SAP PRESS

# Clean SAPUI5

## Lesbarer und wartbarer SAPUI5-Code

- Clean Code für SAPUI5-Programme
- Praktische Beispiele für JavaScript, Module, Funktionen, Variablen und mehr
- Namenskonventionen, Kommentare und Formatierung

Bertolozi · Buchholz · Häuptle
Jordão · Lehmann · Vaithianathan

Rheinwerk Publishing

# Kapitel 3
# Projektstruktur

*In diesem Kapitel erläutern wir, wie verschiedene Arten von SAPUI5-Projekten auf eine Clean-SAPUI5-Art strukturiert werden können. Außerdem führen wir Sie durch einige wichtige Artefakte in SAPUI5-Projekten.*

Clean SAPUI5 kann auf zahlreiche Arten von SAPUI5-Projekten angewendet werden. In diesem Kapitel besprechen wir die Bausteine von SAPUI5 – Komponenten – und erkunden die wichtigen Artefakte, die an Projekten beteiligt sind. Anschließend besprechen wir wichtige Arten von Projekten, einschließlich Freestyle-Anwendungen, SAP-Fiori-Elements-Projekte und Bibliotheksprojekte. Abschließend betrachten wir, wie das Model-View-Controller-Pattern (MVC-Pattern) in SAPUI5-Projekten implementiert wird.

## 3.1 Komponenten in SAPUI5

Komponenten sind unabhängige und wiederverwendbare Teile einer Anwendung. In SAPUI5 gibt es die folgenden beiden Arten von Komponenten:

- **Oberflächenlose Komponenten**
  Diese Art von Komponenten erweitert die Klasse `sap.ui.core.Component`, die die Basisklasse ist und die Metadaten für die Komponente bereitstellt. Wie der Name vermuten lässt, verfügen diese Komponenten über keine Benutzeroberfläche (engl. User Interface, kurz UI) und werden verwendet, um eine Art Orchestrierung innerhalb der Anwendung durchzuführen. Ein Beispiel für eine oberflächenlose Komponente ist etwa eine Komponente, die nur mit einem Backend-System interagiert, um Daten abzurufen und eine Nachbearbeitungslogik anzuwenden.
- **UI-Komponenten**
  Diese Art von Komponente erweitert die Klasse `sap.ui.core.UIComponent`, die wiederum die Klasse `sap.ui.core.Component` erweitert und der Komponente Rendering-Funktionen hinzufügt. Diese Komponenten stellen einen Bereich auf dem Bildschirm oder ein benutzerdefiniertes UI-Element auf der Benutzeroberfläche dar, z. B. einen Button, zusammen mit den entsprechenden Einstellungen und Metadaten.

Anwendungen können mehrere Komponenten aus derselben Quelle oder aus verschiedenen Quellen verwenden. Komponenten können die Kapselung durch Design unterstützen, was das Verständnis und die Wartung von Komponenten erleichtert. Komponenten werden über die Component-Factory-Funktion `create` der Klasse `sap.ui.core.Component` geladen und angelegt, wie in Listing 3.1 gezeigt, wodurch die oberflächenlose Komponente `sap.cleanui5.demo.watermark` angelegt wird.

```
sap.ui.define([
  'sap/ui/core/Component'
], function (Component) {
  'use strict';

  return Component.create({
    name: 'sap.cleanui5.demo.watermark'
  });
});
```

**Listing 3.1** Oberflächenlose Komponente, die mit der Component-Factory-Funktion angelegt wurde

Standardmäßig wird die Deskriptordatei (normalerweise **manifest.json** genannt) geladen, bevor die Komponenteninstanz angelegt wird. Dadurch können Sie die erforderlichen Abhängigkeiten, einschließlich der Bibliotheken, der abhängigen Komponenten und Modelle, vorab laden. Dadurch wird der erste Aufruf der Anwendung performanter, indem das Vorabladen von Ressourcen optimiert und parallelisiert wird. Wie Sie in Listing 3.2 sehen, wird die UI-Komponente `sap.cleanui5.demo.watermark` angelegt und lädt den Standarddeskriptor **manifest.json**, der sich im Ordner **/root** des Verzeichnisses befindet. Alternativ kann ein Pfad für die Deskriptordatei angegeben werden. Nach dem Laden des Deskriptors lädt die Component-Factory die im Deskriptor definierten Abhängigkeiten parallel zum Component-Preload.

```
sap.ui.define([
  'sap/ui/core/UIComponent'
], function (UIComponent) {
  'use strict';

  return UIComponent.extend('sap.cleanui5.demo.watermark', {
    metadata: {
      manifest: 'json'
    }
  });
});
```

**Listing 3.2** UI-Komponente mit Standard-Deskriptordatei

Sehen wir uns nun an, wie die verschiedenen Dateien in einem Projekt platziert werden müssen. Komponenten sind in eindeutigen Namensräumen organisiert, und der Namensraum einer Komponente entspricht ihrem Komponentennamen. Typischerweise besteht eine Komponente aus einer Component-Controller-Datei (**Component.js**) und einer Deskriptordatei (**manifest.json**). Obwohl nur der Component Controller obligatorisch ist, empfehlen wir, die Deskriptordatei für eine optimale Performance beim initialen Laden der Anwendung zu verwenden. Sowohl die erforderlichen als auch die optionalen Ressourcen einer Komponente müssen im Namensraum der Komponente organisiert sein. Abbildung 3.1 zeigt eine Beispielprojektstruktur für eine UI-Komponente mit einem View und ihrem Controller.


**Abbildung 3.1** UI-Komponente mit View und Controller

Wir empfehlen Ihnen, Artefakte nach Semantik zu gruppieren, anstatt nach Typ. Wie in Abbildung 3.2 dargestellt, sind alle Ihre Controller, wenn die Artefakte beispielsweise nach Typ gruppiert sind, Teil des Ordners **/controller**.

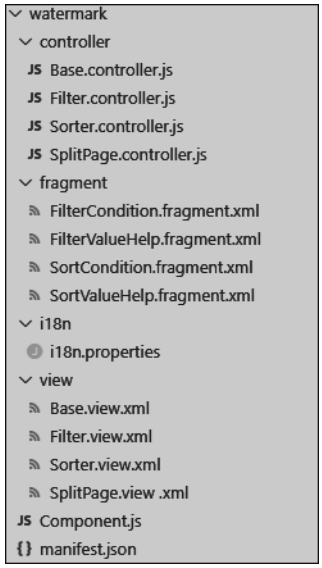


**Abbildung 3.2** UI-Komponente mit Artefakten gruppiert nach Typ

Wenn Sie an der Einführung neuer Funktionen arbeiten oder die vorhandenen Funktionen ändern, arbeiten Sie wahrscheinlich eng mit einer Reihe von Artefakten zusammen. Verwandte Artefakte nah beieinander zu halten, anstatt sie nach Typ zu sortieren, erspart Ihnen den Aufwand, nach ihnen zu suchen und um das Projekt herumzunavigieren. Verwenden Sie Ordner nicht, um Objekte nach ihren Dateitypen (z. B. Bilder, JavaScript-Dateien, CSS-Dateien) oder nach ihren Rollen (z. B. Formatter, Views, Controller) zu gruppieren. Diese Dateien sind nicht miteinander verknüpft, und Sie werden sie nicht zusammen bearbeiten. Wir empfehlen, die View-Dateien **~.controller.js** und **~.view.xml** nebeneinander im selben Ordner zu speichern. Die beiden Objekte stehen in Beziehung zueinander, und Sie werden viel zwischen ihnen navigieren. Alle Views im Ordner **/view** und alle Controller im Ordner **/controller** zu speichern, führt nur zu Verwirrung und endloser Navigation. In Abbildung 3.3 sind die Artefakte so gruppiert, dass Sie einen View und seinen Controller leicht identifizieren können.


**Abbildung 3.3** UI-Komponente mit Artefakten gruppiert nach Semantik

In den nächsten Abschnitten zeigen wir Ihnen, wie Sie Komponenten verschiedener Arten von SAPUI5-Anwendungen verwenden.

## 3.2 Wichtige Artefakte

Sehen wir uns nun einige wichtige Artefakte an, die beim Erstellen von SAPUI5-Anwendungen benötigt werden. In diesem Abschnitt werden wir den Component Controller, die Deskriptordatei (**manifest.json**), den Root-View und die Datei **index.html** erläutern.

