Coding Guidelines – Quick UI5

Von Experten – Für Experten

Coding Principles

DRY – Don't repeat yourself

Das Pinzip DRY ist besonders wichtig in großen Projekten und sollte daher schon von Anfang an beachtet werden. Aber was genau besagt dieses DRY?

Wie der englische Name **Don't repeat yourself** andeutet, sollte man Aufgaben und Codezeilen nicht unnötig oft wieder neu implementieren. Hier ist auch auf die Wiederverwendbarkeit des Codes zu achten. Angenommen man sieht sich die **onInit**-Funktion eines Detail-Controllers an. In fast jedem Fall gibt es einen Aufruf des **Routers**, um, falls das Routingpattern stimmt, eine gewisse Funktion auszuführen. Jedoch muss immer wieder Code geschrieben werden, um den Router neu zu laden. Oder man hat 4 verschiedene Views mit Email-Eingabefeldern. Dann möchte man ja nicht in jedem Controller eine neue Funktion zur Validierung schreiben.

Das Problem des sich wiederholenden Codes wird meist mit einem sogenannten **BaseController** gelöst. Dieser nimmt die wichtigsten Arbeiten wie **getRouter, getModel, validateXYZ, setContentDensity**, etc. ab und erspart somit viel Code und erleichtert die Übersicht.

Single Responsibility

Dieses Prinzip ist leicht zu verstehen: Jede Klasse, jeder Controller, jede Funktion sollte einen **einzigartigen Aufgabenbereich** haben. Zum Beispiel: eine **Formatterklasse** formatiert nur Datumswerte, ein **PersonDetailController** gehört nur zu einer **PersonDetailView** und eine **setModel**-Funktion setzt nur ein Model und sonst nichts.

Open – Closed

Open-Closed bedeutet: Meine Applikation soll **offen/open** für **Erweiterungen**, aber **geschlosssen/closed** für **Veränderungen** sein. Dieses Prinzip sieht man in den UI5-Application Extensions. Die Basisapplikation bleibt unverändert, sie wird nur in einer neuen Applikation erweitert. Man sollte also die Applikation so planen, dass Erweiterungen leicht machbar sind. Hier spielen **IDs** und **Controller-Hooks** eine wichtige Rolle, denn mit diesen kann ich meine Applikation erweitern und die originale Applikation unverändert lassen.

XML

Namespaces

Wenn die Applikation wächst, werden auch Controls benötigt, die vielleicht nicht im Standart-Namensraum **sap.m** liegen. Angenommen, ich möchte einen **sap.ui.core.Title** verwenden. Also muss ich in meinem View-File in dem View-Controll den benötigten Namespace **sap.ui.core** implementieren. Die allgemeine Schreibweiße lautet: **xmlns:*kürzel*="*namensraum*".** In unserem Fall wäre dass **xmlns:core="sap.ui.core".**

```
1.  <mvc:View
2.      xmlns:core="sap.ui.core"
3.      xmlns:mvc="sap.ui.core.mvc"
4.      xmlns="sap.m"
5.      xmlns:l="sap.ui.layout"
6.      xmlns:f="sap.ui.layout.form"
7.      xmlns:html="http://www.w3.org/1999/xhtml"
8.      xmlns:layout="sap.ui.layout"
9.      xmlns:codeEditor="sap.ui.codeeditor"
10.     controllerName="at.clouddna.controller.Main">
```

Somit kann ich alle Controlls unter diesem Namensraum ansprechen. Angesprochen werden müssen diese wie folgt:

```
<core:Title text="Titel"/>
```

Aggregationen von Controls, die nicht im Standart-Namensraum liegen, müssen auch mit dem Kürzel angesprochen werden.

```
<f:SmartForm ....>
    <f:content>
        <Label .../>
        <Text  .../>
    </f:content>
</f:SmartForm>
```

Controls

Ein Control besteht aus Properties, Aggregations, Associations und Events. Der XML-Aufbau eines Controls sieht folgendermaßen aus:

```
<Control property1="Wert1" propertyN="WertN" event1="onEvent1”>
    <aggregation1>
    </aggregation1>

    <aggregationN>
    </aggregationN>
</Control>
```

Properties

Properties sind Werte, die das Control beschreiben und steuern. Properties werden entweder in der gelaufenen Anwendung angezeigt, steuern die Sichtbarkeit des Controls und Eigenschaften, bestimmen das Binding, etc. Die häufigsten Properties sind:

- **id**
- **text**
- **value**
- **editable**
- **visible**

Jedes Control kommt mit einem eigenen Set an Properties. Da bei Controls in UI5 die Vererbungshierarchie oft sehr ausgeprägt ist und alle Controlls/Elemente von dem gleichen Basis-Control erben, haben sie oft überschneidende Properties wie **id** oder **text**.

