eine Referenz auf das `Input`-Feld der ersten Fragmentinstanz besorgt und anschließend der `value`-Eigenschaft des `Input`-Feldes der Wert `Hello 1` zugewiesen wird. Das Fragment hat beim Einbetten in die View die ID `idFrag1` erhalten. Gemäß obiger Regel hat nun das `Input`-Control eine zusammengesetzte ID, die auch die View-ID besitzt. Der Zugriff kann daher nicht einfach über die Funktion `byId` und die Übergabe der ID des `Input`-Controls erfolgen. Vielmehr müssen Sie zunächst die ID des `Input`-Controls inklusive Präfix generieren und über diese ID dann die `byId`-Funktion befüllen. Zum Erzeugen einer ID mit Präfix bietet die Fragmentklasse die Funktion `createId` an (siehe Listing 6.28).

```
onPress1 : function(oEvent) {
 var oInput = this.byId(
        sap.ui.core.Fragment.createId("idFrag1","idInput"));
 oInput.setValue("Hello 1");
}
```

Listing 6.28 Zugriff auf ein UI-Control mit Fragment-ID

Nun wollen wir noch das `Input`-Feld des zweiten Fragments mit einem neuen Wert befüllen. Da das zweite Fragment keine ID hat, ist der Zugriff über die `byId`-Funktion der View möglich (siehe Listing 6.29).

```
onPress2 : function(oEvent) {
   var oInput = this.byId("idInput");
    oInput.setValue("Hello 2");
}
```

Listing 6.29 Zugriff ohne Fragment-ID

6.3.4 Fragmente zur Strukturierung einer Anwendung

Nachdem Sie die Implementierung von Fragmenten kennengelernt haben, wollen wir unsere Beispielanwendung mittels Fragmenten besser strukturieren. Legen Sie dafür zunächst im Ordner **view** die folgenden Dateien an:

- **Input.fragment.xml**
- **Header.fragment.xml**
- **AddressSection.fragment.xml**

Erstellen Sie in jeder der drei Dateien ein Control vom Typ `sap.ui.core.FragmentDefinition`, und kopieren Sie den entsprechenden Inhalt der einzelnen Bereiche in die Dateien. Fügen Sie zum Abschluss die entsprechenden XML-Aliasse für die in den Fragmenten verwendeten SAPUI5-Namensräume ein.

Listing 6.30 zeigt Ihnen am Beispiel des `Input`-Feldes, wie das entsprechende Fragment aufgebaut ist.

```
<core:FragmentDefinition
 xmlns="sap.m"
 xmlns:core="sap.ui.core">
  <Input id="idInputField" width="20rem"
    placeholder="BusinessPartner ID..."
    suggestionItemSelected="onSuggestionItemSelected"
    showValueHelp="true"
    valueHelpRequest="onValueHelpRequest"
    showSuggestion="true" suggest="handleSuggest"
    suggestionItems="{/d/results}">
    <suggestionItems>
      <core:Item text="{BusinessPartnerID} {CompanyName}" />
    </suggestionItems>
  </Input>
</core:FragmentDefinition>
```

Listing 6.30 Implementierung des Fragments »Input.fragment.xml«

Nachdem Sie die drei Fragmente angelegt und die Inhalte entsprechend kopiert haben, ändern Sie die Implementierung der Datei **App.view.xml** so ab, dass die Fragmente verwendet werden (siehe Listing 6.31).

```
<mvc:View controllerName="de.sapui5buch.demo.controller.App"
  displayBlock="true"
  xmlns="sap.m" xmlns:layout="sap.ui.layout"
  xmlns:mvc="sap.ui.core.mvc" xmlns:core="sap.ui.core"
  xmlns:uxap="sap.uxap">
  <App id="idAppControl" >
    <pages>
     <uxap:ObjectPageLayout id="ObjectPageLayout"
        showHeaderContent="false"
        toggleHeaderOnTitleClick="false">
      <uxap:headerTitle>
       <uxap:ObjectPageDynamicHeaderTitle>
        <uxap:heading>
          <core:Fragment
          fragmentName="de.sapui5buch.demo.view.Input"
          type="XML"/>
        </uxap:heading>
       </uxap:ObjectPageDynamicHeaderTitle>
      </uxap:headerTitle>
      <uxap:headerContent>
       <core:Fragment
        fragmentName="de.sapui5buch.demo.view.Header"
        type="XML"/>
      </uxap:headerContent>
      <uxap:sections>
       <core:Fragment
        fragmentName="de.sapui5buch.demo.view.AddressSection"
        type="XML"/>
      </uxap:sections>
     </uxap:ObjectPageLayout>
    </pages>
  </App>
</mvc:View>
```

Listing 6.31 Implementierung der »App.view.xml« mit Fragmenten

Testen Sie die Anwendung zum Abschluss erneut. Bis auf die bessere Struktur der Anwendung haben sich keine Laufzeitänderungen, also Funktionsänderungen der Anwendung, ergeben.

6.4 Dialoge implementieren und verwenden

Dieser Abschnitt widmet sich der Implementierung von Dialogen. Wir zeigen Ihnen, wie Sie unter Verwendung von Fragmenten eigene Dialoge gestalten und wie Sie vorgefertigte Dialoge, die von SAPUI5 bereitgestellt werden, nutzen können.

6.4.1 Implementierung eigener Dialoge

»sap.m.Dialog«

Für die Implementierung eigener Dialoge, bei der Sie den Inhalt und die angebotene Funktionalität selbst implementieren, bietet SAPUI5 die Klasse `sap.m.Dialog` an. Die Implementierung leitet sich von `sap.ui.core.Control` ab.

Dialoge sind eine spezielle Form eines Controls, sie werden über dem eigentlichen View-Inhalt angezeigt und gehören somit nicht zur View. Dies ist besonders dahingehend interessant, dass HTML-Artefakte eines Dialogs nicht Bestandteil der HTML-Artefakte einer View sind. Sie existieren, nachdem der Dialog angezeigt wird, unabhängig voneinander.

Fragmente und Dialoge

Fragmente spielen bei der Entwicklung von Dialogen eine wichtige Rolle, denn es ist übliche Praxis, dass die Implementierung eines Dialogs innerhalb eines Fragments erfolgt. Das Fragment dient als Klammer um das eigentliche `Dialog`-Control. Das Programmbeispiel zeigt das Grundgerüst einer Dialogimplementierung:

```
<core:Fragment xmlns="sap.m" xmlns:core="sap.ui.core"
    xmlns:f="sap.ui.layout.form">
   <Dialog id="myDialog" title="Demo Dialog">
   </Dialog>
</core:Fragment>
```

Nachdem das Grundgerüst des Dialogs steht, müssen Sie den Inhalt implementieren. Für den Inhalt des Dialogs stellt die Klasse `sap.m.Dialog` die Aggregation `content` zur Verfügung. Des Weiteren bietet die Dialogimplementierung die Aggregationen `beginButton`, `endButton` und `buttons`. Die beiden erstgenannten Aggregationen können Sie verwenden, um am Anfang bzw. am Ende der Fußzeile des Dialogs jeweils eine Schaltfläche hinzuzufügen. Die Aggregation `buttons` kann eingesetzt werden, um in der Fußzeile eine beliebige Anzahl von Schaltflächen einzufügen. Wenn `buttons` benutzt wird, werden die möglicherweise bereits vorhandenen Schaltflächen in den anderen beiden Aggregationen ignoriert und nicht dargestellt. Listing 6.32 zeigt eine Beispielimplementierung.

```
<beginButton>
  <Button press="onCloseDialog" text="Ok"/>
</beginButton>
<content>
  <f:SimpleForm >
   <f:content>
    <Label text="Name"/>
    <Input id="idInput"/>
   </f:content>
  </f:SimpleForm>
</content>
```

Listing 6.32 Implementierung eines Dialogs

Dialoge werden durch den Controller einer View instanziiert und zur Anzeige gebracht. Die Instanziierung eines Dialogs erfolgt nach dem gleichen Muster, wie Fragmente über JavaScript geladen werden. Listing 6.33 zeigt die Implementierung der Funktion `onOpenDialog` im Controller der `App.controller.js`.

```
onOpenDialog : function () {
   if (!this.byId("myDialog")) {
     Fragment.load({
          id: this.getView().getId(),
          name: "de.sapui5buch.demo.view.Dialog",
          type: "XML",
          controller : this
     }).then(function (oDialog) {
        oView.addDependent(oDialog);
        oDialog.open();
     });
   } else {
     this.byId("myDialog").open();
   }
}
```

Listing 6.33 Implementierung der Funktion »onOpenDialog«

Über die `if`-Abfrage wird geprüft, ob im DOM-Baum bereits ein Element mit der ID `myDialog` existiert. Ist dies nicht der Fall, wird das Fragment entsprechend geladen. In der Funktion, die der `then`-Funktion des Promise übergeben wurde, ist besonders die Verwendung der Funktion `addDependent` interessant. Wie bereits erwähnt, sind Dialoge unabhängig von der View. Dies bedeutet, dass Modelle, die im Kontext der View stehen, nicht automatisch

im Dialog erreichbar sind. Zudem ist der Lebenszyklus der beiden Artefakte entkoppelt. Wird die View verlassen und daraufhin aus dem DOM-Baum entfernt, wird der dazugehörige Dialog (oder die dazugehörenden Dialoge) nicht automatisch mit aus dem DOM entfernt. Dies führt unter Umständen zu DOM-Element-Leichen. Um die Lebenszyklen des Dialogs und der View miteinander zu verbinden und die durch die View zur Verfügung gestellten Modelle im Dialog erreichbar zu machen, ist es notwendig, die Funktion `addDependent` aufzurufen. Nachdem dies erfolgt ist, wird der Dialog über die Funktion `open` angezeigt.

Das Schließen eines Dialogs erfolgt über die `close`-Funktion der Dialoginstanz:

```
onCloseDialog : function(oEvent) {
    this.byId("myDialog").close();
}
```

6.4.2 Verwendung von Dialogklassen

Nachdem Sie gelernt haben, wie Sie einen eigenen Dialog implementieren, möchten wir an dieser Stelle noch kurz die Verwendung eines vordefinierten Dialogs zeigen. Das Beispiel soll so verändert werden, dass der Nutzer nicht nur eine Vorschlagsliste angezeigt bekommt, sondern zusätzlich die Möglichkeit hat, über F4 eine echte Wertehilfe aufzurufen. Hierzu kommt das Control `sap.m.SelectDialog` zum Einsatz. `SelectDialog` zeigt eine Liste von Werten und bietet ein Suchfeld, über das der Inhalt der Liste durchsucht werden kann.

Erweiterung des »sap.m.Input«-Controls

Damit Sie die F4-Hilfe über das `Input`-Control verwenden können, müssen Sie die Implementierung in **App.view.xml** um zwei Argumente erweitern. Fügen Sie das Attribut `showValueHelp` ein, und setzen Sie den Wert des Attributs auf `true`. Führen Sie danach einen Eventhandler für das Event `valueHelpRequest` ein (siehe Listing 6.34). Wir legen für das Event die Funktion `onValueHelpRequest` fest.

```
<Input id="idInputField" width="20rem"
      placeholder="BusinessPartner ID..."
      suggestionItemSelected="onSuggestionItemSelected"
      showValueHelp="true"
      valueHelpRequest="onValueHelpRequest"
      showSuggestion="true" suggest="handleSuggest"
      suggestionItems="{/d/results}">
      <suggestionItems>
```

```
        <core:Item text="{BusinessPartnerID} {CompanyName}" />
      </suggestionItems>
</Input>
```

Listing 6.34 Erweiterung des »Input«-Controls um eine F4-Funktionalität

Implementierung des Dialogs

Wie Sie bereits wissen, werden Dialoge als Fragmente in eigenen Dateien implementiert. Dies gilt auch für `SelectDialog`. Legen Sie daher im **view**-Ordner des Projekts eine neue Datei namens **SelectDialog.fragment.xml** an. Als Nächstes prägen Sie die Implementierung aus.