### 3.2.1 Component Controller

Der Component Controller, allgemein **Component.js** genannt, stellt die Laufzeitmetadaten und die Methoden der Komponente bereit. Wir empfehlen Ihnen, die Syntax *Asynchronous Module Definition* (AMD) zu nutzen, um den Component Controller zu definieren. AMD ist eine Spezifikation, die eine API definiert, die wiederum die Module und deren Abhängigkeiten definiert und ermöglicht, diese asynchron zu laden. Wenn die Factory-Funktion des Component Controllers aufgerufen wird, geht die Funktion davon aus, dass die Abhängigkeiten leicht verfügbar sind, aber diese Annahme könnte katastrophal sein und zum Abbruch der Anwendung führen. Mithilfe der AMD-Syntax können Abhängigkeiten für den Component Controller eindeutig definiert werden, sodass sie asynchron geladen werden können. Die AMD-Syntax beschränkt sich nicht nur auf Component Controller, sondern kann für beliebige Controller verwendet werden. In Listing 3.3 definiert die globale SAPUI5-Funktion `sap.ui.define` einen Controller, und die Funktion akzeptiert zwei Parameter: Das erste Argument ist ein Array, das die Liste der Abhängigkeiten enthält, in unserem Beispiel die Klasse `sap.ui.core.UIComponent`, die vom Controller `sap.cleanui5.demo.watermark` benötigt wird. Sobald die Abhängigkeiten geladen sind, wird die Callback-Funktion, die das zweite Argument für die globale Funktion ist, zusammen mit Modulen als Parameter aufgerufen (in diesem Fall `UIComponent`). Die Parameter müssen nicht denselben Namen wie der Modulname haben, aber dies bietet ein klares Bild und einen einheitlichen Ansatz für Ihren Anwendungscode, da in SAPUI5-Anwendungen abhängige Bibliotheken und Module geladen werden müssen.

```
sap.ui.define([
  'sap/ui/core/UIComponent'
], function (UIComponent) {
  'use strict';

  return UIComponent.extend('sap.cleanui5.demo.watermark', {
    metadata: {
      manifest: 'json'
    }
  });
});
```

**Listing 3.3** Component Controller in der AMD-Syntax definiert

Die Komponentenmetadaten sollten extern in der Deskriptordatei (**manifest.json**) definiert werden, da der Deskriptor für moderne Komponenten obligatorisch ist und auch eine optimierte Performance ermöglicht.

Komponenten können Schnittstellen implementieren. In dem Beispiel in Listing 3.4 ermöglicht das Marker Interface `sap.ui.core.IAsyncContentCreation` das asynchrone Anlegen einer Komponente.

```
sap.ui.define([
  'sap/ui/core/UIComponent'
], function (UIComponent) {
  'use strict';

  return UIComponent.extend('sap.cleanui5.demo.watermark', {
    metadata: {
      manifest: 'json',
      interfaces: [
        'sap.ui.core.IAsyncContentCreation'
      ]
    }
  });
});
```

**Listing 3.4** Component Controller mit dem Marker Interface »sap.ui.core.IAsyncContentCreation«

Die Implementierung dieser Schnittstellen hat folgende zusätzliche Auswirkungen:

- Die Routing-Konfiguration und der Root-View werden automatisch auf `async=true` gesetzt, wodurch sichergestellt wird, dass ihre Abhängigkeiten asynchron geladen werden.
- Geschachtelte Views, einschließlich der Fragmente, werden ebenfalls asynchron verarbeitet.
- Die Fehlerbehandlung beim Laden und Verarbeiten von Views ist strenger und schlägt fehl, wenn eine View-Definition Fehler enthält (z. B. ein falscher Pfad für die Abhängigkeiten). Daher werden die Controls in `sap.ui.core.Component.create` nur angelegt, wenn die Abhängigkeiten aufgelöst werden. Andernfalls schlägt die View-Erstellung fehl.

Durch die Erweiterung der Klasse `sap.ui.base.ManagedObject` stellt die Klasse `sap.ui.core.Component` spezifische Metadaten für Komponenten bereit. Die Getter und Setter dieser Eigenschaften werden automatisch generiert, können aber bei Bedarf überschrieben werden. Wie in Listing 3.5 gezeigt, wird das Objekt `properties` als Teil der Metadaten definiert, und der *Titel* und die `description` werden als Eigenschaften der Komponente `sap.cleanui5.demo.watermark` definiert. Sie können `defaultValue` verwenden, um einen Wert für die Eigenschaft zu definieren, wenn die Komponente angelegt wird. In diesem Beispiel hat die Eigenschaft `description` der Komponente den Standardwert `'No Description'`.

```
sap.ui.define([
  'sap/ui/core/UIComponent'
], function (UIComponent) {
  'use strict';

  return UIComponent.extend('sap.cleanui5.demo.watermark', {
    metadata: {
      manifest: 'json',
      properties: {
        title: {
          name: 'Title',
          type: 'string'
        },
        description: {
          name: 'Description',
          type: 'string',
          defaultValue: 'No Description'
        }
      }
    }
  });
});
```

**Listing 3.5** Eigenschaften der in den Metadaten definierten Komponente

Ähnlich wie Eigenschaften können Sie auch Aggregationen, Ereignisse und Public-Methoden für die Komponente definieren. Eine *Aggregation* ist eine spezielle Beziehung zwischen UI-Elementtypen. *Ereignisse* benachrichtigen einen Verwender über Änderungen an einer Komponente; z. B. ist »press« ein Ereignis, das ausgelöst wird, wenn ein Button gedrückt wird. Public-Methoden sind Funktionen, die vom Verwender der Komponente auf der Komponente aufgerufen werden können.

In dem Beispiel in Listing 3.6 hat die Komponente die Aggregation `sorter`, die eine Kardinalität mit mehreren Einträgen hat. Das heißt, es können beliebig viele Komponenten hinzugefügt werden, die durch den Wert von `type` innerhalb der Aggregation definiert werden. Die Aggregation `filters` hat eine Kardinalität von 1, d. h., es kann nur eine solche Komponente als Aggregation zur Komponente `sap.cleanui5.demo.watermark` hinzugefügt werden. Die Komponente hat ein Ereignis `rendered`, das keine Parameter hat. Sie können jedoch Listener zu diesem Ereignis in der konsumierenden Anwendung hinzufügen. Die Komponente verfügt auch über eine Public-Methode `render`, die von konsumierenden Anwendungen aufgerufen werden kann.

```
sap.ui.define([
  'sap/ui/core/UIComponent'
], function (UIComponent) {
  'use strict';

  return UIComponent.extend('sap.cleanui5.demo.watermark', {
    metadata: {
      manifest: 'json',
      properties: {
        title: {
          name: 'Title',
          type: 'string'
        },
        description: {
          name: 'Description',
          type: 'string',
          defaultValue: 'No Description'
        }
      },
      aggregations: {
        sorter: {
          type: 'sap.cleanui5.demo.watermark.sort.Sorter',
          multiple: false
        },
        filters: {
          type: 'sap.cleanui5.demo.watermark.filter.Filter',
          multiple: true,
          singularName: 'filter'
        }
      },
      events: {
        rendered: {
          parameters: {}
        }
      },
      publicMethods: [
        'render'
      ]
    }
  });
});
```

**Listing 3.6** Aggregationen, Ereignisse und Public-Methoden einer Komponente in ihren Metadaten definiert

Die Klasse `sap.ui.core.UIComponent` stellt zusätzliche Metadaten für die Konfiguration von Benutzeroberflächen und für die Navigation zwischen Views bereit. In SAPUI5 gibt es zwei Methoden für die initiale Instanziierung der Komponente: `init` und `createContent`.

Die Methode `init` wird automatisch aufgerufen, wenn die Instanz einer Komponente angelegt wird. Sie können die Methode `init` überschreiben, um das Modell zwischen dem Control und der Komponente zu verknüpfen.

Das SAP Fiori Launchpad fungiert als Anwendungscontainer und instanziiert die App, ohne dass eine lokale HTML-Datei für den Bootstrap vorhanden ist. Stattdessen wird die Deskriptordatei geparst, und die Komponente wird in die aktuelle HTML-Seite geladen. Dadurch können mehrere Apps im selben Kontext angezeigt werden. SAPUI5 unterstützt die hash-basierte Navigation, die das Verhalten des SAP Fiori Launchpad wie eine Single-Page-Anwendung unterstützt. Navigation und Routing werden über einen *Router* implementiert, um die Hash-Änderung und die Daten im Hash an einen oder mehrere Views der Anwendung weiterzuleiten. Die Routing-Instanz kann in der Methode `init` der Komponente initialisiert werden, wie in Listing 3.7 gezeigt. Stellen Sie jedoch sicher, dass die übergeordnete `init`-Methode aufgerufen wird, damit nichts gestört wird. Die Router-Instanz wird automatisch zerstört, wenn die Komponente zerstört wird.

```
sap.ui.define([
  'sap/ui/core/UIComponent'
], function (UIComponent) {
  'use strict';

  return UIComponent.extend('sap.cleanui5.demo.watermark', {

    // Some code in here

    init: function () {
      // call the init function of the parent first
      UIComponent.prototype.init.apply(this, arguments);

      // this automatically initializes the router
      this.getRouter().initialize();
    }
  });
});
```

**Listing 3.7** Komponente mit »init«-Methode überschrieben

Die Komponente `sap.ui.core.UIComponent` legt das Root-Control so an, wie es in **manifest.json** als `sap.ui5/rootView` deklariert ist. Bei Bedarf, wie bei dynamischem View-Inhalt, kann jedoch die Methode `createContent` überschrieben werden, um das Root-Control programmatisch anzulegen. Listing 3.8 zeigt die Komponente `sap.cleanui5.demo.watermark` zum Überschreiben der Methode `createContent`, was dazu dient, ein Control `sap.m.Label` zurückzugeben, das der View hinzugefügt werden kann.

```
sap.ui.define([
  'sap/ui/core/UIComponent',
  'sap/m/Label'
], function (UIComponent, Label) {
  'use strict';

  return UIComponent.extend('sap.cleanui5.demo.watermark', {
    metadata: {
      manifest: 'json'
    },

    createContent: function () {
      return new Label({ text: 'Hello!' });
    }
  });
});
```

