Von Knockout zu AngularJS

Table of Contents

1. Einleitung

Das JavaScript-Framework Knockout.js konzentriert sich klar auf eine Aufgabe. Diese ist das Bereitstellen einer MVVM-Engine. Zusätzliche Funktionalitäten wie modularer Code oder clientseitiges Routing müssen durch weitere Bibliotheken hinzugefügt werden. Gerade für größere SinglePage Anwendungen sind zahlreiche weitere JavaScript-Libraries nach und nach hinzuzufügen. Wer auf Knockout.js als Technologie für eine SPA (Single-Page Application) gesetzt hat, kann durch den Zechnologie-Zoo mitunter etwas ernüchtert sein.

Im Kontrast hierzu steht AngularJS. Dieses Framework bietet einen viel größeren Funktionsumfang. Es sind viele Funktionalitäten vorhanden, die für eine homogene SPA-Architektur verwendet werden können. AngularJS schickt sich an "Marktführer" für SPAs zu werden und diese Position dank der Unterstützung von Google auch zu behaupten.

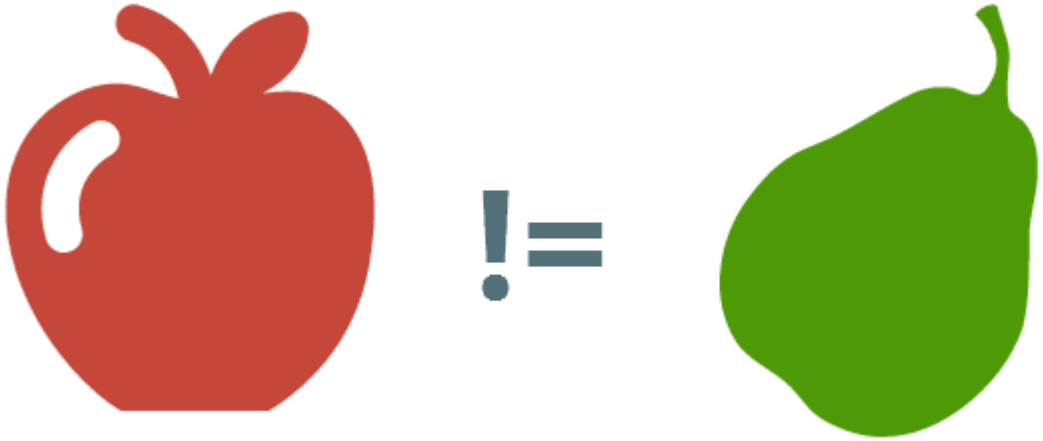
In dieser Session (und mit dem Ihnen hier vorliegenden Handout) werden Knockout und AngularJS miteinander vergleichen. Anhand **ausgewählter Schwerpunkte** sollen jeweilige Vorteile und Schwächen herausgearbeitet und mit Code-Beispielen belegt werden. Johannes Hoppe beleuchtet hierbei stets die Frage ob und wie ein Umstieg von Knockout auf Angular JS sinnvoll und machbar ist bzw. wäre.

2. Schwerpunkte

Bei einer SPA-Architektur geht es im stets darum, möglichst viel Kontrollfluss- und Rendering-Logik vom Webserver auf den Browser zu bringen. Der Webserver liefert im Idealfall nur noch eine einziges HTML-Dokument aus, welches dann die Kontrolle übernimmt. Prinzipiell kann man die verschiedensten Entwurfsmuster (Pattern) auf diesem einzelnen HTML-Dokument anwenden. In der Praxis zeigt sich aber, dass das MVC (Model-View-Controller) Pattern die bevorzugte Umsetzung ist. Diese Enwurfsmuster hat sich auf dem Server als Standard durchgesetzt. Es ist keine schlechte Idee, bewährtes auf den Browser zu übertragen. Ebenso bieten die meisten SPA-Frameworks eine MVVM (Model-View-ViewModel) Engine an. Diese beiden Prinzipien ergänzen sich gut. Durch ihre große inhaltliche Überschneidung bietet es sich an, beide Prinzipien in einen Topf zu werfen und kräftig umzurühren. Genauso sehen es auch die Macher von AngularJS, welche ihr Framework ganz pragmatisch ein MVW (Model-View-Whatever (works for you)) Framework nennen.