Das Control `sap.m.SelectDialog` besitzt eine Aggregation mit dem Namen `items`. Sie enthält die Daten, die wir in der Liste im Dialog anzeigen wollen: das `BusinessPartnerSet`. Aus dieser Liste kann der Nutzer einen Wert auswählen, oder er kann über das Suchfeld nach einem Listeneintrag suchen. Zudem besitzt das Control vier Events. Für uns interessant sind die Events `search`, `cancel` und `confirm`. `search` wird aufgerufen, wenn der Nutzer einen Suchbegriff in das Suchfeld eingibt und die Suche startet. Das Event `cancel` wird aufgerufen, wenn der Nutzer auf die Abbrechen-Schaltfläche des Dialogs klickt. `confirm` wird bei Auswahl eines Eintrags aus der Liste aufgerufen.

Implementierung des »SelectDialog«-Fragments

Nun implementieren wir das Fragment. Wir legen für die besprochenen Events die Eventhandler `onSearch` und `onClose` an. `onClose` wird sowohl für `cancel` als auch für `confirm` verwendet (siehe Listing 6.35).

```
<core:FragmentDefinition
   xmlns="sap.m"
   xmlns:core="sap.ui.core">
  <SelectDialog noDataText="Kein Business-Partner gefunden"
      title="Wählen Sie einen Business-Partner"
      search="onSearch" confirm="onClose" cancel="onClose"
      items="{path: '/d/results'}">
      <StandardListItem
        title="{BusinessPartnerID}"
        description="{CompanyName}"
        iconDensityAware="false"
        iconInset="false"
        type="Active" />
  </SelectDialog>
</core:FragmentDefinition>
```

Listing 6.35 Den »SelectDialog« implementieren

»valueHelpRequest«

Damit der Dialog angezeigt wird, müssen wir den Eventhandler für `valueHelpRequest` implementieren. Die Aufgabe dieser Funktion ist es, den Wert

auszulesen, der im `Input`-Feld eingeben wurde. Danach erstellt sie den Dialog über das Fragment sowie einen Filter für den eingegebenen Wert, den sie direkt anwendet (siehe Listing 6.36).

```
onValueHelpRequest : function(oEvent) {
   var sInputValue = oEvent.getSource().getValue();
   this._sInputId = oEvent.getSource().getId();
   if (!this._oValueHelpDialog) {
    this._oValueHelpDialog = sap.ui.xmlfragment(
       "de.sapui5buch.demo.view.SelectDialog",
       this);
    this.getView().addDependent(this._oValueHelpDialog);
   }
   this._oValueHelpDialog.getBinding("items").filter(
     [new Filter("BusinessPartnerID",
      FilterOperator.Contains, sInputValue
   )]);
   this._oValueHelpDialog.open(sInputValue);
}
```

Listing 6.36 Implementierung der Funktion »onValueHelpRequest«

Eventhandler implementieren

Nun müssen noch die Eventhandler des `SelectDialog`-Objekts implementiert werden. Wir beginnen mit der Funktion `onSearch`. Die Funktion liest den Parameter `value` aus dem Event aus. Der Parameter enthält den im Suchfeld eingegebenen Wert, der verwendet wird, um ein `Filter`-Objekt zu instanziieren und entsprechend zu konfigurieren. Das erstellte `Filter`-Objekt wird auf der `items`-Aggregation der Liste im `SelectDialog` angewandt (siehe Listing 6.37).

```
onSearch: function(oEvent) {
  var sValue = oEvent.getParameter("value");
  var oFilter = new Filter(
     "BusinessPartnerID",
     FilterOperator.Contains, sValue
  );
   oEvent.getSource().getBinding("items").filter([oFilter]);
}
```

Listing 6.37 Implementierung der Funktion »onSearch«

Nun müssen Sie noch die `onClose`-Funktion implementieren. Sie wird einerseits beim Schließen des Dialogs, andererseits bei der Auswahl eines Eintrags aus der Liste aufgerufen. Die Funktion liest den Parameter `selected-`

`Item` aus dem Eventobjekt und ruft im Anschluss die Funktion `_updateUI` auf. Der Funktion wird der Binding-Pfad des ausgewählten Listeneintrags übergeben (siehe Listing 6.38).

```
onClose : function(oEvent) {
   var oSelectedItem =
     oEvent.getParameter("selectedItem");
   if (oSelectedItem) {
     this._updateUI(
      oSelectedItem.getBindingContext().getPath());
   }
}
```

Listing 6.38 Die Funktion »onClose«

Nachdem die Implementierung fertiggestellt ist, können Sie das Ergebnis testen. Starten Sie die Anwendung neu, setzen Sie, falls noch nicht geschehen, den Cursor in das Eingabefeld, und drücken Sie F4. Das Ergebnis: Der `sap.m.SelectDialog` wird korrekt angezeigt (siehe Abbildung 6.13).

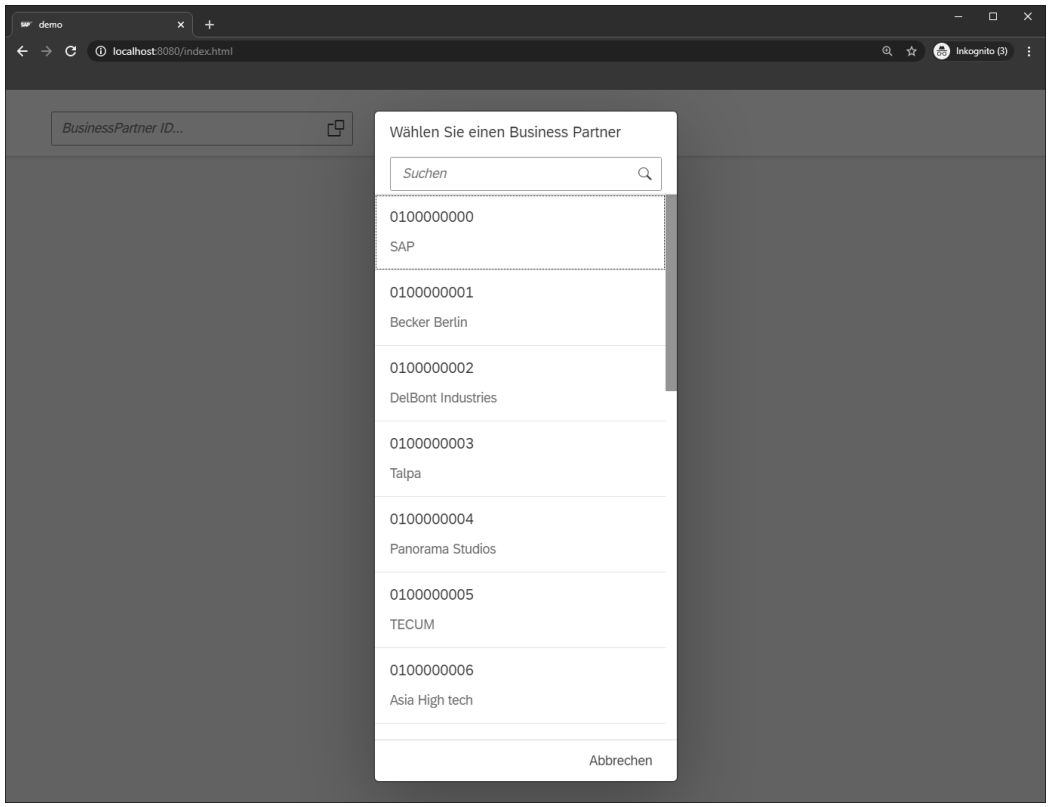

Abbildung 6.13 Der »sap.m.SelectDialog« im Einsatz

Geben Sie zum Testen den Wert »3« in das Suchfeld ein, und bestätigen Sie mit ↵ oder einem Klick auf das Lupensymbol. Die `BusinessPartner` werden anhand des Filterkriteriums gefiltert. Zum Abschluss wählen Sie einen Geschäftspartner aus der Liste aus, zum Beispiel die Nummer **0100000013**. Die Anwendung zeigt Ihnen die Details an (siehe Abbildung 6.14).

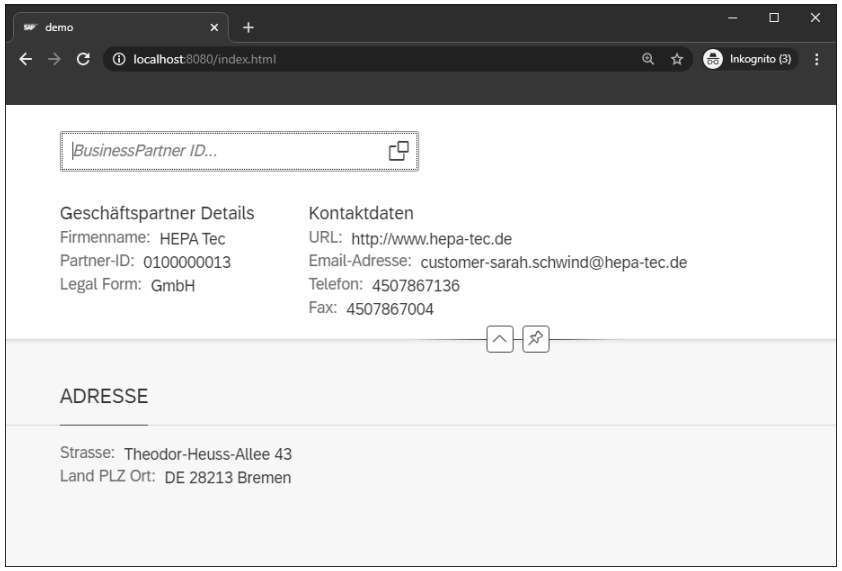

Abbildung 6.14 Details zum ausgewählten Geschäftspartner HEPA Tec

Kapitel 11

Routing und Navigation

In diesem Kapitel lernen Sie die Standardaspekte der Navigation in SAPUI5 in Vollbildanwendungen und Master-Detail-Szenarien kennen und erhalten einen Einblick in die komponentenübergreifende Navigation.

Beim Wechsel zwischen verschiedenen Ansichten oder Seiten innerhalb Ihrer Anwendungen kommt das *SAPUI5-Routing* zum Einsatz. In diesem Kapitel lernen Sie die grundlegenden Navigationskonzepte von SAPUI5 kennen und wenden sie auf unser Master-Detail-Szenario an.

11.1 Einführung in die Navigationskonzepte von SAPUI5

Navigationskonzepte

In SAPUI5 kommt ein hashbasiertes Routingsystem zum Einsatz. Anhand einer Notation, die bestimmten, festzulegenden Mustern folgt, kann eine Navigationsroute innerhalb der Anwendung angesteuert werden. Dieser sogenannte *Hash* folgt in der Browser-URL auf das Hash-Zeichen #, das wiederum der grundlegenden URL der Applikation angehängt wird. Eine typische URL zu einer SAPUI5-Anwendung hat daher diese Form:

http(s)://basislink_zur_anwendung/#/route

Das Auslesen des Hash und die darauffolgende Navigation ist Aufgabe des *SAPUI5-Routers*. Dieses Modul wird beim Anwendungsstart durch die Komponente initialisiert und reagiert fortan auf Änderungen der Browser-URL bzw. des Hash.

11.1.1 Übersicht

Routingkonfiguration

Mit Hilfe einer *Routingkonfiguration*, die im Anwendungsdeskriptor definiert ist, kann der Router eine getroffene Route erkennen, die dazugehörigen Navigationsziele und damit assoziierten Views laden und in den dafür vorgesehenen Navigationscontainern platzieren (siehe Abbildung 11.1).