**Listing 3.8** Komponente mit überschriebener »createContent«-Methode und sofort zurückgegebenem Control

In Listing 3.9 implementiert die Komponente `sap.cleanui5.demo.watermark` das Interface `sap.ui.core.IAsyncContentCreation`. Die Methode `createContent` gibt das Promise der `XMLView.create`-Factory zurück. Dieser Ansatz wird der Verwendung von Promises vorgezogen und stellt sicher, dass die Controls und ihre Abhängigkeiten aufgelöst werden, bevor die Komponente gerendert werden kann.

```
sap.ui.define([
  'sap/ui/core/UIComponent',
  'sap/ui/core/mvc/XMLView'
], function (UIComponent, XMLView) {
  'use strict';

  return UIComponent.extend('sap.cleanui5.demo.watermark', {
    metadata: {
      manifest: 'json',
      interfaces: [
        'sap.ui.core.IAsyncContentCreation'
```

```
      ]
    },

    createContent: function () {
      return XMLView.create({
        // Some code in here
      });
    }
  });
});
```

**Listing 3.9** Komponente mit überschriebener »createContent«-Methode und asynchron zurückgegebenem Control

### 3.2.2 Deskriptor

Die Deskriptordatei, die im Allgemeinen **manifest.json** genannt wird, basiert auf dem Konzept Web Application Manifest (siehe *https://www.w3.org/TR/appmanifest/*), das vom W3C (siehe *https://www.w3.org/*) eingeführt wurde. Das Web Application Manifest gibt an, dass es sich bei einem Deskriptor um ein auf JavaScript Object Notation (JSON) basierendes Dateiformat handelt, das als zentraler Ort dient, um die mit einer Anwendung, einer Anwendungskomponente oder einer Bibliothek verknüpften Metadaten in einem maschinenlesbaren Format in einem leicht zugänglichen Format zu speichern. Ein Deskriptor enthält die Komponentenmetadaten und drückt die Komponentenabhängigkeiten und die Konfiguration aus. Wenn Sie einen Deskriptor verwenden, müssen Sie weniger Anwendungscode schreiben.

Zu den wichtigsten Attributen, die im Deskriptor definiert sind, gehören folgende Informationen:

- Modelle, z. B. die Konfiguration des OData-Services (Standardmodell oder unbenanntes Modell) und Sprachdateien (das i18n-Modell). In der Deskriptordatei definierte Modelle werden automatisch instanziiert, wenn die Komponente gestartet wird.
- in der Anwendung verwendete Bibliotheken und Komponenten, die während der Anwendungsinitialisierung geladen werden müssen
- der Root-View der Anwendung
- Routing-Konfiguration, die die Navigation zwischen Views definiert

Der Deskriptor für Bibliotheken enthält eine Teilmenge der Attribute, die im Deskriptor für Anwendungen und Komponenten definiert sind. Obwohl nur der Component Controller (**Component.js**) für eine Komponente obligatorisch ist, da die Konfiguration inline als Objekt zum Component Controller definiert werden kann, empfehlen

wir, den Deskriptor als separate Datei (**manifest.json**) zu verwenden, um den Anwendungscode klar von den Konfigurationseinstellungen zu trennen. Auf diese Weise können Sie die Komponente noch flexibler gestalten, da der Deskriptor als separate Datei auch dafür sorgt, dass er von SAP-Fiori-Werkzeugen analysiert werden kann. Alle SAP-Fiori-Anwendungen sind als Komponenten realisiert und werden mit einer Deskriptordatei ausgeliefert, die im SAP Fiori Launchpad gehostet werden soll.

### 3.2.3 Root-View

Die Datei **App.view.xml** definiert den Root-View der Anwendung. Sie sollten nicht den gesamten Inhalt der Seite in dieser Datei sichern. Der Root-View sollte nur das grundlegende Layout der Anwendung oder des Containers für das Control enthalten. SAPUI5 unterstützt verschiedene View-Typen, die wir in Abschnitt 3.6, »Model-View-Controller-Assets«, beschreiben werden. Wir empfehlen jedoch die Verwendung von XML-Views, die sicherstellen, dass die Controller-Logik (z. B. **App.controller.js**) von der View-Definition entkoppelt ist, wodurch der Code lesbarer und wartbarer wird. Wir empfehlen außerdem, für jeden View, den Sie in Ihrer Anwendung verwenden möchten, eine eigene View-Datei anzulegen.

### 3.2.4 Die Datei »index.html«

Wenn Sie eine komponentenbasierte Anwendung als eigenständige Anwendung ausführen, d. h. ohne das SAP Fiori Launchpad zu verwenden, müssen Sie eine Datei **index.html** im Ordner **/webapp** Ihrer Anwendung anlegen und das Bootstrapping von SAPUI5 nutzen. Ein Beispiel für einen Bootstrapping-Code sehen Sie in Listing 3.10.

```
<script id="sap-ui-bootstrap"
  src="resources/sap-ui-core.js"
  data-sap-ui-theme="sap_fiori_3"
  data-sap-ui-resourceroots='{"sap.cleanui5.demo.sample": "./"}'
  data-sap-ui-compatVersion="edge"
  data-sap-ui-oninit="module:sap/ui/core/ComponentSupport"
  data-sap-ui-async="true"
  data-sap-ui-frameOptions="trusted">
```

**Listing 3.10** Bootstrapping von SAPUI5

**Bootstrapping SAPUI5**

Bootstrapping ist der Prozess, bei dem SAPUI5 in einer Anwendung geladen und initialisiert wird.

## 3.3 Freestyle-Anwendungen

Anwendungen, die mit einem Freestyle-Ansatz entwickelt werden, werden mithilfe von SAPUI5-Controls von Grund auf neu angelegt. UI-Elemente auf dem Screen platzieren, Logik hinzufügen, um das Modell an die UI-Controls zu binden, das entsprechende Backend aktualisieren, wenn die Modelle über Controls geändert werden, Seitenübergänge einrichten und mehr – all dies muss vom Anwendungsentwickler bzw. von der Anwendungsentwicklerin adressiert werden. In der Regel sind diese Aktivitäten zeitaufwendig und fehleranfällig. Die Einhaltung von Produktstandards und Designrichtlinien ist eine mühsame, aber notwendige Aufgabe bei der Erstellung von Anwendungen in einem Freestyle-Ansatz, insbesondere bei komplexen UI-Anforderungen.

Die drei Hauptordner in Freestyle-Anwendungen sind der Ordner **/root**, der Ordner **/webapp** und der Ordner **/test**. Der Ordner **/webapp** befindet sich im Ordner **/root**, und der Ordner **/test** sollte sich im Ordner **/webapp** befinden, wie in Abbildung 3.4 dargestellt.


**Abbildung 3.4** Ordnerstruktur einer Freestyle-SAPUI5-Anwendung

Schauen wir uns zunächst die einzelnen Ordner genauer an:

- **/root**
  Der Ordner **/root**, mit demselben Namen wie Ihre Anwendung, darf nur Dateien enthalten, die nicht Teil Ihres Anwendungscodes sind, wie die Build-Konfigurationsdateien (**pom.xml** für Maven oder **Gruntfile.js** für Grunt) und Dokumentationsdateien (**README.md**). Der Ordner **/root** enthält auch die SAPUI5-Tooling-Konfigurationsdateien des Projekts (siehe *https://sap.github.io/ui5-tooling/*), bei denen es sich in der Regel um YAML-Dateien (**ui5.yaml**) handelt.

- **/webapp**
  Der Ordner **/webapp** enthält alle für den Anwendungscode relevanten Dateien, einschließlich Dateien, die sich auf die Lokalisierung beziehen (**i18n.properties-Dateien**).
- **/test**
  Der Ordner **/test** enthält alle Dateien, die zum Ausführen automatisierter Tests für Ihre Anwendung sowie zum Starten Ihrer Anwendung im Sandbox-Modus erforderlich sind, sodass Sie manuelle Tests durchführen und die Anwendung mit einem Mock-Server ausführen können, um die Abhängigkeiten einer Anwendung vom Backend-OData-Service zu isolieren.