Unter der Prämisse, das wir uns im Kontext einer **MVW** Anwendung bewegen, ist es sinnvoll, folgende Schwerpunkte als Vergleichsgegenstand auszuwählen:

1. Bindings
2. Templating
3. Modularer Code
4. Routing



Es bleibt ein kleines Dilemma. Knockout.js und AngularJS sind eigentlich nicht miteinander vergleichbar. Wie Äpfel und Birnen haben beide Frameworks einen unterschiedlichen Schwerpunkt. AngularJS hat den Anspruch ein universales JavaScript-Framework für SPAs zu sein, Knockout hingegeben beschränkt sich hingegen darauf, eine MVVM Engine zur Verfügung zu stellen. Für einen fairen Vergleich lönnte man z.B. eher Durandal und AngularJS miteinander messen.

2.1. Bindings

Das folgende Beispiel basiert auf einem einfachem Formular, welches bei Wertänderung den Inhalt eines gelben Notizzettels verändert:

In den Zeiten vor MVVM waren einfache UI-Themen zuweilen sehr komplex. Folgender Sourcode demonstriert, wie z.B. nur mit jQuery der Notizzettel verarbeitet werden muss.

```
<script>
    $(function ($) {

        $('#title').change(function () {
            var title = $(this).val();
            $('#jQuery_output h1').text(title);
        });
```

```
        $('#message').change(function () {
            var message = $(this).val();
            $('#jQuery_output p').text(message);
        });

        $('#title').change();
        $('#message').change();
    });
</script>

<form>
    <label for="title">Title</label>
    <input id="title" value="Remeber">

    <label for="message">Message</label>
    <input id="message" value="the milk">
</form>

<div id="jQuery_output" class="sticky_note">
    <div>
        <h1></h1>
        <p></p>
    </div>
</div>
```

Demo

Man sieht, dass auf die einzelnen HTML-Elemente umständlich zugriffen werden muss. Eine deutliche Vereinfachung bietet hier MVVM.

Die Hauptaufgabe eine MVVM Engine besteht darin den **View** (welcher in unserem Fall reines HTML ist) möglichst elegant mit dem so genannten **ViewModel** zu verbinden. Das **ViewModel** kann man als einen speziellen Controller sehen. Er stellt einerseits Daten der Geschäftslogik bzw. des Models zu Verfügung und stellt weiterhin auch Methoden für diese dar. Durch die Zwischenschicht "ViewModel", werden View und Model voneinander getrennt. Es ist nun irrelevant wo und wie das tatsächliche Model existiert. Das ViewModel "versteckt es" und stellt eine standardisierte Sicht darauf her. Häufig wird es der Fall sein, das das eigentliche Model nur auf dem Server wirklich greifbar ist. Hierzu leitet dann das ViewModel alle Operationen per **AJAX** an den Server weiter.

Knockout

Diese Verbindung zwischen View und ViewModel nennt sich **Binding**, diese geht für gewöhnlich in beide Richtungen. Ändert sich das ViewModel, so wird der View aktualisiert. Ändert sich der Wert eines Interaktions-Elements (z.b. hier

eines Input-Felds), so wird das ViewModel ebenso geändert. Genau dies geschieht in folgendem Beispiel, welches mit Knockout.js umgesetzt ist.

```
<script>
    $(function () {

        var ViewModel = function () {
            this.title = ko.observable('Remember');
            this.message = ko.observable('the milk');
        };

        var viewmodel = new ViewModel();
        ko.applyBindings(viewmodel);

    });
</script>

<form>
    <label for="title">Title</label>
    <input id="title" data-bind="value: title">

    <label for="message">Message</label>
    <input id="message" data-bind="value: message">
</form>

<div class="sticky_note">
    <div>
        <h1 data-bind="text: title"></h1>
        <p data-bind="text: message"></p>
    </div>
</div>
```

Demo

Bei Knockout verwendet man für die Two-Way-Bindings Objekte vom Typ `Observable`. Diese implementieren (wie der Name bereits suggeriert), das Observer Pattern. Entsprechend dazu werden die Bindungs auf HTML-Elemente mit dem data-Attribut `data-bind` spezifiziert.