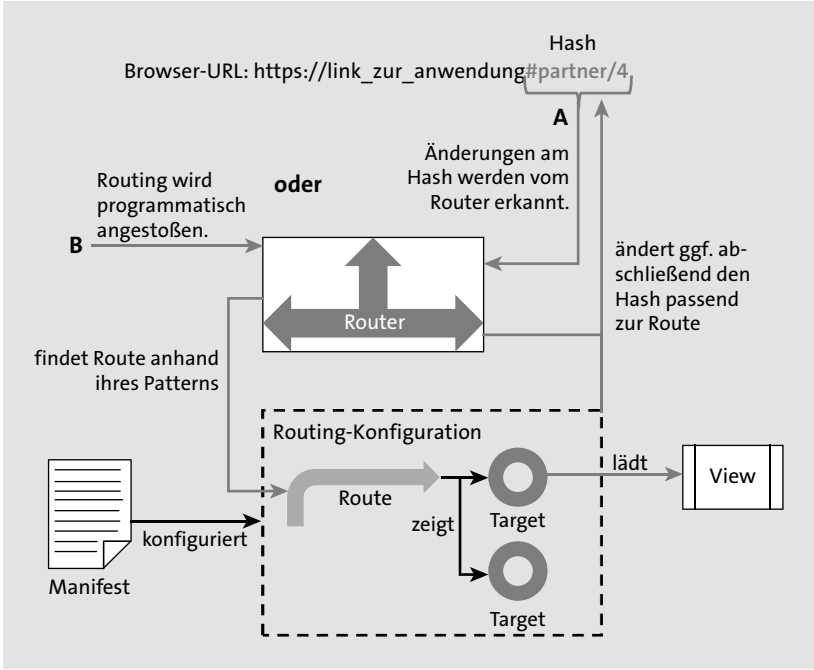

Abbildung 11.1 Routingkonzept in SAPUI5

Ein Navigationsvorgang kann auf zwei Wegen angestoßen werden:

- automatisch durch Ändern des Hash in der Browser-URL
- programmatisch, etwa durch Aufruf der `navTo()`-Funktion des Routers

Routerfunktion

Ändert sich der Hash in der Browser-URL, wird er ausgelesen. Dies gilt sowohl für den ersten Aufruf der Anwendung mit leerem Hash als auch für Hashes mit Routingparametern. Über diesen Navigationsweg ist es Anwendern möglich, eine Route direkt einzugeben oder über ein gespeichertes Lesezeichen aufzurufen.

Die programmatische Navigation folgt typischerweise aus der Bedienung der Anwendung. Eine solche Navigation kann beispielweise stattfinden, wenn Anwender eine Ressource auswählen und die Anwendung zur Detailansicht für diese Ressource navigiert. Ebenso kann der Anwender von einer Ansicht auf die vorige zurücknavigieren.

Bei der programmatischen Navigation wird die Route direkt über ihren Namen angesteuert. Bei der Navigation über den Hash wird der Name gegen ein Muster (Pattern) abgeglichen, und bei einer Übereinstimmung wird die entsprechende Route angesteuert. Wird eine Route programmatisch angesteuert, ersetzt der Router den Hash in der Browser-URL automatisch passend zum Muster der Route. Eventuell übergebene Parameter werden ebenfalls Teil des Hash.

Routen und Targets

Routen

Eine *Route* ist somit das zentrale Objekt des Navigationskonzeptes. Sie wird durch die Klasse `sap.ui.core.routing.Route` abgebildet. Neben ihrem Namen für die programmatische Ansteuerung und ihrem Muster für die hashbasierte Ansteuerung gehört zu jeder Route eine Liste von Navigationszielen, sogenannte *Targets*. Eine Route verweist auf ein oder mehrere Targets.

Targets

Ein Target beschreibt eine Zielansicht und definiert die Details für das Laden dieser Ansicht. Unter anderem kann ein Target definieren, welche View geladen wird und wo sie – innerhalb der vom jeweiligen Navigationscontainer gebotenen Möglichkeiten – platziert wird. Weitere Optionen umfassen das Definieren einer Übergangsanimation für den Navigationsvorgang oder das Setzen einer Ebene (`viewLevel`), über die eine implizite Hierarchie unterschiedlicher Targets festgelegt werden kann.

Targets bieten den Vorteil, dass sie erst dann geladen werden, wenn sie wirklich benötigt werden. Die dazugehörigen Views werden erst instanziiert, wenn die entsprechende Route angesteuert wird, was sich positiv auf die Anwendungsperformance auswirken kann.

Navigationscontainer

Root-Control für die Navigation

Eine View, die durch ein Target geladen wird, wird zur Anzeige in einem *Navigationscontainer* platziert. Der Navigationscontainer stellt das Basis-Control dar, innerhalb dessen die Navigation stattfindet. Dies kann beispielsweise ein `sap.m.App`-Control oder auch ein `sap.m.SplitApp` oder `sap.f.FlexibleColumnLayout` sein und wird typischerweise auf der `rootView` der Anwendung platziert.

Der Navigationscontainer enthält je nach Layout eine oder mehrere Control-Aggregationen, in die die Views geladen werden. Abbildung 11.2 zeigt den grundsätzlichen Ablauf.

Beim Start der Anwendung werden die zur Standardroute gehörenden Views in der Control-Aggregation des Navigationscontainers platziert. Wird dann zu einem anderen Target navigiert, ersetzen dessen Views die bisher vorhandenen.

Beim Ansteuern einer Route können die Views mehrerer in der Route enthaltener Targets in verschiedenen Control-Aggregationen platziert werden. Dies ist beispielsweise in Master-Detail-Szenarien sinnvoll (siehe Abbildung 11.3).

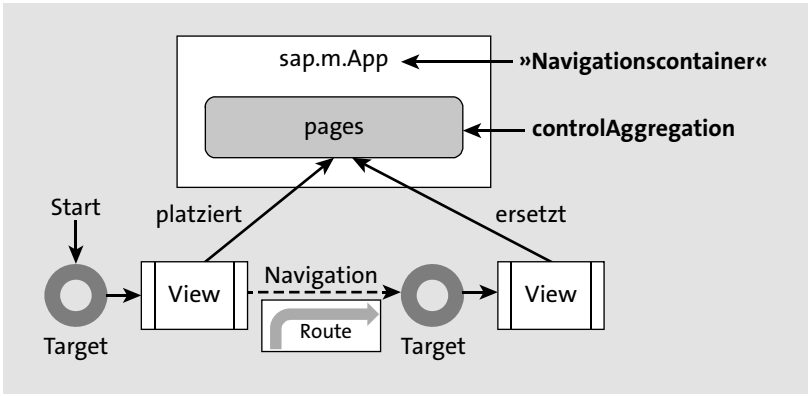

Abbildung 11.2 Navigationscontainer und Control-Aggregation

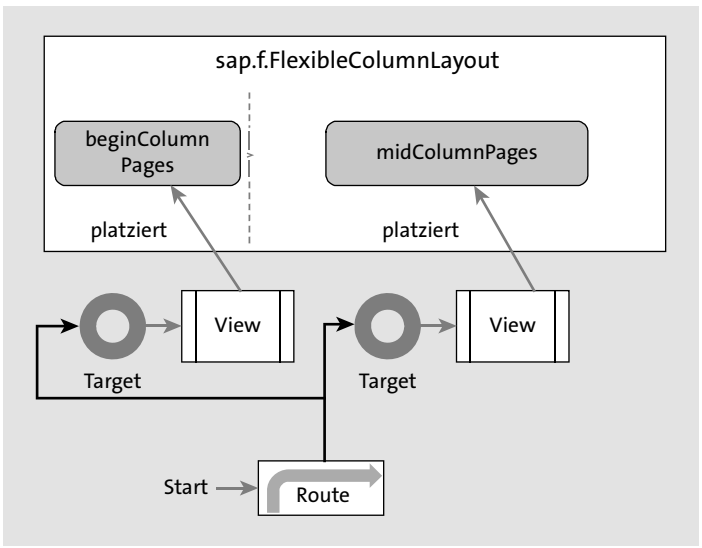

Abbildung 11.3 Navigation mit mehreren Control-Aggregationen

11.1.2 Routingkonfiguration

Die Routen mitsamt ihren Namen, Pattern und Referenzen auf Targets sind in der Routingkonfiguration im Manifest deklariert. Die Routingkonfiguration ist im Abschnitt `sap.ui5.routing` des Manifests verortet.

Standardeinstellungen der Routingkonfiguration

Im Unterpunkt `config` legen Sie die Standardeinstellungen der Routingkonfiguration fest. Diese können gegebenenfalls von Routen und Targets überschrieben werden, ersparen Ihnen aber ansonsten eine mehrfache Konfiguration derselben Werte für unterschiedliche Routen oder Targets.

Zu den Einstellungen im `config`-Bereich gehören:

- `routerClass` – die Klasse des instanziierten Routers
- `viewType` – der Standardtyp der geladenen Views (z. B. `XML`)
- `path` – der Standardpfad zu den Views der Anwendung
- `controlId` – die ID des zu verwendenden Navigationscontainers
- `controlAggregation` – die Standardaggregation, in der geladene Views platziert werden
- `async` – die Angabe, ob Targets asynchron geladen werden

Die weiteren Einstellungen werden je Route bzw. je Target vorgenommen.

Router

Routerklasse

Welche Routerklasse für eine Anwendung gewählt wird, hängt maßgeblich von dem verwendeten Navigationscontainer ab. Die Basisklasse `sap.ui.core.routing.Router` enthält den Großteil der für das Routing benötigten Funktionalität. In der Regel verwenden Sie jedoch eine davon abgeleitete Spezialisierung, die entsprechende Zusatzmöglichkeiten für die jeweilige Anwendung bietet.

Verwenden Sie die Klasse `sap.m.routing.Router` für mobile Anwendungen, in denen Sie mit einer `sap.m.App` oder `sap.m.SplitApp` als Navigationscontainer hauptsächlich zwischen verschiedenen Seiten im Vollbild navigieren. Der hierfür spezialisierte Router bietet Ihnen die Möglichkeit, Übergangsanimationen und View-Ebenen zu definieren, die zu einem konsistenten Navigationsfluss beitragen.

Nutzen Sie hingegen die Klasse `sap.f.routing.Router`, um innerhalb eines `sap.f.FlexibleColumnLayout`-Controls zu navigieren. Der Router stellt Ihnen hierbei die Möglichkeit zur Verfügung, das Ziellayout der jeweiligen Route deklarativ festzulegen.

Vorbereitung auf das Routing

In der Demoimplementierung verwenden wir die in Listing 11.1 gezeigte Konfiguration in **manifest.json**, um die Anwendung für das Routing vorzubereiten:

```
"sap.ui5": {
  "rootView": {
    "viewName": "de.sapui5buch.demo.view.App",
    "type": "XML",
    "async": true,
    "id": "idAppControl"
  },
  "routing": {
```

```
    "config": {
      "routerClass": "sap.f.routing.Router",
      "viewType": "XML",
      "path": "de.sapui5buch.demo.view",
      "controlId": "masterDetailBase",
      "controlAggregation": "beginColumnPages",
      "async": true
    },
    "routes": [],
    "targets": {}
  }
}
```

Listing 11.1 Grundlegende Routingkonfiguration im Manifest (Demo)

Stellen Sie zunächst sicher, dass die `rootView` korrekt eingestellt ist. Fügen Sie dann `routing.config` mit den gezeigten Werten ein, falls sie noch nicht vorhanden ist.

In der Demoimplementierung arbeiten wir mit dem `FlexibleColumnLayout`. Setzen Sie daher als `routerClass` die Klasse `sap.f.routing.Router`. Da wir, soweit möglich, ausschließlich XML-Views verwenden, geben Sie `XML` als `viewType` an.

Geben Sie im Parameter `path` unser Verzeichnis **view** mit dem Anwendungspräfix an. Ihre Targets beziehen sich später auf diesen Pfad, so dass Sie die dortigen Angaben relativ zum **view**-Verzeichnis setzen können. So können Sie in der Regel direkt den Dateinamen der View ohne weitere Präfixe angeben.