Vielleicht fragen Sie sich, warum Sie separate Ordner benötigen und warum nicht alles in einem Ordner abgelegt werden kann. Performance ist einer der wichtigsten Produktstandards für jede Anwendung, die Sie entwickeln. Kunden möchten nicht lange auf das Laden einer Anwendung warten, und daher ist eine solide Performance beim Laden der Anwendung entscheidend. Um dieses Ziel zu erreichen, darf die Anwendung, wenn sie gebündelt ist, keine unnötigen Dateien haben und muss die Dateien **component-preload.js** und **manifest.json** enthalten, die helfen, Abhängigkeiten asynchron zu laden. Wenn sich bei der Verwendung von Paketierungstools alle Dateien, die Sie bereitstellen möchten, im selben Ordner befinden, ist die Dateiverwaltung effizienter. Der Ordner **/webapp**, der sich im Ordner **/root** der Anwendung befindet, muss alle Dateien enthalten, die Sie auf Produktionsservern deployen möchten. Andere Dateien, die für die Ausführung der Anwendung nicht benötigt werden, können direkt im Ordner **/root**, aber außerhalb des Ordners **/webapp** abgelegt werden. Der Ordner **/test**, der sich ebenfalls im Ordner **/webapp** befindet, enthält die Ressourcen, die für die Ausführung automatisierter Tests während der Designzeit erforderlich sind, und enthält Testautomatisierungen. Obwohl der Inhalt des Ordners **/test** von der Anwendung nicht benötigt wird, müssen die Inhalte innerhalb des Ordners **/webapp** referenziert werden, weshalb der Ordner **/test** im Ordner **/webapp** abgelegt werden muss, um über relative Pfade auf diese Inhalte zugreifen zu können. Durch die Gruppierung von Artefakten im Ordner **/test** können Sie Werkzeuge erstellen, um sie von der Aufnahme in **component-preload.js** für das produktive Deployment auszuschließen.

Neben dem Ordner **/test** enthält der Ordner **/webapp** in einer SAPUI5-Anwendung drei weitere Ordner, die sich auf das MVC-Muster beziehen, das häufig in SAPUI5-Anwendungen verwendet wird (die wir in Abschnitt 3.6, »Model-View-Controller-Assets«, behandeln), sowie einen Lokalisierungsordner und einen lokalen Services-Ordner für die Emulation von OData-Services für manuelle und automatisierte Tests. Sehen wir uns diese Ordner genauer an:

- **/i18n**
  Enterprise-Anwendungen sollen mehrere Sprachen unterstützen; diese Funktion wird als *Internationalisierung* (i18n) bezeichnet. Dabei werden die Texte von der Anwendung in der vom Benutzer gewünschten Sprache gerendert. Auch SAPUI5 unterstützt diese Funktion, und die auf der Benutzeroberfläche gepflegten Texte, in der Regel statische Texte, werden der Datei **i18n.properties** hinzugefügt. Diese Datei wird von Übersetzerinnen und Übersetzern verwendet, um die Inhalte in unterschiedliche Sprachen zu übersetzen. Dabei folgt der Name der Datei der Namenskonvention `i18n_<LANGUAGE_KEY>.properties` mit einem Sprachenschlüssel zur Unterscheidung. Üblicherweise heißt der Ordner, der alle diese Dateien enthält, auch **/i18n**.
- **/localService**
  SAPUI5-Anwendungen basieren größtenteils auf OData-Services. Sie können sich beim Entwickeln einer Anwendung nicht immer auf einen Backend-Service verlassen. Sie können jedoch Emulatoren wie den OData-V2-Mock-Server verwenden, um nicht nur Anwendungen in der Vorschau anzuzeigen, sondern auch automatisierte Tests auszuführen. Solche Emulatoren benötigen das Metadatendokument der OData-Services, die Datei **metadata.xml**, das in der Anwendung verwendet wird. Der Speicherort dieses Dokuments muss im Deskriptor im Abschnitt `data sources` gepflegt werden. Der Mock-Server ist in der Datei **mockserver.js** definiert, die sich ebenfalls im Ordner **/localService** befindet. Der Mock-Server stellt der Anwendung Daten als JSON-Dateien zur Verfügung, die lokal im Ordner **/mockdata** gespeichert werden (der Ordnername ist nur eine Konvention). Es empfiehlt sich, jede Entität als separate JSON-Datei aufzubewahren. Ein Beispiel für den Ordner **/localService** ist in Abbildung 3.5 dargestellt.


**Abbildung 3.5** Lokaler Serviceordner in einer SAPUI5-Anwendung

## 3.4 SAP Fiori Elements

*SAP Fiori* ist eine Designsprache, die dafür sorgen soll, dass Unternehmensanwendungen besonders benutzerfreundlich sind. *SAP Fiori Elements* ist ein Framework zur Entwicklung von Benutzeroberflächen, das auf SAPUI5 aufbaut. Durch das Einhalten der Designprinzipien von SAP Fiori können Sie eine personalisierte, flexible und einfache Benutzererfahrung sicherstellen. SAP Fiori stellt einen Paradigmenwechsel dar,

weg von den monolithischen Enterprise-Resource-Planning-Lösungen (ERP-Lösungen) hin zu einfachen Anwendungen, die auf die Anforderungen der Benutzer zugeschnitten sind. Das SAP Fiori Launchpad ist nach Benutzerrollen organisiert und fungiert als zentraler Einstiegspunkt für den Zugriff auf alle SAP-Fiori-Anwendungen über Kacheln. Im Launchpad sind Services für Navigation, Personalisierung, Single Sign-on und Suche enthalten. Das Launchpad und seine Kacheln sind flexibel und können an die Bedürfnisse der Benutzer angepasst werden.

SAP Fiori bietet die folgenden Layouts zum Erstellen von Anwendungsseiten:

- Vergleichsmuster
- dynamische Seite
- semantische Seite
- flexibles Spaltenlayout
- Letterboxing
- Mehrfachinstanz

Das Standardseitenlayout in SAP Fiori ist die dynamische Seite, die aus einem Kopf-, einem Inhalts- und einem Fußzeilen-Bereich besteht.

Floorplans sind ebenfalls dynamische Seiten, werden jedoch für bestimmte Anwendungsfälle eingesetzt und sind daher mit einer Kombination aus UI-Elementen in Kopfzeile, Inhaltsbereich und Fußzeile ausgestattet. SAP Fiori bietet Floorplans, die folgende Aufgaben ausführen:

- Übersicht über Informationen und Aufgaben bereitstellen (Übersichtsseite oder Overview Page)
- mehrere Objekte auflisten (List Report, Analytical List Report und Arbeitsvorrat)
- Objekte verwalten (Objektseite und Wizard)
- Navigation an einem Objekt ermöglichen (Einstiegsseite)

Um SAP-Fiori-Designkonzepte umzusetzen, wird die SAP-Fiori-Designsprache in verschiedenen UI-Technologien implementiert und für eine effiziente Übernahme in mehrere Frameworks wie SAP Fiori Elements optimiert.

SAPUI5-Anwendungen sind oft komplex und können endlose Zeilen Code enthalten. Auch die Skalierung erfordert häufig einen hohen Entwicklungsaufwand, z. B. bei der Pflege von Screens. SAP Fiori Elements bietet eine einfachere Möglichkeit, Anwendungen zu entwickeln. Diese Bibliothek enthält mehrere Floorplans für SAP Fiori, die in den meisten Geschäftsszenarien verwendet werden. Mit dem Framework können Entwicklerinnen und Entwickler SAP-Fiori-Anwendungen für Unternehmen erstellen, die auf einem OData-Service basieren und Annotationen verwenden, die darauf basieren, welche Views jedes Mal generiert werden, wenn die Anwendung gestartet wird. Dank Annotationen benötigen Sie möglicherweise fast keinen UI-Code. Die ent-

standene Anwendung verwendet vordefinierte View-Templates und Controller, die zentral bereitgestellt werden. Wenn die User Experience aktualisiert wird, übernehmen die Anwendungen automatisch die meisten Änderungen, ohne dass das Entwicklungsteam eingreifen muss, und stellen so die Geschäftskontinuität sicher.

**Vorteile von SAP Fiori Elements**

Der Aufwand beim Erstellen einer Anwendung mit SAP Fiori Elements kann zunächst höher sein als beim Anlegen einer Freestyle-SAP-Fiori-App. Wenn Sie jedoch mehrere Anwendungen auf diese Weise erstellt haben, werden Sie für diesen Aufwand reichlich belohnt, da Ihre Anwendungen von der Verwendung des Frameworks profitieren werden. Folgende Funktionen stehen zur Verfügung:

- Unterstützung mehrerer Geräte
- Navigation durch Deep Link
- Speichern und Wiederherstellen von Applikationszuständen
- Rücknavigation unter Berücksichtigung der Historie
- Busy Handling zur Vermeidung von versehentlichen Doppelklicks
- UI-Flexibilität
- Unterstützung von Produktstandards wie Barrierefreiheit, UI-Performance und Sicherheit
- Bearbeitungsmodussteuerung
- Handhabung von Entwurfsdokumenten (Entwurfssicherung)
- Nachrichtenbehandlung, einschließlich Nachrichtenlebenszyklen und Meldungsanzeige
- interne und externe Navigation
- Wertehilfeunterstützung

SAP-Fiori-Elements-basierte Anwendungen folgen derselben Art der Projektstrukturierung wie Freestyle-SAPUI5-Anwendungen, jedoch mit den folgenden Ergänzungen, wie in Abbildung 3.6 dargestellt:

- **/annotations**
  Dieser Ordner enthält lokal definierte Annotationen, die nur auf der UI (im XML-Format) verfügbar sind. In diesem Ordner definierte Annotationen können die im Backend für den OData-Service definierten Annotationen überschreiben.
- **/changes**
  Dieser Ordner enthält Dateien mit der Erweiterung **~.changes**, die vom visuellen Editor von SAP generiert werden, wenn Änderungen mit dem Tool für No-Code-Erweiterungen oder Modifikationen an SAP-Fiori-Elements-basierten Anwendungen vorgenommen werden.