Aggregations – Associations

Aggregation in UI5 beschreibt eine **Parent-Child**-Abhängigkeit, bzw. kennzeichnen einen bestimmten Bereich eines Controls.

Ein einfaches Beispiel ist das **Page**-Control. Dieses hat die Aggregations **headerContent, content** und **footer.** Jede Aggregation ist für einen bestimmten Teil der Page verantwortlich und ordnet die darin enthaltenen Controls dementsprechend an. Eine Aggregation kann entweder 1 oder 0 bis n Controls enthalten. Dies ist in der UI5-Docu zu jedem Control definiert.

```
<Page>
    <headerContent>
    </headerContent>

    <content>
    </content>

    <footer>
    </footer>
</Page>
```

Wenn ich basieren auf einem Template eine ungewisse Anzahl von Controls anzeigen möchte, bediene ich mich ebenfalls and den Aggregations. So hat beispielsweiße das List-Control eine Property namens **items** und eine dazugehörige Aggregation namens **items.**

```
<List headerText="Products" items="{/ProductCollection}">
     <items>
       <StandardListItem title="{Name}" counter="{Quantity}"/>
     </items>
</List>
```

Für jeden existierenden Eintrag wird somit ein neues StandardListItem, welches hier als Template dient, erzeugt und in die items-Aggregation eingefügt.

Mit **Associations** werden die Beziehungen zwischen Controls bezeichnet. Die einfachste Beziehung/Abhängigkeit ist die zwischen Labels und Texten.

```
<Label id="label_firstname" text="Vorname" labelFor="text_firstname"/>
<Text id="text_firstname" text="Max"/>
```

Events

Events lösen beim Eintreten eines bestimmten Ereignisses eine Funktion aus, die dem Event zugeordnet werden muss. Hier liegen im Hintergrund oft HTML-Events dahinter.

So hat zum Beispiel der Button ein **press**-Event, welches reagiert, was bei einem Andrücken des Buttons passieren soll.

```
<Button id="button" press="onButtonPress"/>
```

Die angegebene Funktion muss im dazugehörigen Controller ausprogrammiert werden und bekommt in diesem Fall ein **Event**-Objekt übergeben. Über dieses **Event**-Objekt lassen sich verschiedene Parameter auslesen, etwa die Source des Events mit **oEvent.getSource()**.

```
onButtonPress: function(oEvent){
   //do something
}
```

IDs

Jedes Control sollte eine ID zugewiesen bekommen. Hiermit wird ermöglicht, dass das Control eindeutig identifizierbar ist und im Controller gefunden werden kann. Auch um nachträgliche Erweiterungen zu ermöglichen, sollten IDs verwendet werden. Die einzige Ausnahme sind Controls, die in Aggregationen wie der items-Aggregation verwendet werden. Da diese mehr als einmal geladen werden, würde es hier zu einem Konflikt kommen, da mehrmals die gleiche ID verwendet wird.

Die Namenskonvention der ID-Vergebung ist frei wählbar, die ID sollte aber einen sprechenden Namen haben. Ein Beispiel: **viewname_controlart_text**, also **detailview_input_firstname**. Somit weiß man sofort in welcher View sich das Control befindet, um welche Art von Control es sich handelt und für was das Control zuständig ist. Wenn man sich das DOM ansieht, wird man merken, das das UI5-Framework die ID's noch zusätzlich erweitert.

Bindings

Um eine Verbindung zwischen View und Model herzustellen, müssen Daten an die View gebinded werden. In der View sprechen ich ein bestimmtes Binding mit **{…}** in einer Property an. Auf einen bestimmten/absoluten Eintrag eines Models greife ich mit **{/…}** zu. So wird der Eintrag direkt über einen **absoluten** Pfad angesprochen. Wenn bereits ein Binding auf ein bestimmtes Element in einem Control gemacht wurde, kann in den Child-Controls auf die Properties des Elements ohne **/** zugegriffen werden. Somit verweise ich **relativ**.