Im Parameter `controlId` referenzieren Sie das Root-Control, das als Navigationscontainer dienen wird. Geben Sie als Wert die ID des in der View **App.view.xml** verwendeten `FlexibleColumnLayout`-Controls an. Hier stehen Ihnen mehrere Aggregationen zur Verfügung (siehe Abbildung 11.3): `beginColumnPages`, `midColumnPages` und (nicht in der Abbildung gezeigt) `endColumnPages`. Die im Standardlayout (`OneColumn`) genutzte Aggregation ist `beginColumnPages`. Geben Sie sie für den Parameter `controlAggregation` an.

Routerinitialisierung

Stellen Sie abschließend sicher, dass der Router instanziiert wird. Dies geschieht in der `init()`-Funktion der **Component.js** (siehe Listing 11.2).

```
init: function() {
  // call the base component's init function
  UIComponent.prototype.init.apply(this, arguments);
  // enable routing
  this.getRouter().initialize();
  // …
}
```

Listing 11.2 Routerinitialisierung in der »Component.js«

In den nächsten Schritten legen Sie Routen und Targets an, so dass der Router zum Einsatz kommt.

Routen

Routen konfigurieren

Sie konfigurieren die Routen Ihrer Anwendung im Abschnitt `sap.ui5.routing.routes` des Manifests. `routes` stellt hierbei ein Array, in dem Sie Routen als JSON-Objekte anlegen. Bei der Konfiguration sind je Route drei Parameter von zentraler Bedeutung:

- `name` – der eindeutige Name der Route
- `pattern` – das Muster, nach dem die Route angesteuert wird, sofern der Hash in der Browser-URL diesem Muster entspricht
- `target` – eine Liste von Targets, die durch die Route geladen werden

Wie in Abschnitt 11.1.1 erwähnt, können Routen direkt programmatisch oder über einen passenden Hash angesteuert werden. Über den konfigurierten Parameter `name` kann die Route von der Anwendung aus programmatisch aufgerufen werden. Der Parameter `pattern` legt das Muster fest, über das die Route per Hash aufgerufen wird. Das Pattern bietet wiederum verschiedene Konfigurationsmöglichkeiten:

- hartkodierte Patterns
- parametrisierte Patterns
- Patterns mit Query-Parametern
- *Rest-as-String-Parameter*

Routenpatterns

Hartkodierte Patterns enthalten einen fest vorgegebenen Teil, der durch den Hash genau wiedergegeben werden muss, um die Route aufzurufen. Ein Beispiel für ein solches Pattern wäre `/partner` in Ihrer Routingkonfiguration. Die URL zum Aufruf entspricht dann dem Muster *https://link_zur_anwendung/#/partner*. Nutzen Sie hartkodierte Patterns, um statische Bereiche Ihrer Anwendung zu kennzeichnen.

Routenparameter

Parametrisierte Patterns erlauben Ihnen, dynamische Routen anzulegen, die Pflicht- oder optionale Parameter enthalten. Sie konfigurieren Pflichtparameter, indem Sie den gewünschten Parameternamen in geschweifte Klammern setzen: `/partner/{partnerId}`. Die Route wird nun durch Hashes

wie *#/partner/1*, *#/partner/a* etc. aufgerufen. Beim programmatischen Ansteuern einer solchen Route müssen Sie einen Parameter `partnerId` übergeben.

Optionale Parameter konfigurieren Sie mit Doppelpunkten: `/partner/:partnerId:`. Eine solche Route wird sowohl durch einen Hash *#/partner* als auch durch *#/partner/1* angesteuert. Beim programmatischen Aufruf steht es Ihnen frei, den Parameter `partnerId` mitzugeben. Nutzen Sie parametrisierte Routen, um spezifische, benannte Parameter zu definieren und diese bei der Navigation zu übergeben. Auf diese Art realisieren Sie beispielsweise die Auswahl von Detailseiten und generell die Interaktion mit den Geschäftsobjekten Ihrer Anwendung.

Patterns mit Query-Parametern

Patterns mit Query-Parametern enthalten eine beliebige Reihe von URL-Parametern. Wie bei parametrisierten Patterns können sie als Pflicht- oder optionale Parameter angelegt werden.

Konfigurieren Sie Query-Parameter beispielsweise entweder mit `/partner{?query}` für Pflichtparameter oder `/partner:?query:` für optionale Parameter. Beide Patterns reagieren auf Hashes wie *#/partner?tab=Orders&filter=none*, während das zweite Pattern entsprechend auch bei *#/partner* reagiert. Beim Aufruf einer Route mit Query-Parametern erhalten Sie ein Objekt mit dem definierten Parameternamen – in unserem Beispiel `?query` –, aus dem Sie die übergebenen URL-Parameter und ihre Werte auslesen können. Nutzen Sie Query-Parameter, wenn Sie mehrere, nicht fest spezifizierte Parameter in einer Route übergeben möchten, die sich je nach Situation unterscheiden können und nicht die direkte Interaktion mit Geschäftsobjekten betreffen.

Rest-as-String-Parameter

Mit einem *Rest-as-String-Parameter* können Sie eine Route mit einem fest definierten und einem beliebigen flexiblen Teil anlegen. Legen Sie hierzu den fest definierten Teil der Route mit den bereits gezeigten hartkodierten oder parametrisierten Patterns an, und ergänzen Sie sie um einen Parameter, der mit dem Zeichen * versehen ist. Dieser repräsentiert einen beliebigen weiteren Hash, den Sie unter dem vergebenen Parameternamen individuell auslesen können.

Konfigurieren Sie eine solche Route beispielsweise mit `/partner/{partnerId}/:all*:`, so erhalten Sie bei einem Aufruf von *#/partner/3/details?filter=none* den zusätzlichen Parameter `all`, aus dem Sie den Hash */details?filter=none* auslesen können.

Stellen Sie sicher, dass der Rest-as-String-Parameter an letzter Stelle steht, wenn Sie mehrere Parameter kombinieren. Nutzen Sie den Rest-as-String-Parameter nur für konkrete Anwendungsfälle, die dies erfordern. Nutzen

Sie ihn nicht zum Abfangen ungültiger Hashes. Hierfür ist das Target `bypassed` vorgesehen, das wir in Abschnitt 11.2.2 gesondert betrachten.

Sie können die hier beschriebenen Parameter grundsätzlich miteinander kombinieren, um die Routen für Ihre Anwendung sinnvoll zusammenzustellen.

Reihenfolge von Routen

Beachten Sie, dass bei der Konfiguration Ihrer Routen deren Reihenfolge im Manifest eine bedeutende Rolle einnimmt. Grundsätzlich wird nur die erste Route, deren Pattern dem aktuellen Hash entspricht, angesteuert, und die dazugehörigen Events werden ausgelöst.

Passen potenziell mehrere Patterns zu bestimmten für Ihre Anwendung vorgesehene Hashes, ordnen Sie die Routen entsprechend Ihrer beabsichtigten Prioritäten an.

Nutzen Sie alternativ den Parameter `greedy` einer Route, um ihre Events auch dann auslösen zu lassen, wenn bereits eine andere Route auf den aktuellen Hash reagiert hat.

Standardroute

Im Parameter `target` konfigurieren Sie alle Targets der Route in einem Array. Sie referenzieren dabei den Schlüssel des jeweiligen Targets. Legen Sie für die Demoimplementierung eine erste Standardroute an (siehe Listing 11.3).

```
"routes": [
  {
    "name": "home",
    "pattern": "",
    "target": [
      "businessPartnerMaster"
    ],
    "layout": "OneColumn"
  }
],
```

Listing 11.3 Standardroute für die Anwendung (Demo)

Vergeben Sie `home` als `name` und als `pattern` einen leeren String. Hierdurch definieren Sie die Standardroute, die beim Start der Anwendung – also bei einem leeren Hash – angesteuert wird. Da im `FlexibleColumnLayout` zuerst nur die Master-Tabelle angezeigt werden soll, definieren Sie ein einzelnes Target `businessPartnerMaster` und geben das Layout `OneColumn` vor. Letzteres ist möglich, da wir die Klasse `sap.f.routing.Router` verwenden.

Targets

Targets konfigurieren

Sie konfigurieren Targets im Abschnitt `sap.ui5.routing.targets` des Manifests. `targets` ist hierbei ein JSON-Objekt, dem Sie pro Target ein weiteres Objekt, identifiziert durch einen eindeutigen Schlüssel, hinzufügen. Fügen Sie in der Demoimplementierung ein erstes Target hinzu (siehe Listing 11.4).

```
"targets": {
  "businessPartnerMaster": {
    "type": View
    "name": "BusinessPartnerMaster"
  }
}
```

Listing 11.4 Target für die Master-Tabelle (Demo)

Zielressource

An dieser Stelle benötigen Sie zunächst nur die Angabe der Zielressource im Parameter `name`. Über den Parameter `type` geben Sie zudem an, welche Art von Zielressource – View oder Komponente – durch das Target geladen wird. In unserem Fall laden wir eine View `BusinessPartnerMaster`, die noch angelegt werden muss.

Alle weiteren Parameter sind in der Standardkonfiguration enthalten, die Sie zuvor definiert haben. Beachten Sie, dass Sie auch dem Parameter `type` einen Standardwert vorgeben könnten. Wir entscheiden uns hier jedoch, den Typ je Target an Ort und Stelle zu kennzeichnen. Bei Abweichungen können Sie je Target u. a. die folgenden Felder konfigurieren:

- `viewType` – gibt den Typ der zu ladenden View (XML, JS, JSON, HTML) an.
- `viewName` – gibt den Namen der zu ladenden View an; kann in Verbindung mit `viewPath` als Alternative zu `name` genutzt werden.
- `viewPath` – gibt den Pfad innerhalb der Anwendung zu der zu ladenden View an. Die Angabe wird als Präfix genutzt und kann in Verbindung mit `viewName` als Alternative zum Parameter `path` genutzt werden. Bevorzugen Sie jedoch, wenn möglich, `name` und `path`.
- `controlId`: ID des Navigationscontainers; weicht in der Regel nicht vom Standardwert ab.
- `controlAggregation` – repräsentiert die Aggregation des Navigationscontainers, in die die Zielressource geladen wird.

Wenn Sie die Klasse `sap.m.routing.Router` nutzen, können Sie außerdem u. a. die folgenden Felder konfigurieren:

- `viewLevel`: Weisen Sie hiermit Targets eine ganzzahlige »Ebene« zu. Hierdurch können Sie verschiedene Views in eine logische Reihenfolge bringen, die dem typischen Nutzungsablauf Ihrer Anwendung entspricht. Bei der Navigation passt sich die Animation an die entsprechende Ebene an: »vorwärts« bei der Navigation auf eine höhere, gleiche oder undefinierte Ebene, »zurück« bei der Navigation auf eine niedrigere Ebene.
- `transition` definiert die Art, auf die Navigation von und zu dem Target animiert wird. Wählen Sie zwischen `slide` (Standard), `baseSlide`, `flip`, `fade` und `show`.

11.1.3 Erste Navigation in der Anwendung

Nachdem Sie sichergestellt haben, dass der Router initialisiert wird, eine Routingkonfiguration im Manifest definiert haben und eine Standardroute und ein dazu passendes Target angelegt haben, ist die Anwendung für die Benutzung des Routers konfiguriert.

Routing in der Demoimplementierung

Für die Demoimplementierung ist jetzt noch eine Umstrukturierung notwendig. Das Target für die Standardroute `home` erwartet eine View. Bislang laden wir die `BusinessPartnerMaster`-Tabelle als Fragment in der View **App.view.xml**, die nun aber nur noch das `FlexibleColumnLayout` enthalten soll.

An dieser Stelle beginnen wir, die Funktionalitäten und Verantwortlichkeiten, die bisher allein durch **App.controller.js** übernommen wurden, aufzutrennen und auf die Controller der neu erstellten View aufzuteilen.