- **/extension oder /ext**
  Der Ordner **/extension** enthält den Controller und die View-Erweiterung für die Templates. Dieser Ordner kann auch Fragmente für die Erweiterung von Floorplans enthalten.

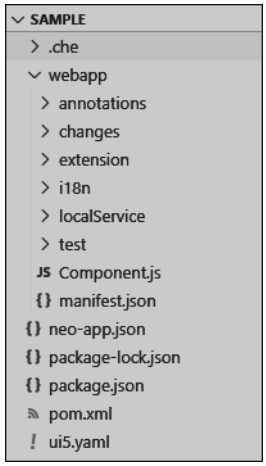


**Abbildung 3.6** SAP-Fiori-Elements-basierte Anwendung

Sehen wir uns die verschiedenen verfügbaren Floorplans an und lernen als Nächstes einige wichtige Aspekte kennen, wie SAP-Fiori-Element-basierte Anwendungen generiert werden.

### 3.4.1 Floorplans

Der Floorplan *List Report* ist einer der am häufigsten verwendeten Floorplans. Er ist Nachfolger des klassischen Master-Detail-Floorplans und bietet erweiterte Funktionen wie das Sortieren und Filtern großer Datenmengen. Die Liste zeigt nur eine Übersicht der Positionen an. Dies ist der allgemeine Einstiegspunkt, um auf die Details der Positionen zuzugreifen. Dieser Floorplan bietet auch Aktionen, die Sie für diese Positionen ausführen können. Ein Beispiel ist in Abbildung 3.7 dargestellt.

Der andere Nachfolger des klassischen Master-Detail-Floorplans ist der Floorplan *Objektseite* (engl. Object Page), mit dem Details zu einer Position angezeigt werden können. Inhalte können in verschiedene Abschnitte gruppiert werden, und jeder Abschnitt kann mithilfe der im Floorplan verfügbaren Anker schnell aufgerufen werden, wie im Beispiel in Abbildung 3.8 dargestellt. Vorlagen können Standardaktionen wie das Umschalten vom Anzeige- in den Bearbeitungsmodus, das Löschen einer Position und andere benutzerdefinierte Aktionen, die oben links platziert sind, enthalten. Sie können diese Seite auch als Vorlage für das Anlegen einer Position verwenden.

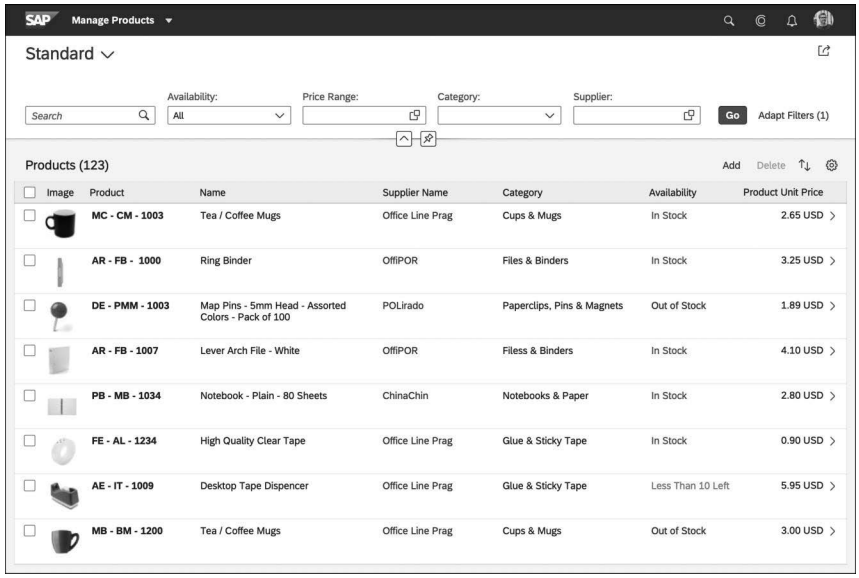


**Abbildung 3.7** Anwendung mit dem List-Report-Floorplan

Beachten Sie, dass dieser Floorplan nur verwendet werden kann, um jeweils eine Instanz zu bearbeiten. Während der Entwicklung werden häufig der List Report und die Floorplans für Objektseiten zu einer Anwendung kombiniert, aber der Floorplan Objektseiten kann auch verwendet werden, um eigenständige Seiten mit einem anderen Einstiegspunkt als dem List Report zu entwickeln.

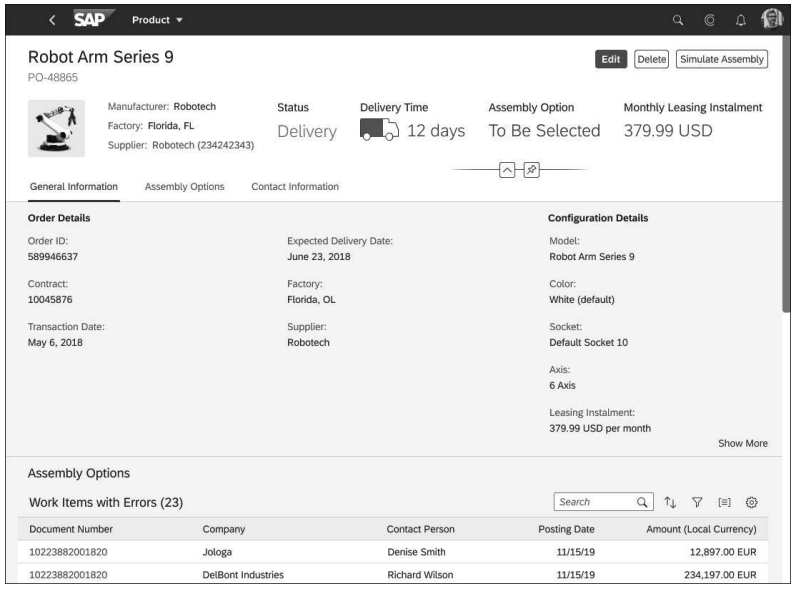


**Abbildung 3.8** Anwendung mit dem Objektseite-Floorplan

Eine Liste von Elementen anzuzeigen und zu erwarten, dass der Benutzer die Elemente herausfiltert, die seine Aufmerksamkeit erfordern, ist nicht unbedingt der beste Weg, die Zeit von Geschäftsanwendern richtig einzusetzen. Stattdessen sollte ein eher datengesteuerter Ansatz gewählt werden, um dem Benutzer alle erforderlichen Informationen an einem Ort anzuzeigen und so sicherzustellen, dass der Benutzer schnell handeln kann. Dafür gibt es den Floorplan *Übersichtsseite* (engl. Overview Page, kurz OVP). Ein Beispiel für eine OVP, die einen Überblick über den Beschaffungsprozess gibt, sehen Sie in Abbildung 3.9. Mit diesem Floorplan können Sie verschiedene Arten von Daten in Form von Karten anzeigen. Der Floorplan ermöglicht auch das Sortieren, Filtern und andere Aktionen.

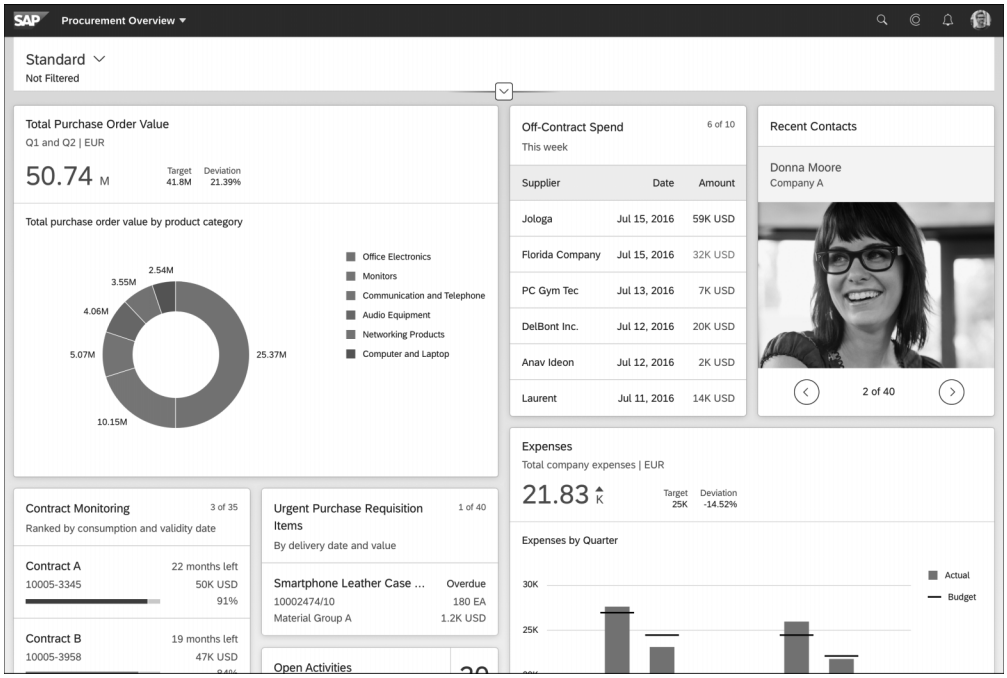


**Abbildung 3.9** Anwendung mit einem Übersichtsseite-Floorplan

Ähnlich wie ein List Report zeigt auch der *Arbeitsvorrat* (engl. Worklist) dem Benutzer eine Liste von Elementen an, jedoch mit einer Teilmenge der Funktionen. Dieser Floorplan wird in der Regel verwendet, wenn ein Benutzer nur eine Liste von Aufgaben abschließen und nicht zu einer anderen Seite navigieren muss, um Details der Position anzuzeigen. Ein Beispiel sehen Sie in Abbildung 3.10 dargestellt. Hier zeigt der Floorplan nur eine Liste von Positionen mit Fehlern an. Darüber hinaus haben Sie Optionen zum Filtern der Elemente nach Status über die segmentierte Schaltfläche. Sie benötigen keine Objektseite, um Details zum Element anzuzeigen. Stattdessen leitet die semantische Navigation den Benutzer zur entsprechenden Objektseite weiter, die möglicherweise als Teil einer anderen Anwendung entwickelt wurde.