Element-Binding

Mit Element-Binding binde ich ein bestimmtes Daten-Element an ein Control. Besonders oft kommt dies bei Master-Details zum Einsatz. So wird zum Beispiel eine **Person** mit ihren **Personaldaten** auf eine VBox gebunden. In der VBox kann nun auf die einzelnen Personaldaten zugegriffen werden, ohne immer auf den direkten Pfad zu verweisen.

```
1.  {
2.      "Person": {
3.          "firstname": "Max",
4.          "lastname": "Mustermann",
5.          "gender": "male",
6.      }
7.  }
```

In der VBox wird nun auf das Element **Person** angesprochen und innerhalb wird auf die einzelnen Attribute zugegriffen. Das erste Text-Control bekommt so über Element-Binding die Firstname-Property des Person-Elements.

```
1.  <VBox id="vbox_person" binding="{/Person}">
2.      <Text text="{firstname}"/>
3.      <Text text="{lastname}"/>
4.      <Text text="{gender}"/>
5.  </VBox>
```

Das Binding kann auch per **.bindElement(…)** im Controller mitgegeben werden.

```
1.  let oVBox = this.getView().byId("vbox_person");
2.
3.  oVBox.bindElement("/Person");
```

Property-Binding

Mit Property-Binding kann direkt auf eine Property eines Elements gebunden werden. So kann per direktem Pfad zugegriffen werden und es muss nicht mit Elementbinding zuerst auf den übergeordneten Container gebunden werden.

```
1.  <Text text="{/Person/firstname}"/>
```

Aggregation-Binding

Mit **Aggregation-Binding** lässt sich eine Master-Child-Beziehung beziehungsweise eine Liste von gleichen Objekten darstellen. Beispiel:

```
1.{
2.   "Person": {
3.       "firstname": "Max",
4.       "lastname": "Mustermann",
5.       "gender": "male",
6.       "children": [{
7.           "firstname": "Heike",
8.           "lastname": "Mustermann"
9.       },
10.      {
11.          "firstname": "Martin",
12.          "lastname": "Mustermann"
13.      }]
14.  }
15.}
```

Wenn man nun die **children** in einer Liste anzeigen möchte, muss man per Aggregation-Binding darauf zugreifen.

```
<List items="{/Person/children}">
   <items>
      <StandardListItem title="{firstname}" description="{lastname}"/>
   </items>
</List>
```

Somit wird für jedes **Children** in der Liste ein neues StandardListItem angelegt und die childen-Liste durchgelaufen. Das hier definierte StandardListItem dient als Template. Da mit dem Pfad „{/Person/children}“ bereits per items-Property auf die Liste gebunden wurde und das StandardListItem in der items-Aggrigation ist, kann dort das Attribut direkt angesprochen werden ohne **/**. (Element-Binding)

Expression-Binding

Mit Expression-Binding lassen sich in der View bestimmte Funktionen direkt im Binding ausführen. Expresion-Binding sieht folgendermaßen aus: **{= <expression>}**

Expression-Binding ist besonders sinnvoll, wenn man keinen dedizierten Formatter für einfache Funktionalitäten schreiben will. Auch kann auf Models per **${..}** zugegriffen werden, um einen Wert auszulesen und mit diesem zu arbeiten.

So kann zum Beispiel der Button-Type dynamisch zugewiesen werden oder ein Control sichtbar gemacht werden.

```
<Button type="{= ${editModel>/isHighlited} ? 'Emphasized' : 'Default'}"
        visible="{= ${editModel>/isButtonVisible}}"/>
```

Javascript

Klassen inkludieren

Wenn man in seinem Controller Controls oder andere Klassen verwenden möchte, müssen diese explizit angegeben werden. Angenommen man möchte die **MessageBox** aus dem **sap.m**-Namensraum im Controller verwenden. Dann muss diese im 1. Array des **sap.ui.define** als Pfad mitgegeben werden, und in der Funktion als Übergabeparameter angegeben werden. Controls in UI5 sind nichts anderes

als Module, die dynamisch zur Laufzeit geladen werden sollten. Und diese kann ich eben in dem define-Array laden.

```
1.  sap.ui.define([
2.      "sap/ui/core/mvc/Controller",
3.      "sap/m/MessageBox"
4.  ], function (Controller, MessageBox) {
5.      "use strict";
6.
7.      return Controller.extend("at.clouddna.controller.Main", {
8.      });
9.  });
```

Bei dem Inkludieren von eigenen Klassen im Projekt müssen die mit dem tatsächlichen Pfad angesprochen werden. Wichtig: Punkte durch / erstetzen.