Erstellen Sie zunächst eine neue View mit dem Namen **BusinessPartnerMaster.view.xml** und einen dazugehörigen Controller **BusinessPartnerMaster.controller.js**. Implementieren Sie danach die View (siehe Listing 11.5).

```
<mvc:View controllerName=
"de.sapui5buch.demo.controller.BusinessPartnerMaster"  displayBlock=
"true" xmlns="sap.ui.core" xmlns:mvc="sap.ui.core.mvc" >
  <Fragment
    fragmentName="de.sapui5buch.demo.view.BusinessPartnerTable"
    type="XML"/>
</mvc:View>
```

Listing 11.5 XML-View »BusinessPartnerMaster« (Demo)

Die View lädt nun als Inhalt das Fragment, das zuvor in den `beginColumnPages` des `FlexibleColumnLayout`-Controls in der View **App.view.xml** geladen wurde. Stellen Sie sicher, dass Sie das Fragment dort entfernen. Das `FlexibleColumnLayout` sollte nun keine Inhalte mehr direkt deklarieren, die Inhalte wollen wir ausschließlich durch den Router bereitstellen lassen. Sie

können die Deklaration des `FlexibleColumnLayout`-Controls auf folgenden Einzeiler reduzieren:

```
<f:FlexibleColumnLayout id="masterDetailBase" />
```

Implementieren Sie nun den `BusinessPartnerMaster`-Controller (siehe Listing 11.6).

```
sap.ui.define([
  "de/sapui5buch/demo/controller/BaseController",
  "sap/ui/model/Filter",
  "sap/ui/model/FilterOperator"
], function(Controller, Filter, FilterOperator) {
  "use strict";
  return Controller.extend("de.sapui5buch.demo.controller.
    BusinessPartnerMaster", {
    onInit: function () {},
    onSuggest: function(oEvent) {
      //…
    },
    onSuggestionItemSelected : function(oEvent) {
      //…
    },
    onListItemSelected: function (oEvent) {
      //…
    },
    onValueHelpRequest : function(oEvent) {
      //…
    },
    onSearch: function(oEvent) {
      //…
    },
    onRefreshBusinessPartnerTable: function () {
      //…
    }
  });
});
```

Listing 11.6 »BusinessPartnerMaster«-Controller (Demo)

Stellen Sie sicher, dass die Module `Filter` und `FilterOperator` als Abhängigkeit deklariert werden, und übernehmen Sie die gezeigten Methoden aus **controller.App.controller.js**. Entfernen Sie die Methoden und Abhängigkeiten dort; sie werden nicht mehr benötigt.

Sie sollten nun die Anwendung starten können und wie vorher die Master-Tabelle sehen. Die Suchhilfe und das Aktualisieren bei Mobilgeräten sollten wie gehabt funktionieren.

Um auch die Auswahl einer Detailseite über das Routing abzubinden, sind einige weitere Schritte notwendig, die wir in Abschnitt 11.2 genauer erklären.

11.2 Navigation am Beispiel einer Master-Detail-Anwendung

Bisher haben wir eine Standardroute `home` angelegt, die unsere Master-Tabelle im Vollbild anzeigt. Wir wollen nun über eine weitere Route zu den Objektdetails gelangen. In einem Master-Detail-Szenario übergeben Sie typischerweise den Schlüssel des aus der Master-Liste ausgewählten Objekts an die Detailseite, um ihn für die Datenbindung zu verwenden.

11.2.1 Implementieren der Detailroute

Detailroute

Legen Sie im Manifest eine weitere Route `detail` mit einem Pflichtparameter an (siehe Listing 11.7).

```
"routes": [
  {
    "name": "home",
    "…"
  },
  {
    "name": "detail",
    "pattern": "partner/{partnerId}",
    "target": [
      "businessPartnerMaster",
      "businessPartnerDetail"
    ],
    "layout": "TwoColumnsMidExpanded"
  }
]
```

Listing 11.7 Detailroute im Manifest (Demo)

Wählen Sie als Pattern `partner/{partnerId}`. Über den hartkodierten Teil des Patterns bilden wir die semantische Bedeutung der Route bzw. des Hash ab, während die nachfolgende Partner-ID das ausgewählte Objekt designiert.

Target zur Detailroute

Geben Sie als Targets sowohl das bisherige Target `businessPartnerMaster` als auch das noch zu erstellende Target `businessPartnerDetail` an. Bei der Navigation in einer Vollbildanwendung geben Sie hier nur das Detail-Target an, im Master-Detail-Szenario wollen wir jedoch sowohl die Master-Liste als auch die Detailseite gleichzeitig einsehen können. Legen Sie als Nächstes das Detail-Target an (siehe Listing 11.8).

```
"targets": {
  "businessPartnerMaster": {
    "type": "View",
    "name": "BusinessPartnerMaster"
  },
  "businessPartnerDetail": {
    "type": "View",
    "name": "BusinessPartnerDetail",
    "viewType": "JS",
    "controlAggregation": "midColumnPages"
  }
}
```

Listing 11.8 Detail-Target im Manifest (Demo)

Da Sie die Detail-View als JavaScript-View anlegen müssen, geben Sie `JS` als `viewType` an. Die Detailseite soll in der mittleren Spalte des `FlexibleColumnLayout`-Controls platziert werden; geben Sie daher `midColumnPages` als `controlAggregation` an.

Parameter übergeben

Ändern Sie als Nächstes den Master-Controller **BusinessPartnerMaster.controller.js**, um den notwendigen Übergabeparameter auszulesen und die Navigation bei der Auswahl eines Objekts zu starten. Passen Sie dazu die Funktionen `onSuggestionItemSelected` und `onListItemSelected` an (siehe Listing 11.9).

```
onSuggestionItemSelected : function(oEvent) {
  var oSelectedItem = oEvent.getParameter("selectedItem");
  var sBindingPath = oSelectedItem.getBindingContext().getPath();
  this.getRouter().navTo("detail", {
    partnerId: encodeURIComponent(sBindingPath)
  });
},

onListItemSelected: function (oEvent) {
  var oSelectedItem = oEvent.getParameter("listItem");
  var sBindingPath = oSelectedItem.getBindingContext().getPath();
```



```
  this.getRouter().navTo("detail", {
    partnerId: encodeURIComponent(sBindingPath)
  });
},
```

Listing 11.9 Navigation mit Parameterübergabe (Demo)

Routerinstanz aufrufen

Über die vom `BaseController` bereitgestellte Funktion `getRouter()` wird die Routerinstanz aufgerufen. Über die Funktion `navTo()` steuern Sie gezielt eine Route an. Geben Sie hierfür den Namen der Route als ersten Parameter an. Der zweite Parameter ist ein JavaScript-Objektliteral, in dem die zu übergebenden Parameter enthalten sind. Hier müssen alle Pflichtparameter – in unserem Fall `partnerId` – angegeben werden, ansonsten träte zur Laufzeit ein Fehler auf.

Als Wert wollen wir idealerweise das Feld `BusinessPartnerID` aus unserem Modell übergeben. Da wir jedoch momentan mit einem JSON-Modell arbeiten, in dem die Objekte nicht über einen Schlüssel, sondern nur über ihren Index referenziert sind, ist eine Übergabe des Binding-Paths einfacher, um aus diesem wiederum den Binding-Kontext des ausgewählten Objekts zu erhalten. Über die ID wäre dies nur mit zusätzlichem Aufwand möglich, zum Beispiel über eine Lookup-Funktion.

Den Binding-Pfad übergeben

Die Übergabe des Binding-Pfads erfordert das Kodieren des Pfades, da Sonderzeichen, insbesondere Schrägstriche, nicht als Teil der URL interpretiert werden sollen. Dies geschieht über die Standard-JavaScript-Funktion `encoreURIComponent()`.

Erstellen Sie nun eine View **BusinessPartnerDetail.view.js** und einen Controller **BusinessPartnerDetail.controller.js**. Wir wollen die Vorgehensweise aus Abschnitt 8.3, »Implementierung adaptiver SAPUI5-Anwendungen«, beibehalten und, abhängig von der Geräteart, entweder eine `sap.m.Page` mit `IconTabBar` oder ein `ObjectPageLayout` für die Darstellung verwenden. Das jeweilige Fragment soll beim Laden der View instanziiert und der View hinzugefügt werden. Da das dynamische Hinzufügen von Inhalten direkt zu einer View-Instanz, z. B. mit Hilfe der Funktion `addContent()`, bei XML-Views nicht unterstützt wird, behelfen wir uns, indem wir eine JavaScript-View anlegen (siehe Listing 11.10).

```
sap.ui.jsview("de.sapui5buch.demo.view.BusinessPartnerDetail", {
 getControllerName: function () {
  return "de.sapui5buch.demo.controller.BusinessPartnerDetail";
 },
 createContent: function (oController) {
```

```
  return [];
 }
});
```

Listing 11.10 Detail-View (Demo)

Verweisen Sie, wie bei JavaScript-Views üblich, in der Methode `getControllerName` auf den dazugehörigen Controller. Lassen Sie die Funktion `createContent` ein leeres Array zurückliefern. Die Inhalte der View werden in unseren bisherigen Fragmenten definiert und beim Initialisieren der View geladen. Erstellen Sie nun den Controller (siehe Listing 11.11).

```
sap.ui.define([
  "de/sapui5buch/demo/controller/BaseController",
  "sap/ui/core/Fragment"
], function (Controller, Fragment) {
  "use strict";
  return Controller.extend("de.sapui5buch.demo.controller.
    BusinessPartnerDetail", {

    onInit: function () {
      var sFragmentName;
      var bDeviceIsPhone =
this.getOwnerComponent().getModel("device").getProperty(
  "/system/phone");

      if (bDeviceIsPhone) {                sFragmentName =
        "de.sapui5buch.demo.view.BusinessPartnerDetailsIconTab";
      } else {
        sFragmentName = "de.sapui5buch.demo.view.
           BusinessPartnerDetailsObjectPage";
      }

      Fragment.load({
        id: this.createId("bpd"),
        name: sFragmentName,
        type: "XML",
        controller: this
      }).then(function (oFragment) {
        this.getView().addContent(oFragment);
      }.bind(this));

      this.getRouter().getRoute("detail")
          .attachPatternMatched(this._onRouteMatched, this);
```



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

Listing 11.11 Detail-Controller (Demo)

In der Funktion `onInit()` wird nun, wie es bisher in **controller.App.controller.js** geschah, der aktuelle Gerätetyp bestimmt und abhängig davon der zu ladende Fragmentname ermittelt. Sie können den Code entsprechend aus dem App-Controller übernehmen. Eine Änderung ergibt sich nach dem Laden des Fragments in der Funktion `then()`: Das Fragment wird nun mit `addContent()` der View hinzugefügt, anstatt es direkt in das `FlexibleColumnLayout` einzufügen. Es wird außerdem kein Promise mehr aufgelöst, und es findet auch keine Abfrage mehr zum Lazy Loading statt, da das beim Laden des Targets bereits der Fall ist. Die Funktion vereinfacht sich dadurch spürbar. Die Funktion `_getDetailPage()`im App-Controller wird nun nicht mehr benötigt.

Routingevents

Im letzten Teil der Funktion `onInit()` wird ein Eventhandler für das Event `patternMatched` der `detail`-Route deklariert. Diesen verwenden Sie zum Auslesen der übergebenen Parameter und zum Binden der Objektdaten an die Benutzeroberfläche.

Events bei der Navigation

Bei der Navigation treten verschiedene Events auf. Auf manche dieser Events sollen, je nach Anwendung, Eventhandler reagieren, andere wiederum sollen gegebenenfalls ignoriert oder erst gar nicht ausgelöst werden. Sie erhalten daher hier eine Übersicht der wichtigsten Events.