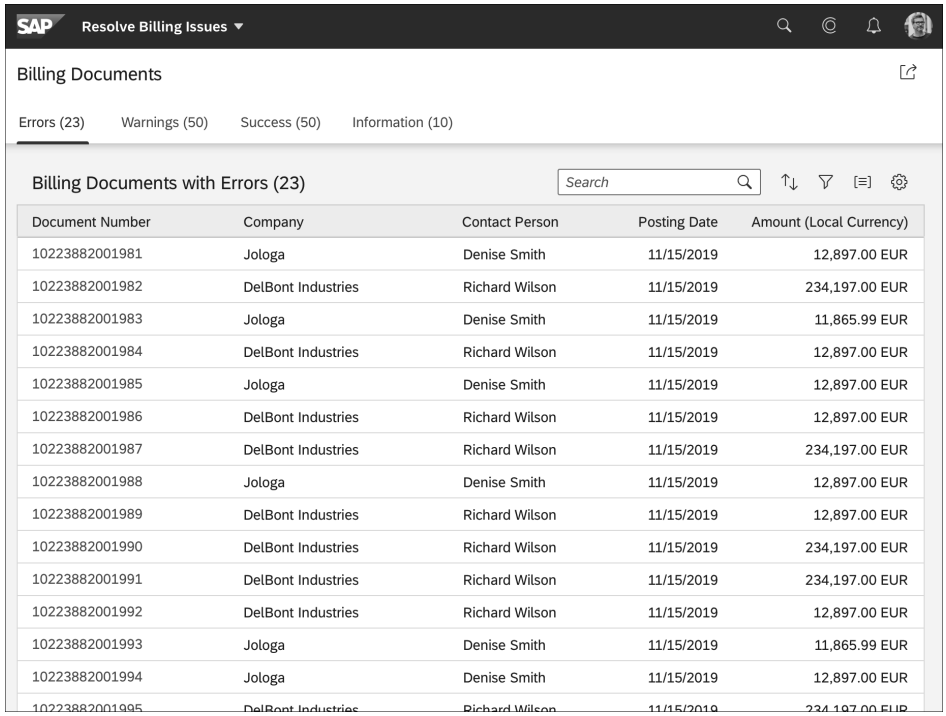


**Abbildung 3.10** Anwendung mit einem Arbeitsvorrat-Floorplan



**Floorplans in SAP Fiori Elements**

Mit Ausnahme des Wizards und der Einstiegsseite sind alle Floorplans in SAP Fiori Elements verfügbar. Weitere Informationen zu unterstützten Floorplans finden Sie unter *http://s-prs.de/v1005027*.

### 3.4.2 Anwendungsgenerierung

Das SAP-Fiori-Elements-Framework generiert die Anwendung zur Laufzeit basierend auf den folgenden drei Schlüsselaspekten:

- Der *OData-Service*, der in der Deskriptordatei der Anwendung (**manifest.json**) definiert ist, ist die primäre Datenquelle für die Anwendung sowohl für Abfrage- als auch für Änderungsvorgänge. Die goldene Regel ist, sicherzustellen, dass eine SAP-Fiori-App höchstens ein OData-Service-Binding hat, um eine saubere Trennung zu gewährleisten. Diese Regel gilt jedoch nicht zwingend.
- *Annotationen* sind die Provider zusätzlicher Metadaten, um Bildschirmelemente und Benutzerinteraktionen zu beeinflussen. Im Backend definierte Annotationen werden Teil des OData-Services. Annotationen können auch lokal auf der Benut-

zeroberfläche definiert werden, was manchmal bevorzugt wird, da nicht alle Annotationen im Backend-System verfügbar sind. Wenn sie lokal auf der Benutzeroberfläche definiert sind, können Annotationen jedoch nicht wiederverwendet werden, wenn die Entitäten in anderen OData-Services wiederverwendet werden.

- Das *SAP-Fiori-Elements-Framework* verwendet zusammen mit dem ausgewählten Floorplan den OData-Service und die Annotationen, um den View zu generieren, und stellt auch den Standard-Controller für die Anwendungslaufzeit bereit.

Abbildung 3.11 zeigt, wie eine auf SAP Fiori Elements basierende Anwendung zur Laufzeit generiert wird. Beachten Sie auch, dass die Floorplans nur häufig verwendete Funktionen bieten. Bei Bedarf kann die Anwendung jedoch mit verschiedenen Techniken erweitert werden, die vom SAP-Fiori-Elements-Framework und den zugehörigen Designwerkzeugen angeboten werden.

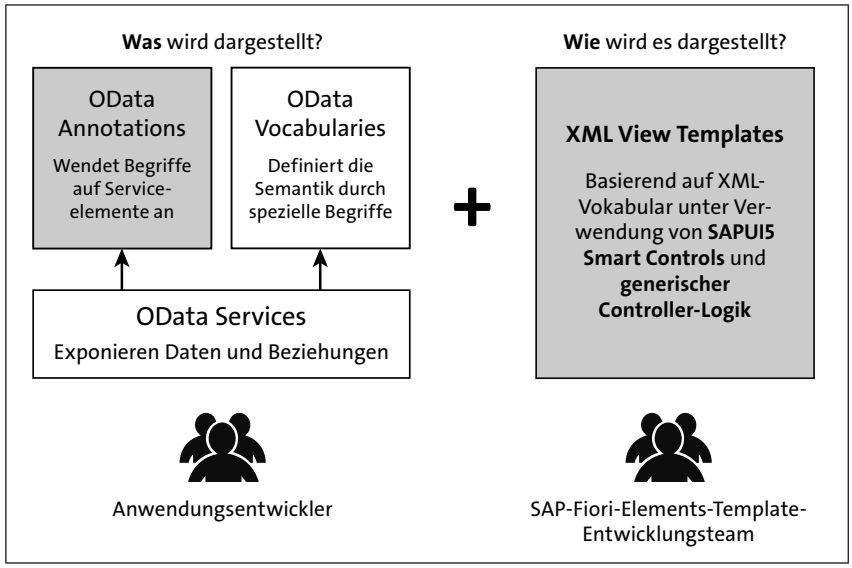


**Abbildung 3.11** Funktionsweise von SAP Fiori Elements

*Vokabulare* sind Namensräume, die Begriffe enthalten, mit denen die Metadaten eines OData-Services erweitert werden können, der Clients erklärt, wie der Service und seine Daten zu interpretieren sind. Es gibt zwei Arten von Annotationen:

- *Metadatenannotationen* werden verwendet, wenn Sie einem Service, Entitätstyp, einer Eigenschaft, Funktion, Aktion oder einem Parameter zusätzliche Merkmale oder Fähigkeiten hinzufügen möchten. Ein Beispiel für dieses Szenario wäre `sap:label="Currency"`, eine allgemeine Annotation, die als Teil des Metadatendokuments eines OData-Services verfügbar ist und den Bezeichner eines bestimmten Metadatenelements definiert.

- *Instanzannotationen* werden verwendet, um zusätzliche Informationen hinzuzufügen, die mit einem Ergebnis, einer Entität oder einer Eigenschaft verknüpft sind. Diese Annotationen werden mit der Ergebnismenge der OData-Querys hinzugefügt. Ein Beispiel für dieses Szenario wäre `Update_mc: true`, das definiert, dass die Instanz dieser Entität aktualisiert werden kann (vorausgesetzt, die aktualisierbare Eigenschaft der Entität ist instanzspezifisch). Als Richtlinie müssen instanzunabhängige Annotationen als Teil der Metadaten angegeben werden, während Annotationen, die von der Instanz abhängen, für jede Instanz angegeben werden müssen.



**Floorplans in SAP Fiori Elements**

SAP Fiori Elements unterstützt nur einen einzelnen Service als den Service hinter den Daten für alle seine Controls.

## 3.5 Bibliotheksprojekte

Eine UI-Bibliothek ist eine Einheit von Controls, die deployt werden kann. Controls werden nie allein implementiert, sondern sind immer Teil einer Control-Bibliothek. Bibliotheken können auch für das Deployment wiederverwendbarer Komponenten verwendet werden, z. B. wiederverwendbare Komponenten des Application Logs.