Angenommen man implementiert einen BaseController als zentrale Klasse und ein Controller soll von diesem BaseController erben. Dann muss der BaseController zuerst mit dem tatsächlichen Pfad angesprochen werden, in der Funktion als Übergabeparameter übergeben werden und der Controller muss von dem BaseController erben.

```
1.  sap.ui.define([
2.      "at/clouddna/controller/BaseController",
3.      "sap/m/MessageBox"
4.  ], function (BaseController, MessageBox) {
5.      "use strict";
6.
7.      return BaseController.extend("at.clouddna.controller.Main", {
8.      });
9.  });
```

Variablenbenennung

Da in Javascript-Variablen nicht mit dem Typ der Variable initialisiert werden(Loose-Data-Typing), etwa wie in Java mit **double d = 1.1**, sonder Variablen mit dem Schlüsselwort **let** oder **var** deklariert werden, ist es oft nicht sofort zu erkennen, welchen Typ die Variable haben sollte. Daher wird folgende Namenskonvention empfohlen:

- **o**Variable für Objekte
- **s**Variable für Strings
- **a**Variable für Arrays
- **b**Variable oder **is**Variable für Booleans
- **f**Variable für Float
- **i**Variable für Integer
- **d**Variable für Dates
- **r**Variable für RegExp
- **fn**Variable für Funktionen
- **$** für jQuery

So etwa **sCustomerID** oder **isVisible**.

Funktionen

Controller bringen Standardfunktionen, sogenannte Lifecycle-Methoden, mit. Diese werden automatisch ausgeführt und müssen nicht extra implementiert werden.

- **onInit** – beim Initialisieren des Controllers

- **onBeforeRender** – bevor die View gerendert wird.
- **onAfterRender** – nachdem die View gerendert wird
- **onExit** – beim Verlassen des Controllers.

Diese können überschrieben werden. Meistens geschieht das in Verbindung mit Routing oder um Daten vor/nach dem Rendern zu ändern.

Eigene Funktionen können auch implementiert werden. Private Funktionen werden mit einem _ gekennzeichnet und sollten nur von der eigenen Klasse aufgerufen werden.

```
1.  saveFile: function(sFileName) {
2.     //does something
3.  }
4.
5.  _loadMetadataFromURL: function(){
6.     //does something private
7.  }
```

Models

In UI5 gibt es 4 verschiedene Standard-Models, um Daten zu speichern:

- JSONModel – Client Side
  - Als Basis des JSONModels stehen Javascript-Objekte. Diese lassen sich leicht erstellen und dienen zum Speichern von kleineren Datenmengen, wie z.B.: ein Input-Model zur Eingabe von Kundendaten.
- ODataModel – Server Side
  - Das ODataModel besteht aus einem OData-Service, der per Destination in der Cloud Platform eingebunden wird. Es dient zur Speicherung und Verwaltung großer Datenmengen, da der OData-Service aus einem Backend kommt und die Daten aus diesem bereitstellt. Unterstützt alle CRUDQ-Funktionalitäten.
- RessourceModel – Client Side
  - Dient zum Laden von Ressourcen, wie dem i18n-RessourceModel. Wird meist benutzt, um Daten nur einmal zu laden und wenn die Daten nicht zur Laufzeit verändert werden.
- XMLModel – Client Side
  - Speichert Daten als XML-Daten ab. Wird in der Paxis nicht oft verwendet.

Models können in Controllern, in der Component.js oder im manifest.json als Default- oder als Named-Model gesetzt werden und in der View angesprochen werden. Es gibt nur **1** Default-Model, es können aber **N** Named-Models existieren

```
1.  let oEditModel = new JSONModel({
2.          bEditMode: true
3.      });
4.
5.  this.getView().setModel(oEditModel, "editModel");
```

UI5 unterstützt mehrere Binding-Modes. Diese regeln, ob Änderungen die über das Binding in der View Auswirkungen auf das Model haben.

- **One-Way-Binding**
  - Daten werden zwar an die View gebunden und angezeigt, Änderungen der Daten über die View werden aber nicht automatisch ins Model eingetragen.
- **Two-Way-Binding**

    - Daten werden an View gebunden und können direkt aus der View geändert werden. Diese müssen aber bei einem ODataModel per **submitChanges** nach hinten ins Backend geschrieben werden, da sie zuerst nur lokal in den **pendingChanges** gespeichert sind.
- **One-Time-Binding**
    - Daten werden nur einmal geladen. Wird bei i18n-Ressource-Models verwendet, da die Daten zu Laufzeit nicht verändert werden.

Schleifen/Loops

Um Arrays von Daten durchzulaufen, gibt es in Javascript mehrere Möglichkeiten:

forEach

Mit der **forEach**-Schleife werden alle Elemente eines Arrays durchiteriert und einzeln angesprochen.

```
1.  let oPerson = this.getView().getModel().getData().Person;
2.
3.  oPerson.children.forEach(function(oChild){
4.     console.log(oChild.firstname);
5.  });
```

Die function wird so oft aufgerufen, wie es Elemente im Array gibt. Der Nachteil bei forEach ist, dass diese Schleife nicht manuell abgebrochen werden kann, beispielsweise durch ein **break**, nur durch eine Exception.