Eine Route, auf der navigiert wird, kann die Events `matched`, `patternMatched`, `beforeMatched` und `switched` auslösen.

Das `matched`-Event einer Route wird ausgelöst, wenn entweder das Pattern der Route selbst oder das Pattern einer verschachtelten Route mit dem aktuellen Hash übereinstimmt. Somit werden beim Ansteuern von untergeordneten Routen per `attachMatched` auch die Handler-Funktionen der übergeordneten Routen aufgerufen.

Vermeiden Sie dies, indem Sie das Event `patternMatched` verwenden. Es wird nur bei der Übereinstimmung des Patterns der eigenen Route ausgelöst und ist in der Regel ausreichend.

Das Event `beforeMatched` entspricht dem `matched`-Event, es wird jedoch ausgelöst, bevor die dazugehörigen Targets geladen werden.

Ein internes `switched`-Event wird zudem ausgelöst, wenn eine andere Route als die bisherige angesteuert wurde.

Zu den Events, die vom Router selbst ausgelöst werden, gehören `routeMatched`, `routePatternMatched` und `bypassed`. Die Events `routeMatched` und `routePatternMatched` werden für jede Route ausgelöst werden. Die angesteuerte Route ist Teil der Eventparameter und kann entsprechend ausgelesen werden. Ziehen Sie im Sinne des möglichst spezifischen Eventhandlings die Routenevents den Routerevents vor.

Das Event `bypassed` wird ausgelöst, wenn ein Hash keiner Route zugeordnet werden kann. Der fehlerhafte Hash ist wiederum als Parameter enthalten. Nutzen Sie dieses Event, um ungültige Routen abzufangen und Ihren Anwendern eine sinnvolle Fehlerseite zu präsentieren.

Das Event `display` eines geladenen Targets wird ausgelöst, wenn ein Target über das Ansteuern einer Route geladen wird oder wenn es direkt über seine Funktion `display()` angezeigt wird. Auch hier können Sie u. a. Informationen über die geladene View, das enthaltene Control und die übergebenen Daten auslesen.

Für weitere Detailangaben und Updates konsultieren Sie die Routingsektion der API-Dokumentation, insbesondere die Referenzen zu Routen und Targets.

Parameter auslesen

Implementieren Sie als Nächstes die `_onRouteMatched()`-Funktion (siehe Listing 11.12).

```
_onRouteMatched: function (oEvent) {
  var oArgs = oEvent.getParameter("arguments");
  var sPartnerId = oArgs.partnerId;
  var sBindingPath = decodeURIComponent(sPartnerId);

  this.getView().bindElement(sBindingPath);
  var oAdrSection =
 Fragment.byId(this.createId("bpd"), "addressSection");
  oAdrSection.setVisible(true);
  oAdrSection.bindElement(sBindingPath + "/Address");
}
```

Listing 11.12 Eventhandler für »patternMatched« im Detail-Controller (Demo)

Im Parameter `arguments` des `patternMatched`-Events werden die übergebenen Parameter hinterlegt. Der übergebene Binding-Pfad wird daraus ausgelesen und mit `decodeURIComponent()` wieder dekodiert. Der Binding-Kontext, der durch diesen Pfad adressiert wird, wird per Element-Binding an die

View und der im Binding-Kontext enthaltene `Address`-Bereich nochmals gesondert an die entsprechende Sektion gebunden.

Schließen der Detailsicht

Abschließend müssen auch das Schließen der Detailansicht und die Rückkehr zur Master-Tabelle navigatorisch umgesetzt werden. Implementieren Sie im Controller **BusinessPartnerDetail.controller.js** die Funktion `onCloseDetail()`, indem Sie die `home`-Route aufrufen:

```
onCloseDetail: function () {
  this.getRouter().navTo("home");
}
```

Definieren Sie dazu passend im Controller **BusinessPartnerMaster.controller.js** eine Eventhandler-Funktion, und weisen Sie sie dem `patternMatched`-Event der `home`-Route zu (siehe Listing 11.13).

```
onInit: function () {
  this.getRouter().getRoute("home")
      .attachPatternMatched(this._onObjectMatched, this);
},
// …
_onRouteMatched: function (oEvent) {
  this.byId("idInputField").setValue();
  this.byId("businessPartnerTable").removeSelections(true);
}
```

Listing 11.13 Eventhandler für »patternMatched« im Master-Controller (Demo)

Die Master-Ansicht zurücksetzen

Im Eventhandler zum Event `patternMatched` werden die durchzuführenden Befehle zum Zurücksetzen der Master-Ansicht – das Löschen eventueller Eingaben im Suchfeld und das Aufheben der Selektion in der Master-Tabelle – implementiert. Damit ist nun die gesamte Funktionalität aus dem App-Controller in den Master- bzw. Detail-Controller verschoben. Die eventuell noch im App-Controller vorhandenen Funktionen werden nicht mehr benötigt.

Die Anwendung ist nun in ihrem bisherigen Stand vollständig auf das SAPUI5-Navigationskonzept umgestellt. Sie können wie gewohnt Partner auswählen, ihre Details betrachten und die Detailansicht wieder schließen. Zudem erscheint der übergebene Binding-Pfad, den wir in Kapitel 12 durch die Partner-ID ersetzen werden, im Hash der Browser-URL (siehe Abbildung 11.4).

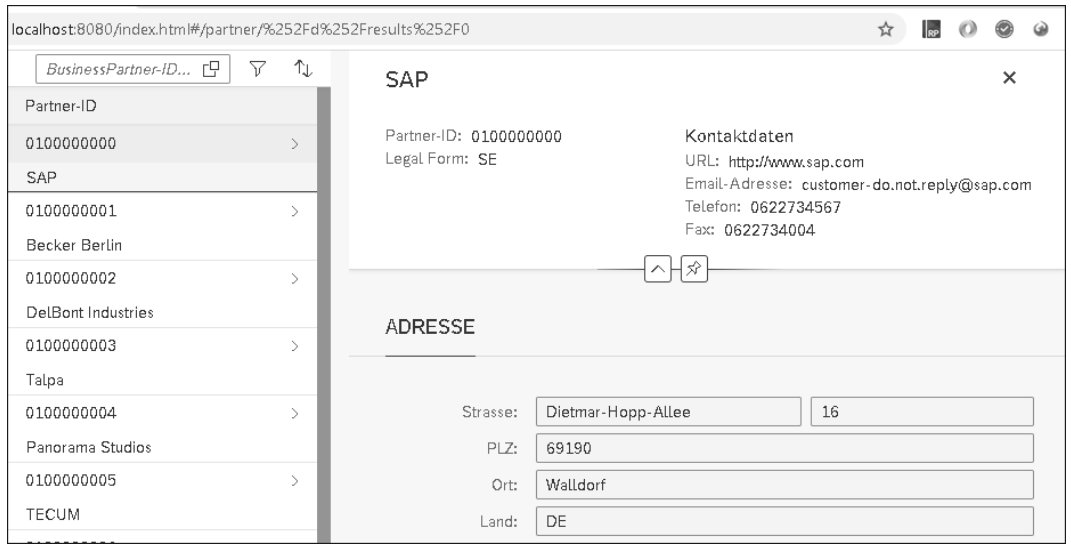

Abbildung 11.4 Anwendung mit Navigation und parametrisiertem Hash (Demo)

Auf diese Art können Ihre Anwender nun auch Detailseiten direkt verlinken und beispielsweise als Lesezeichen sichern.

11.2.2 Ungültige Routen abfangen

Bei der Navigation innerhalb einer Anwendung kann es dazu kommen, dass eine Route aufgerufen wird, zu der kein passendes Target gefunden werden kann. Dies kann beispielsweise bei der hashbasierten Navigation auftreten, wenn in der URL ein fehlerhafter Hash verwendet wird. Auch bei der Navigation zu einem Zielobjekt ist es denkbar, dass zu einem übergebenen Schlüsselparameter kein passendes Objekt gefunden werden kann.

Ungültige Routen abfangen mit »bypassed«

Um Ihre Anwender in diesen Szenarien passend darüber zu informieren und ihnen eine Möglichkeit zur Weiter- bzw. Rücknavigation zu bieten, blenden Sie eine entsprechende Informationsseite als Target ein. Im Falle eines fehlerhaften Hash nutzen Sie das Event des Routers `bypassed`, um zu dieser Informationsseite zu navigieren. Bei einem nicht vorhandenen Objekt in einem Master-Detail-Szenario blenden Sie die Informationsseite anstatt der Objektdetails ein.

Nichtexistente Routen

»bypassed«-Targets

Versuchen Sie, im derzeitigen Stand der Demoimplementierung zu einem Hash zu navigieren, zu dem keine Route existiert, z. B. *#/partners*, so hat dies keinen Effekt; es findet schlicht keine Navigation statt.

Für Ihre Anwender ist dies natürlich keine optimale Situation. Der Versuch, in der Anwendung zu navigieren, sollte in jedem Fall eine Reaktion nach sich ziehen, wenn auch nur eine rein informative.

Um in diesem Fall auf das `bypassed`-Event, also das Fehlen einer passenden Route zum aktuellen Hash, zu reagieren, müssen Sie zunächst keine explizite Handler-Funktion implementieren. Stattdessen können Sie bereits im Manifest deklarativ ein `bypassed`-Target anlegen. Fügen Sie dazu der Routerkonfiguration in **manifest.json** ein `notFound`-Target hinzu (siehe Listing 11.14).

```
"routing": {
  "config": {
    "routerClass": "sap.f.routing.Router",
    "…",
    "controlAggregation": "beginColumnPages",
    "bypassed": {
      "target": "notFound"
    },
    "async": true
  },
"…",
  "targets": {
    "…",
    "notFound": {
      "type": "View",
      "name": "NotFound"
    }
  }
}
```

Listing 11.14 Einfügen eines »bypassed«-Targets im Manifest (Demo)

Die Anwendung wird nun automatisch die View `NotFound` anzeigen, wenn versucht wird, zu einer nichtexistenten Route zu navigieren.

»NotFound«

Erstellen Sie die View **NotFound.view.xml**, und nutzen Sie für ihre inhaltliche Gestaltung ein `sap.m.MessagePage`-Control (siehe Listing 11.15).

```
<mvc:View
  controllerName="de.sapui5buch.demo.controller.NotFound"
  xmlns="sap.m" xmlns:mvc="sap.ui.core.mvc">
  <MessagePage
    title="Nicht verfügbar"
    text="Die gewünschte Ressource konnte nicht geladen werden"
    icon="sap-icon://document"
```

```
      showNavButton="true"
      navButtonPress=".onNavBack">
    </MessagePage>
</mvc:View>
```

Listing 11.15 »NotFound«-View zum Abfangen fehlerhafter Routen (Demo)

Für die `NotFound`-View ist es wichtig, den Anwendern eine Möglichkeit zu bieten, von der Informationsseite wieder zurückzunavigieren. Typischerweise wird hierfür die Standard-Rücknavigation genutzt, wie sie von der Funktion `onNavBack()` des `BaseController`-Elements geboten wird.

Einrichtung der »NotFound«-View

In unserem Master-Detail-Szenario mit `FlexibleColumnLayout` müssen wir außerdem sicherstellen, dass die `NotFound`-View als Vollbildseite angezeigt wird. Das Target `notFound` lädt die View in die Aggregation `beginColumnPages`. Rufen wir also von einer Master-Detail-Anzeige aus eine fehlerhafte Route auf, so würde die View `NotFound` momentan anstatt der Master-Tabelle auf der linken Seite angezeigt, die Detailansicht des zuletzt angezeigten Objekts bliebe aber bestehen.

Um eine solche Situation zu vermeiden, stellen Sie sicher, dass das `layout` des `FlexibleColumnLayout`-Controls bei der Anzeige des `notFound`-Targets wieder auf `OneColumn` umgestellt wird.