Während wir dieses Buch schreiben, wird die Generierung eines Bibliotheksprojekts über eine Vorlage in SAP Business Application Studio nicht unterstützt. Einige unserer Beispiele wurden mit dem Wizard zum Anlegen eines Projekts aus Vorlagen (SAP-Fiori-Bibliotheken) in SAP Web IDE generiert.

In den folgenden Abschnitten stellen wir Ihnen einige wichtige Artefakte in SAPUI5-Bibliotheksprojekten vor und bieten einen Überblick über die Projektstruktur, die für Bibliotheksprojekte befolgt werden muss.

### 3.5.1 Die Datei »library.js«

Ähnlich wie Komponenten eine Component-Controller-Datei **Component.js** enthalten, haben auch SAPUI5-Bibliotheken eine Controller-Datei **library.js**, die den Code für Typen, Interfaces, Controls und Elemente enthält, die von der Bibliothek bereitgestellt werden, sowie eine bibliotheksspezifische Initialisierung. Diese Initialisierung initialisiert nicht die Controls innerhalb der Bibliothek. Die Funktion `sap.ui.getCore().initLibrary` legt eine Instanz der Bibliothek im angegebenen Namensraum an. Neben dem Namensraum können auch Abhängigkeiten für die Bibliothek definiert werden, die dann asynchron geladen wird. Wie in Listing 3.11 zu sehen, wird die Instanz der Bibliothek `sap.cleanui5.demo` zurückgegeben.

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

  // initialize the library with global name 'sap.cleanui5.demo'
  sap.ui.getCore().initLibrary({
    name: 'sap.cleanui5.demo'
  });

  return sap.cleanui5.demo;
});
```

**Listing 3.11** Definieren der »sap.cleanui5.demo«-Bibliothek

**Deskriptor für Bibliothek**

Der Deskriptor der Bibliothek enthält eine Teilmenge der Attribute im Deskriptor der Anwendungen und Komponenten.

### 3.5.2 SAPUI5-Reuse-Komponenten

Komponenten sind unabhängige Steuerelemente, die in Anwendungen wiederverwendet werden können, um unnötigen Mehraufwand zu vermeiden. Um Komponenten wiederzuverwenden, müssen sie Teil einer Bibliothek sein, die in der Regel Controls und Komponenten enthält. Komponenten können jedoch auch als Teil von konsumierenden Anwendungen eingebunden werden, wenn die Komponente nicht für die Wiederverwendung in anderen Anwendungen vorgesehen ist. Dadurch können Sie die Anwendungslogik von der Kontrolllogik oder der Komponentenlogik trennen. Listing 3.12 zeigt die UI-Komponente `watermark`, die in der Bibliothek `sap.cleanui5.demo` definiert ist.

```
sap.ui.define([
  'sap/ui/core/UIComponent'
], function (UIComponent) {
  'use strict';

  return UIComponent.extend('sap.cleanui5.demo.watermark', {
    metadata: {
      manifest: 'json',
      library: 'sap.cleanui5.demo'
    }
  });
});
```

**Listing 3.12** Komponente »watermark«, definiert in der Bibliothek »sap.cleanui5.demo«

Reuse-Komponenten können entweder als Teil einer SAP-Fiori-Element-basierten Anwendung oder einer Freestyle-SAPUI5-Anwendung eingebunden werden. Dieselbe Komponente kann jedoch nicht für beide Arten von Anwendungen verwendet werden, da die Reuse-Komponente in einer SAP-Fiori-Element-basierten Anwendung die Methoden `stStart` und `stRefresh` im Component Controller implementieren muss. Diese Lebenszyklusmethoden werden zur Laufzeit vom SAP-Fiori-Elements-Framework aufgerufen, wobei hervorzuheben ist, wie wichtig es ist, separate Komponenten zu haben, die sich auf eine gemeinsame Implementierung beziehen.

### 3.5.3 Ordnerstruktur

Die Ordnerstruktur einer Bibliothek unterscheidet sich grundlegend von der Ordnerstruktur von Anwendungen. Die drei Hauptordner in Bibliotheksprojekten sind der Ordner **/root**, der Ordner **/src** und der Ordner **/test**. Die Ordner **/src** und **/test** befinden sich im Ordner **/root**, wie in Abbildung 3.12 dargestellt.

Schauen wir uns jeden Ordner im Einzelnen genauer an:

- Der Ordner **/root**, der denselben Namen wie Ihre Bibliothek hat, sollte Dateien enthalten, die nicht Teil Ihres Bibliothekscodes sind, z. B. Build-Konfigurationsdateien wie **pom.xml** für Maven oder **Gruntfile.js**. Der Ordner **/root** enthält auch die SAP-UI5-Tooling-Konfiguration des Projekts (siehe *https://sap.github.io/ui5-tooling/*), die sich in der Regel in einer YAML-Datei namens **ui5.yaml** befindet.
- Der Ordner **/src** enthält die verschiedenen auslieferbaren Inhalte einer Bibliothek, in der Regel Controls oder Reuse-Komponenten.
- Der Ordner **/test** enthält alle Dateien, die zum Ausführen automatisierter Tests für Ihre Controls oder Komponenten sowie zum Starten Ihrer Testanwendungen im Sandbox-Modus erforderlich sind, sodass Sie manuelle Tests der Controls oder Komponenten durchführen können.

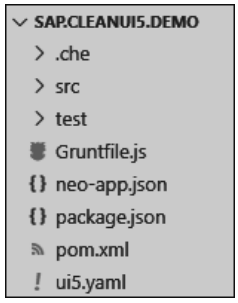


**Abbildung 3.12** Einfaches Bibliotheksprojekt

Wie Abbildung 3.13 zeigt, enthält unsere Beispielprojektstruktur für die Bibliothek `sap.cleanui5.demo.watermark` mehrere Steuerelemente.

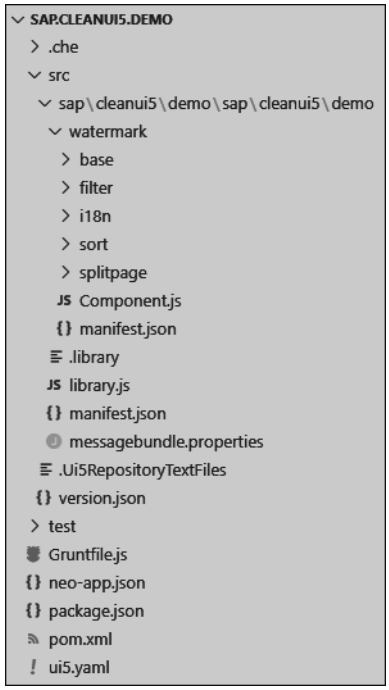


**Abbildung 3.13** Bibliotheksprojekt mit der Komponente »watermark«

## 3.6 Model-View-Controller-Assets

Model View Controller ist ein Architekturdesignmuster, das für die Entwicklung von Benutzeroberflächen verwendet wird. SAPUI5 hat dieses Designmuster ebenfalls übernommen, um die Darstellung von Informationen von der Benutzerinteraktion zu trennen. Dieses architektonische Entwurfsmuster unterteilt eine Anwendung in drei logische Teile, die die Entwicklung und Änderung von Teilen unabhängig voneinander erleichtern:

- Das Modell enthält die relevanten Anwendungsdaten.
- Der View ist alles, was für die Benutzer sichtbar ist, basierend auf den Informationen aus dem Modell.
- Der Controller steuert den Datenfluss in das Modell und aktualisiert den View, wenn sich das Modell ändert.

Abbildung 3.14 zeigt, wie das MVC-Architekturdesign-Musterkonzept mit SAPUI5 verwendet wird.

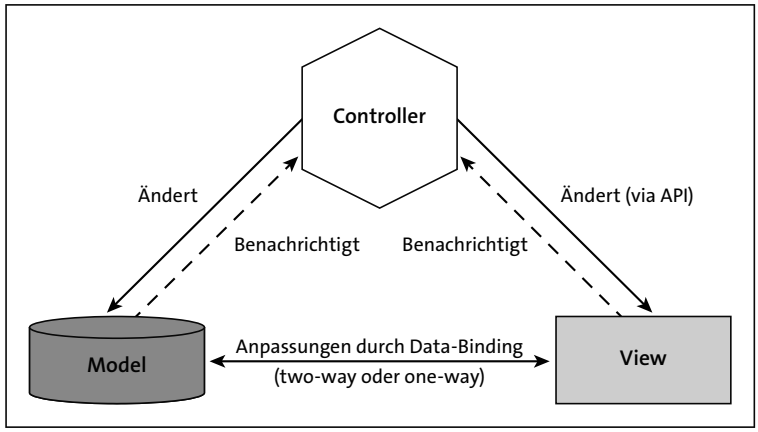


**Abbildung 3.14** MVC Architektur Design Pattern in SAPUI5

SAPUI5 verwendet *Data-Binding*, eine gängige Technik, die zwei Datenquellen bindet und synchron hält, sodass eine Änderung in einer Quelle in der anderen widergespiegelt wird. Lassen Sie uns dieses Konzept weiter untersuchen. Das Modell enthält die Daten und stellt Methoden bereit, um sie zu ändern. Außerdem wird eine Methode zum Anlegen von Binding-Instanzen zu den Daten bereitgestellt. Die Binding-Instanz enthält nicht nur die Binding-Informationen, sondern auch Ereignisse, die ausgelöst werden, wenn die gebundenen Daten geändert werden. SAPUI5-Controls verwenden das Data-Binding, um das Modell, das die Daten enthält, an das Control zu binden. Wenn Daten geändert werden, werden die Controls automatisch aktualisiert, und umgekehrt. Wenn eine Control-Änderung eine Änderung der Daten auslöst, wird diese Änderung auch automatisch im Modell aktualisiert, was als *Two-Way-Binding* bezeichnet wird. Mithilfe von Data-Binding können Sie Modelle, Views und Controller in unabhängige Teile entkoppeln. Diese Trennung ermöglicht eine bessere Lesbarkeit, Wartbarkeit und Erweiterbarkeit. In SAPUI5 kann ein View nur einen Controller haben, aber Controller können auch ohne View existieren. Diese Controller heißen *Application Controller*. Es gibt auch Views ohne Controller, was jedoch ein statisches UI bedeuten würde. Views können ihre eigenen Modelle haben, sie von einem übergeordneten Modell erben oder überhaupt kein Modell haben.

Wie Sie bereits gesehen haben, enthalten Modelle die Daten der Anwendung, aber darüber hinaus stellen Modelle auch Methoden zum Abrufen und Aktualisieren der Daten bereit. SAPUI5 unterstützt die folgenden Modelltypen (jeweils mit eigenem Zweck):

- **OData-Modell**
  Das Modell `sap.ui.model.odata` wird verwendet, um Controls an Daten aus OData zu binden. Hierbei handelt es sich um ein REST-basiertes offenes Protokoll zum Abfragen und Ändern datenbasierter Services. Das Modell wird in den Versionen v2

und v4 des OData-Protokolls ausgeliefert. Dieses Modell wird häufig verwendet, um Daten aus Backend-Systemen an UI-Controls zu binden.

- **JSON-Modell**
  Das Modell `sap.ui.model.json.JSONModel` ist ein clientseitiges Modell, das verwendet wird, wenn Sie JSON-Objekte an die UI-Controls binden möchten. Dieses Modell wird häufig mit kleinen Datenmengen auf dem Client verwendet.
- **XML-Modell**
  Das Modell `sap.ui.model.xml.XMLModel` ist auch ein clientseitiges Modell, das auch für kleine Datenmengen gedacht ist, aber nicht häufig verwendet wird.
- **Ressourcenmodell**
  Anwendungen auf Unternehmensebene erfordern Texte, die auf der bevorzugten Sprache des angemeldeten Benutzers basieren, und das Modell `sap.ui.model.resource.ResourceModel` stellt diese Funktion für Texte bereit, die als Teil der UI-Anwendung in der Datei **i18n.properties** ausgeliefert werden. Da diese Texte statisch sind, unterstützt das Ressourcenmodell nur einmaliges Binding.

Anwendungen können nicht nur eine beliebige Anzahl von Modellen haben, sondern auch verschiedene Bereiche einer Anwendung können unterschiedliche Modelle mit Schachtelung haben. Clientseitige Modelle werden vollständig geladen, aber das Servicemodell, bei dem es sich um das OData-Modell handelt, lädt nur die Daten, die von der Benutzeroberfläche angefordert werden. Der View-Teil des MVC-Konzepts ist alles, was Benutzer sehen können, aber der View könnte auch einige ausgeblendete Elemente auf der Benutzeroberfläche enthalten. SAPUI5 unterstützt die folgenden vordefinierten View-Typen:

- **XML-Views**
  Der View `sap.ui.core.mvc.XMLView` ist im XML-Format definiert, und der Dateiname endet auf **~.view.xml**. Dieser View-Typ unterstützt eine Mischung aus XML und purem HTML.
- **JSON-Views**
  Der View `sap.ui.core.mvc.JSONView` ist im JSON-Format definiert, und der Dateiname endet auf **~.view.json**.
- **HTML-Views**
  Der View `sap.ui.core.mvc.HTMLView` wird durch deklaratives HTML definiert und unterstützt auch eingebettetes HTML. Der Dateiname endet auf **~.view.html**.
- **Typisierte Views**
  Typisierte Views werden programmatisch durch Erweiterung der Klasse `sap.ui.core.mvc.View` konstruiert. Typisierte Views werden für JSON- und HTML-Views nicht unterstützt.

Wir empfehlen die Verwendung von XML-Views, die eine klare Trennung der UI-Definition von der Anwendungslogik erzwingen, die im Controller implementiert werden muss. Diese Trennung macht Ihren Code lesbarer und somit pflegbarer.

Ein Controller steuert den Datenfluss in das Modell und aktualisiert den View, wenn sich das Modell ändert. Sie können einen einfachen Controller definieren, so wie in Listing 3.13. Der Dateiname muss auf **~.controller.js** enden.

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

  return Controller.extend('sap.cleanui5.demo.Sample', {
    // Controller Logic
  });
});
```