Spezielle forEach-Varianten sind:

- .**every**(function….)
    - Hat jedes Element einen bestimmten gleichen Wert?
    - true oder false zurückliefern.
- .**some**(function…)
    - Befindet sich ein Element mit einem bestimmten Wert im Array?
    - true oder false zurückliefern.
- .**find**(function…)
    - Liefert das erste Element zurück, dass einen bestimmten Wert hat.
    - Element zurückliefern.

while

Mit der **while**-Schleife werden Aktionen so oft wiederholt, solange ein bestimmter Fall zutrifft. Kann per **break;** abgebrochen werden.

```
1.  let oPerson = this.getView().getModel().getData().Person;
2.  let iIdx = 0;
3.
4.  while(iIdx < oPerson.children.length){
5.     console.log(oPerson.children[iIdx].firstname);
6.
7.     iIdx++;
8.  }
```

for

Mit der **for**-Schleife kann per Index und Länge des Arrays auf Daten zugegrifen werden. Hier besteht eine größere Möglichkeit des individuellen Durchlaufs von Arrays. Kann auch per **break;** abgebrochen werden.

```
let oPerson = this.getView().getModel().getData().Person;

for(let h=0; h < oPerson.children.length; h++){
   console.log(oPerson.children[h].firstname);
}
```

Der Vorteil von while- und for-Schleifen ist der, dass man sich im gleichen Kontext befindet. Es ist also möglich, innerhalb des Schleifenrumpfes **this** zu verwenden. Bei einer **forEach**-Schleife befinden wir uns jedoch in einem anderen Kontext. Dies ist auch bei asynchronen Callbacks ein Problem. Abhilfe schafft hier ein **.bind(this)**.

```
let oPerson = this.getView().getModel().getData().Person;

oPerson.children.forEach(function(oChild){
   let oView = this.getView();

   console.log(oChild.firstname);
}.bind(this));
```

If/Else

Mit **if/else** können gewisse Werte abgefragt werden. Wenn der Wert im **if**-Block zutrifft, wird der darunterliegende Codeblock ausfegührt, ansonsten der Codeblock des **else**-Blocks.

```
let oPerson = this.getView().getModel().getData().Person;

oPerson.children.forEach(function(oChild){
   if(oChild.firstname === "Martin"){
      console.log("Das Kind heißt Martin");
   } else {
      console.log("Das Kind heißt nicht Martin");
   }
}.bind(this));
```

Die wichtigsten Vergleichsoperatoren sind:

- ===
  - Prüft, ob der Wert und der Typ zweier Variablen gleich ist. == würde nur den Wert vergleichen.
- !==
  - Prüft, ob der Wert zweier Variablen einen unterschiedlichen Wert haben.
- >, <, >=, <=
  - Prüft, ob Wert 1 größer, kleiner, größer gleich oder kleiner gleich dem Wert 2 ist.

Frequently used Methods

get/setModel

Setzen und Zurückliefern eines Models im Controller.

```
let oModel = new JSONModel({...}):

//als default model setzen
this.getView().setModel(oModel);

//als named-model setzen
this.getView().setModel(oModel, "customModel");

let oDefaultModel = this.getView().getModel();
let oCustomModel = this.getView().getModel("customModel");
```

bindElement

Binden eines Wertes aus einem Model gegen die View.

```
let oModel = this.getModel();

this.getView().bindElement("/Person(guid'asd234-asdf234-dftg333-2234dt')");
```

i18n-Texte

Auslesen eines i18n-Textes aus dem Ressource-Model.

```
let oBundle = this.getOwnerComponent().getModel("i18n").getResourceBundle();

console.log(oBundle.getText("button.create"));
```

MessageBox

Ausgeben einer einfachen MessageBox.

```
MessageBox.show("Person 1 updated successfully");
```

navTo

Weiternavigation auf die Route „Detail“ mit dem Parameter „customerid“.

```
let oRouter = sap.ui.core.UIComponent.getRouterFor(this);

oRouter.navTo("Detail", {
    customerid: "1"
}, false);
```

getBindingContext

Auslesen des Bindings in einem onPress-Event eines Buttons.

```
onButtonPress: function(oEvent){
    let sCustomerPath = oEvent.getSource().getBindingContext().sPath,
};
```