[+]

Lokale Modelle zur Verwaltung von View-Zuständen

Für Fälle, in denen einzelne Eigenschaften bestimmter Controls gesetzt und ausgelesen werden müssen, bieten sich lokale Modelle – in der Regel JSON-Modelle – an, um den Wert dieser Controls zu hinterlegen und für Akteure außerhalb des betreffenden Controls zugänglich zu machen.

Die Eigenschaft der Controls wird dann per Property-Binding gegen dieses lokale Modell gebunden. Kapitel 12, »Arbeiten mit Modellen«, geht im Detail auf diese Möglichkeiten ein.

Da der Controller `NotFound` keinen direkten API-Zugriff zum `FlexibleColumnLayout` hat, behelfen wir uns mit einem lokalen JSON-Modell im App-Controller, das durch die Modellvererbung auch im `NotFound`-Controller verfügbar ist. Implementieren Sie den Controller **NotFound.controller.js** (siehe Listing 11.16).

```
sap.ui.define([
  "de/sapui5buch/demo/controller/BaseController"
], function (Controller) {
```



```
  "use strict";
return Controller.extend("de.sapui5buch.demo.controller.NotFound", {
    onInit: function () {
      this.getRouter().getTarget("notFound")
          .attachDisplay(this._onNotFoundDisplayed, this);
    },
    _onNotFoundDisplayed : function () {
      this.getModel("appView")
          .setProperty("/layout", "OneColumn");
    }
  });
});
```

Listing 11.16 Der Controller »NotFound« mit Layoutanpassung (Demo)

Lokales View-Modell

Die Handler-Funktion für das Aufrufen des Targets wird hier über das Event des Targets `display` zugewiesen. Somit stellen Sie sicher, dass die jeweiligen Befehle umgesetzt werden, unabhängig davon, ob das `notFound`-Target über seine `display()`-Funktion oder über den Router geladen wurde.

In der Eventhandler-Funktion wird der Parameter `/layout` in einem JSON-Modell `appView`, das im App-Controller gesetzt wird, auf den gewünschten Wert `OneColumn` gesetzt. Das entsprechende JSON-Modell müssen wir nun noch erstellen und die Eigenschaft `layout` des `FlexibleColumnLayout`-Controls dagegen binden. Ergänzen Sie die View **App.controller.js** daher entsprechend (siehe Listing 11.17).

```
sap.ui.define([
  "sap/ui/core/mvc/Controller",
  "sap/m/MessageBox",
  "sap/ui/model/json/JSONModel"
], function(Controller, MessageBox, JSONModel) {
  "use strict";
  return Controller.extend("de.sapui5buch.demo.controller.App",  {
    onInit: function () {
      this.getView().setModel(new JSONModel(), "appView");
    },
    // …
  });
});
```

Listing 11.17 Setzen eines lokalen JSON-Modells im App-Controller (Demo)

Passen Sie danach die View **App.view.xml** an:

```
<f:FlexibleColumnLayout id="masterDetailBase"
   layout="{appView>/layout}" />
```

Die View `NotFound` wird nun aus jeder Anwendungssituation heraus im Vollbild angezeigt (siehe Abbildung 11.5).

Abbildung 11.5 Ausschnitt aus der »NotFound«-View (Demo)

Ungültige Detailobjektschlüssel

»NotFound«-View für Detailobjekte

Bisher haben wir ungültige Hashes – also solche, zu denen keine Route in unserer Anwendung existiert – abgedeckt. In einem Master-Detail-Szenario kann es aber auch vorkommen, dass die Navigation zu einem Detailobjekt fehlschlägt, das heißt, die Route an sich ist gültig, es kann aber zum übergebenen Objektschlüssel kein passendes Detailobjekt geladen werden. Dies spielt insbesondere bei Deep Links eine Rolle, bei denen gegebenenfalls eine URL gespeichert ist, deren Objekt mittlerweile nicht mehr existiert.

In solchen Fällen wollen wir eine ähnliche Informationsseite anzeigen, hier aber anstelle des Detailobjekts. Sie setzen dies mit einer separaten `DetailObjectNotFound`-View und dem dazugehörigen Target um, da sich die Funktionalität von der bisherigen `NotFound`-View unterscheidet. Legen Sie zunächst eine View **DetailObjectNotFound.view.xml** an (siehe Listing 11.18).

```
<mvc:View
  controllerName="de.sapui5buch.demo.controller.DetailObjectNotFound"
  xmlns="sap.m" xmlns:mvc="sap.ui.core.mvc">
  <MessagePage
    title="BusinessPartner nicht gefunden"
    text="Das ausgewählte Objekt konnte nicht geladen werden"
    icon="sap-icon://product"
    description=""
    showNavButton="{= ${device>/system/phone} ||
```



```
      ${device>/system/tablet} &&
      ${device>/orientation/portrait}
    }"
    navButtonPress=".onNavBack">
  </MessagePage>
</mvc:View>
```

Listing 11.18 »DetailObjectNotFound«-View für Detailobjekte (Demo)

»DetailObjectNotFound«

Die View ist fast identisch mit `NotFound`. Da die View `DetailObjectNotFound` aber als Detailseite dargestellt werden soll, muss sie auf ausreichend großen Geräten keinen Button für die Zurücknavigation enthalten. Das Attribut `showNavButton` der `MessagePage` ist daher mit einem Expression-Binding belegt, das den Button nur auf Mobiltelefonen und Tablets im Hochformat anzeigt.

Erstellen Sie zur View einen leeren Controller, der von unserem `BaseController` erbt (siehe Listing 11.19). Es ist hier keine eigene Funktionalität notwendig.

```
sap.ui.define([
  "de/sapui5buch/demo/controller/BaseController"
], function (Controller) {
  "use strict";
return Controller.extend("de.sapui5buch.demo.controller.
  DetailObjectNotFound", {});
});
```

Listing 11.19 Leerer Controller für »DetailObjectNotFound« (Demo)

Ergänzen Sie danach das Target `detailObjectNotFound` in **manifest.json** (siehe Listing 11.20).

```
"detailObjectNotFound": {
  "type": "View",
  "name": "DetailObjectNotFound",
  "controlAggregation": "midColumnPages"
}
```

Listing 11.20 Target »detailObjectNotFound« im Manifest (Demo)

Dabei ist es wichtig, dass `midColumnPages` als `controlAggregation` gesetzt wird. Das Target soll in Fällen angezeigt werden, in denen kein gültiges Detailobjekt geladen werden kann. Passen Sie dazu den Controller `BusinessPartnerDetail` an (siehe Listing 11.21).

```
_onRouteMatched: function (oEvent) {
  var oArgs = oEvent.getParameter("arguments");
  var sPartnerId = oArgs.partnerId;
  var sBindingPath = decodeURIComponent(sPartnerId);

  this.getView().bindElement({
    path: sBindingPath,
    events: {
      change: this._onBindingChange.bind(this)
    }
  });
},

_onBindingChange: function () {
  var oView = this.getView();
  var oElementBinding = oView.getElementBinding();
  var oContext = oElementBinding.getBoundContext();

  // No data for the binding
  if (!oContext) {
    this.getRouter().getTargets()
        .display("detailObjectNotFound");
    return;
  }
  var oAdrSection = Fragment.byId(this.createId("bpd"),
      "addressSection");
  oAdrSection.setVisible(true);
  oAdrSection.bindElement(oContext.getPath() + "/Address");
},
```

Listing 11.21 Objektprüfung beim Setzen eines Detailobjekts (Demo)

In der Funktion `bindElement()` wird nun zusätzlich zum Binding-Pfad eine Handler-Funktion für das `change`-Event des durch die Methode erstellten `sap.ui.model.ContextBinding` zugewiesen.

In dieser Handler-Funktion wird anhand des gebundenen `Context`-Objekts geprüft, ob das Binding erfolgreich war. Konnte kein Detailobjekt zum übergebenen Pfad gebunden werden, ist der `Context` undefiniert, und das Target `detailObjectNotFound` wird geladen.

Das weitere Binding der Adresssektion ist nur dann sinnvoll, wenn ein Detailobjekt gebunden werden konnte. Die dazugehörigen Zeilen wurden daher aus der Funktion `_onRouteMatched()` in die Funktion `_onBindingChange()`

verschoben. Nun wird auch bei einer Route, die zwar gültig ist, aber das Objekt nicht laden kann, eine passende Informationsseite gezeigt (siehe Abbildung 11.6).

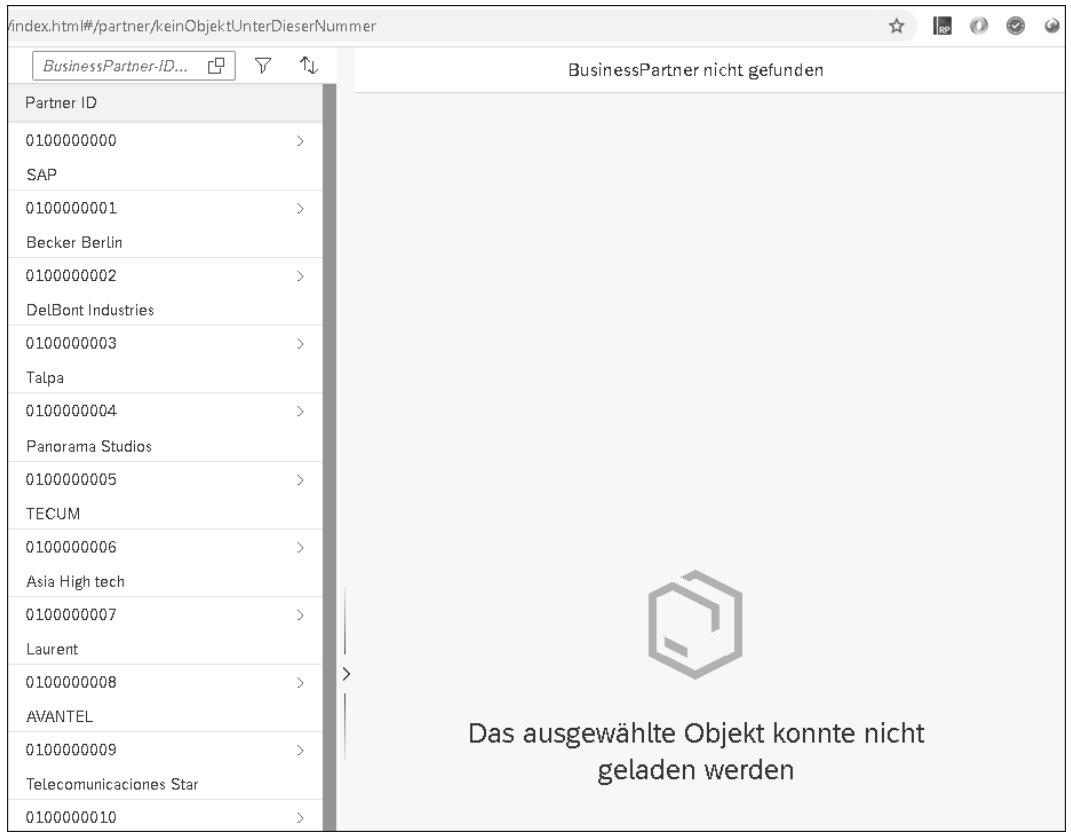

Abbildung 11.6 »DetailObjectNotFound«-View in der Anwendung (Demo)

11.3 Erweiterte Routingkonzepte

In diesem Abschnitt betrachten wir abschließend das Ansteuern verschachtelter Views innerhalb einer Route sowie die Navigation mit verschachtelten Komponenten, die es Ihnen erlauben, Ihre Anwendungen in einer modularen Architektur und einem entsprechend komponentenbasierten Ansatz zu strukturieren.

11.3.1 Target-Kopplung für verschachtelte Views

Wie Sie bisher gesehen haben, können mehrere Targets in einer Route referenziert werden, um bei der Navigation zu dieser Route geladen und ange-

zeigt zu werden. Die so referenzierten Targets sind nur lose gekoppelt, ihre einzige Verbindung besteht über die Route. Darüber hinaus können sie jedoch auch einzeln verwendet werden.