**Listing 3.13** Definition eines einfachen Controllers

Ein SAPUI5-Controller hat die folgenden Lebenszyklus-Hook-Methoden:

- **onInit**
  Diese Methode wird nur einmal ausgelöst, wenn ein View instanziiert wird; sie kann verwendet werden, um einen View zu ändern, bevor er gerendert wird.
- **onExit**
  Diese Methode wird nur einmal ausgelöst, wenn der View zerstört wird.
- **onAfterRendering**
  Diese Methode wird aufgerufen, nachdem der View gerendert wurde.
- **onBeforeRendering**
  Diese Methode wird jedes Mal aufgerufen, wenn der View gerendert oder neu gerendert wird, auch bevor der Renderer aufgerufen und HTML im DOM-Baum platziert wird.

In Controllern ohne View (d. h. in Anwendungs-Controllern) werden Lebenszyklusmethoden nicht vom SAPUI5-Framework aufgerufen.

## 3.7 Zusammenfassung

In diesem Kapitel haben Sie verschiedene Projektarten für die Erstellung von SAPUI5-Anwendungen kennengelernt, und wir haben einige der beteiligten Artefakte beschrieben. Zusammenfassend lässt sich sagen, dass es mehrere Möglichkeiten gibt, eine Anwendung zu entwickeln, von Anwendungsvorlagen bis hin zum komplett

neuen Aufbau. Doch welcher Ansatz ist für Ihre Situation richtig? In den meisten Fällen entscheidet Ihr Fachwissen und/oder der Bedarf an Flexibilität und Freiheit.

Ein einfacher Ausgangspunkt für die Anwendungsentwicklung sind die Anwendungsvorlagen, die von SAP-Fiori-Werkzeugen angeboten werden (siehe *http://s-prs.de/v1005028*). Stellen Sie sich Anwendungsvorlagen als eine Art Best Practice für die Anwendungsentwicklung vor. Sie enthalten aktuelle Empfehlungen und können als Ausgangspunkt für die Entwicklung von Apps gemäß den Designrichtlinien für SAP Fiori dienen. Sie können die enthaltenen generischen Anwendungsfunktionen und Tests bei Bedarf einfach um benutzerdefinierte Funktionen erweitern.

Im nächsten Kapitel werden wir mit Clean-Code-Best-Practices für Module und Klassen fortfahren.

# Inhalt

# Der Clean-Code-Guide für SAPUI5

### Wichtige Konzepte

Verstehen Sie die Grundlage von SAPUI5: JavaScript. Entdecken Sie wichtige JavaScript-ES6-Funktionen und verschiedene Arten von SAPUI5-Projekten, von Bibliotheksprojekten bis zu Freestyle-Anwendungen.

### Best Practices

Lernen Sie die Regeln für Clean Code und wenden Sie sie sicher an. Erfahren Sie, wie Sie Variablen und Literale deklarieren, Funktionen strukturieren, Kontrollflussanweisungen und Loops verwenden und die Fehlerbehandlung angehen.

### Praktische Beispiele

Nicht nur reine Theorie: Profitieren Sie von den detaillierten Code-Beispielen im Buch, um sicher zwischen sauberem und unverständlichem SAPUI5-Code unterscheiden zu können und selbst zum Clean Coder zu werden.

## Auf einen Blick

- JavaScript
- Module und Klassen
- Funktionen
- Namensgebung
- Variablen und Literale
- Kontrollfluss
- Fehlerbehandlung
- Formatierung
- Statische Code-Prüfung
- Tests
- Implementierung

»Werden Sie zum Clean Coder – jetzt auch in SAPUI5!«

**Das Autorenteam**
Daniel Bertolozi, Arnaud Buchholz, Klaus Häuptle, Rodrigo Jordão, Christian Lehmann und Narendran Natarajan Vaithianathan verfügen über jahrzehntelange Erfahrung mit SAPUI5 und JavaScript als Softwareentwickler, Architekten und Entwicklungsexperten.

Rheinwerk Publishing

Gedruckt in Deutschland
Mineralölfreie Druckfarben
Zertifiziertes Papier

**€ 89,90 [D]** € 92,50 [A]
ISBN 978-3-367-10050-7