Beim Ansteuern der Route werden die Targets in der Reihenfolge ihrer Deklaration asynchron geladen. Nutzen Sie das Referenzieren mehrerer Targets für Situationen, in denen Sie die Targets entweder unabhängig voneinander – etwa in einem Master-Detail-Szenario – oder nur für manche Routen verwenden wollen.

Target-Beziehungen

Für Situationen, in denen mehrere Views einen festen Verbund bilden, ist eine engere Kopplung zwischen den beteiligten Targets empfehlenswert. Nutzen Sie diese Option beispielsweise für Views, die als Rahmen für weitere, darin verschachtelte Views dienen. Die eingebettete View wird also durch den Router in eine Aggregation der äußeren View geladen. In diesem Fall ist es notwendig, den Rahmen vor der darin eingebetteten View zu laden. Sie erreichen dies über eine Parent-Beziehung, die Sie in der Routingkonfiguration definieren. Das Target, das die Rahmen-View lädt, dient dabei als Eltern-Target, das Target, das die eingebettete View lädt, wiederum als Kind-Target. Das Kind-Target verweist dabei über den Parameter `parent` auf das Eltern-Target (siehe Listing 11.22).

```
"routes": [{
    "pattern": "…",
    "name": "…",
    "target": [ "einzubettendesTarget" ]
  }],
"targets": {
    "rahmenTarget": {
      "type": "View",
      "name": "Rahmen",
      "controlId": "app",
      "controlAggregation": "pages"
    },
    "einzubettendesTarget": {
      "parent": "rahmenTarget",
      "type": "View",
      "name": "EinzubettendeView",
      "controlId": "rahmenControl",
      "controlAggregation": "content"
    },
}
```

Listing 11.22 Koppeln von Targets in der Routingkonfiguration

Bei der Integration in eine Route muss nur das einzubettende Target durch die Route referenziert werden. Beim Ansteuern dieser Route wird sichergestellt, dass das Rahmen-Target vor dem eingebetteten Target geladen wird (siehe Abbildung 11.7).

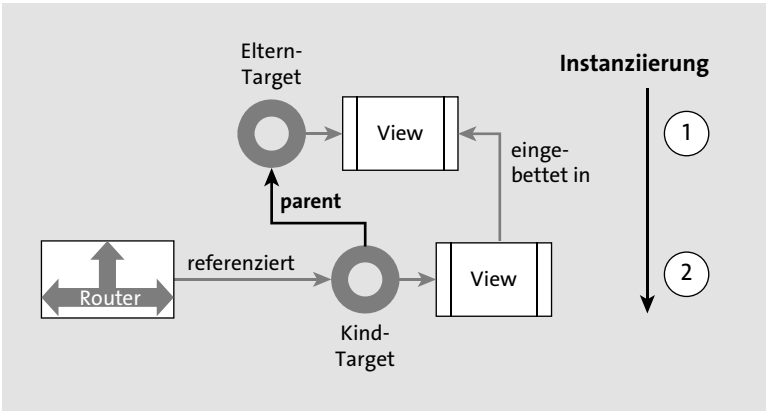

Abbildung 11.7 Enge Kopplung von Targets über den »parent«-Parameter

11.3.2 Routing mit verschachtelten Komponenten

Routen zu Komponenten

Mit zunehmendem Umfang und entsprechender Strukturierung Ihrer Anwendungen werden Sie verschiedene funktionale Komponenten wiederverwendbar anlegen und diese in unterschiedliche Anwendungen integrieren wollen.

Mehrere dieser Komponenten werden hierbei auch alleinstehend verwendet und weisen dafür einen eigenen Router auf. Um das Routing beim Einbetten einer solchen Komponente konsistent beizubehalten, konfigurieren Sie verschachtelte Routen in der einbettenden Komponente. Die notwendigen Schritte hierfür sind:

- Registrierung der eingebetteten Komponente im Manifest (`usage`)
- Anlegen eines Targets, das auf die eingebettete Komponente verweist
- Anlegen einer Route und Zuweisen eines Präfix für die Schachtelung

Bei der gemeinsamen Verwendung mehrerer Komponenten mit jeweils eigenen Routern reagieren alle Router potenziell auf Änderungen im Hash. Bei der Verschachtelung muss daher sichergestellt werden, dass eine Unterscheidungsmöglichkeit gegeben ist, anhand derer die Router »ihren« Teil des Hash kennen und so nur auf für sie relevanten Änderungen reagieren können. Das Präfix, das Sie der verschachtelten Route zuweisen, sorgt für diese Unterscheidung. Es dient als Trennzeichen zwischen der übergeord-

neten und der verschachtelten Komponente und ist in der Form *&/präfix* auch Teil der URL. Das Trennzeichen stellt für die verschachtelte Komponente den Beginn ihres Hash dar, so wie das Rautenzeichen # den Beginn des Hash für die übergeordnete Komponente darstellt (oder wie es für die eingebettete Komponente der Fall wäre, wenn sie alleinstehend betrieben würde). Abbildung 11.8 illustriert die verschiedenen Aspekte des Routings zu einer verschachtelten Komponente und deren Zusammenhänge.

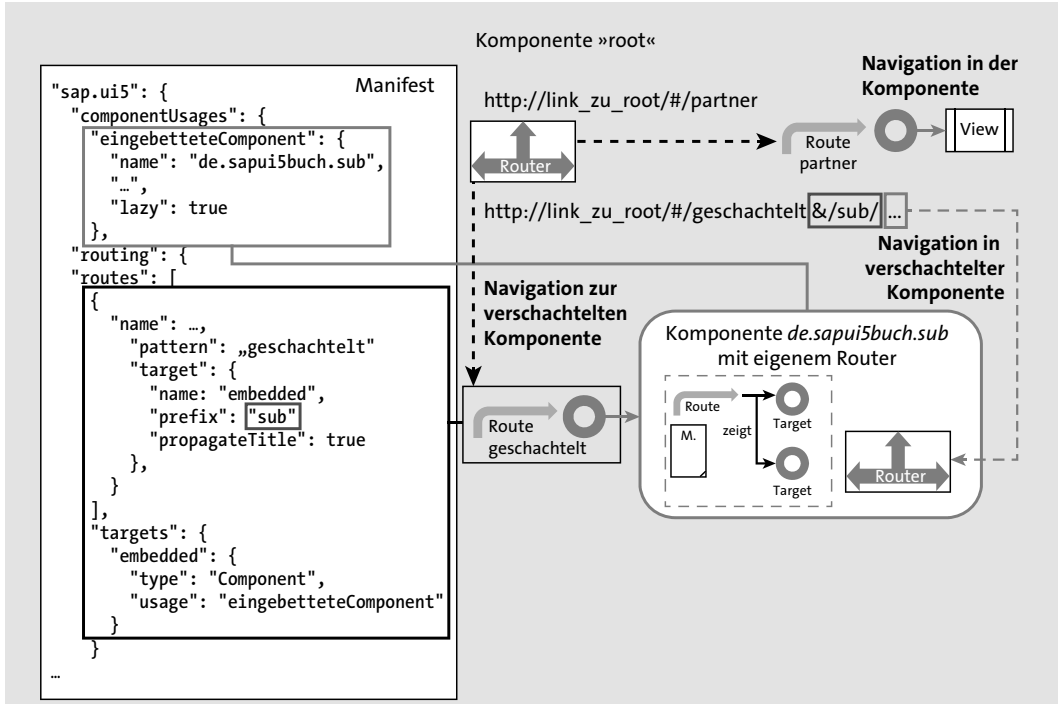

Abbildung 11.8 Navigation mit verschachtelten Komponenten

Komponenten für die Navigation registrieren

Um eine eingebettete Komponente für die Navigation zu registrieren, stellen Sie zunächst sicher, dass die Komponente im Manifest in den `componentUsages` registriert ist. Sie tragen hier den Komponentennamen ein sowie eventuelle Eigenschaften über die Parameter `settings` oder `componentData` und geben an, ob die Komponente `lazy` oder bereits vorab geladen werden sollte.

Legen Sie dann in der Routingkonfiguration ein Target vom Typ `Component` an, und verweisen Sie mit dem Parameter `usage` auf die registrierte Komponente. Sie verwenden dabei den unter `componentUsages` verwendeten Schlüssel. In der Target-Konfiguration haben Sie außerdem eine weitere Möglichkeit, mit dem Parameter `options` zusätzliche Einstellungen zu den in

`componentUsages` eingestellten Optionen zu definieren. Über den Parameter `containerOptions` können Sie außerdem Einstellungen für den `ComponentContainer` angeben, in den die Zielkomponente integriert wird.

Legen Sie wie gehabt eine Route mit einem Namen und einem Pattern an, zu der Sie in der einbettenden Anwendung navigieren können. Als `Target` für die Route definieren Sie ein Objekt, in dem Sie im Parameter `name` den Namen des Targets setzen. In einem weiteren Parameter `prefix` setzen Sie das Präfix, das von der eingebetteten Komponente als Trennzeichen verwendet werden soll.

In der Anwendung können Sie nun die Route ansteuern und so die eingebettete Komponente laden. In der Komponente selbst können Sie gemäß der dortigen Routingkonfiguration navigieren.

Verschachtelte Routen direkt ansteuern

Wenn Sie eine verschachtelte Komponente per Navigation ansteuern, die wiederum eine eigene Routingkonfiguration besitzt, ist es oft sinnvoll, direkt eine andere als die Standardroute aufzurufen. Dies können Sie beim Aufruf der Funktion `navTo()` aus der äußeren Komponente heraus über den Parameter `componentTargetInfo` realisieren. In diesem Parameter machen Sie Angaben zur Navigation innerhalb der angesteuerten Komponente.

Als Beispiel könnten Sie unsere bisher erstellte Demoanwendung als Komponente in einer weiteren Anwendung einbetten. Die Routingkonfiguration in dieser weiteren Anwendung sieht eine Route `businessPartners` vor, deren gleichnamiges Target `BusinessPartners` vom Typ `Component` auf die im Manifest registrierte componentUsage für unsere Demoanwendung verweist. Wollen Sie nun die eingebettete Demoanwendung ansteuern und darin direkt auf eine Detailseite navigieren, so geben Sie die Route `detail` mit dem notwendigen Parameter `partnerId` als `componentTargetInfo` beim Aufruf der Funktion `navTo()` mit an (siehe Listing 11.23).

```
this.getRouter().navTo("businessPartners", {}, {
  BusinessPartners: {
    route: "detail",
    parameters: {
      partnerId: //partnerId…
    }
  }
});
```

Listing 11.23 Weitergabe von Navigationsinformationen an eine verschachtelte Komponente

Das Objekt `componentTargetInfo` wird der Funktion `navTo()` als dritter Parameter übergeben. Der zweite Parameter, das `parameters` Objekt für die Route `businessPartners`, bleibt hier leer, da die Route keine eigenen Parameter aufweist.

Konfigurationsobjekte für Targets angeben

Im Objekt `componentTargetInfo` können Sie für jedes Target, das durch die Route angesteuert wird, ein entsprechendes Konfigurationsobjekt angeben, indem Sie als Objektschlüssel den Namen des Targets verwenden. In unserem Beispiel sind dies nur die Angaben für das Target `BusinessPartners`. In einem solchen Konfigurationsobjekt geben Sie wiederum die gleichen Parameter an, die beim Aufruf der Funktion `navTo()` zum Einsatz kommen: `route`, `parameters` und `componentTargetInfo`. Auf diese Weise können Sie auch Navigationsangaben für tiefer verschachtelte Komponenten weitergeben.

Das Definieren von Komponenten-Targets und die Weitergabe von Navigationsinformationen bietet Ihnen eine große Flexibilität beim Einsatz von Komponenten mit eigenem Routing.