Angular

In AngularJS gestalten sich einfache Szenario recht ähnlich. Erfrischend ist jedoch die Tatsache, das noch weniger JavaScript geschrieben werden muss. Dies wird durch so genannte "Directives" / Direktiven ermöglicht. Direktiven sind Marker im HTML, welche dem HTML compiler (`$compile`) von AngularJS Instruktionen geben. Es wird dadurch eine sehr deklarative Beschreibung der Applikation möglich.

```
<body class="example" ng-app>

<form ng-init="model = { title: 'Remember', 'message': 'the milk' }">
    <label for="title">Title</label>
    <input id="title" ng-model="model.title">

    <label for="message">Message</label>
    <input id="message" ng-model="model.message">
</form>

<div class="sticky_note">
    <div>
        <h1>{{model.title}}</h1>
        <p>{{model.message}}</p>
    </div>
</div>

</body>
```

Demo
Demo2

In diesem Beispiel finden wir die Direktiven `ng-app`, welche eine Anwendung automatisch bereitstellt ("auto-bootstrap"), `ng-init`, welche Code ausführt (eval) und hier z.B. quick-and-dirty ein Model setzt und `ng-model`, welche den View und das Model per Two-Way-Binding verbindet.

Es fällt auf, dass das Model keine Observables implementieren muss. Hier unterscheidet sich Angular fundamental von Knockout. In der Dokumentation von AngularJS wird übrigens nicht zwischen "Model" und "ViewModel" unterschieden. Man spricht eher vom $scope der Referenzen auf ein oder beliebiges Model hält. (z.B. $scope.model) Dies ist konsequent, da man sonst keine **MVW** sondern eine **MVVM** Anwendung hätte. Man kann aber auch leicht argumentieren das der $scope klar ein ViewModel im Sinne von MVVM ist.

Best practices:

> *Das obrige Beispiel ist ein wenig unsauber, das wir (der Kürze wegen) direkt im View den $scope manipulieren. In diesem Fall fügen wir ein neu erstelltes Property "model" an den `$scope` hinzu. Der Scope sollte aber nicht im View verändert werden, dies ist Aufgabe des Controllers. Ein sauberes Beispiel finden sie in der zweiten Demo.*

Hinweis:
In der Welt von Angular besteht keine Regel zum Model. Das Model ist die

"Geschäftslogik" der Anwendung, die nicht im Einflussbereich von Angular liegen. Der $scope wird daher auch nicht als "die Geschäftslogik" verstanden. Der Scope hält stets lediglich Referenzen auf das Model, welches aus ein oder mehreren Objekten bestehen kann:

```
$scope.referenz1 = { test: 'xxx' };
$scope.referenz2 = { test: 'xxx' };
```

Durch diese Verwendung berüchstigen wir die bekannte AngularJS-Empfehlung, das in einem `ng-model` stets einen Punkt `.` verwendet werden muss. (z.B. hier `ng-model="referenz1.test"` So kommt es nicht zu Überraschungen, wenn bei primitiven Typen (wie dem String, dessen Werte kopiert wird) sich bei Two-Way-Bindings scheinbar das Model nicht verändern lässt. (siehe "Scope inheritance" in Understanding Scopes)

Ist ein Wechsel möglich?

Ein Austausch der MVVM Engines ist möglich, da AngularJS prinzipiell den Funktionsumfang von Knockout.js abdeckt und zusätzlich erweitert. Beide Engines verwenden den **DOM** als View, so dass nicht alles neu geschrieben werden muss. Die Direktiven können dabei helfen, die Anzahl an Code-Zeilen zu minimieren. Stolpersteine wird es definitiv durch den Umstand geben, dass ein AngularJS Model nicht "observable" ist. Dieses Prinzip nennt sich "**dirty checking**". Hinter den Szenen setzt AngularJS für jedes Binding eine so genannte `$watch` in eine Liste. (Info zu $watch und $digest) Die Watches werden verwendet um Änderungen zu erkennen. Hinzu kommen Standardfunktionalitäten wie `$timeout` oder `$http`, welche das dirty checking berücksichtigen. In den meisten Fällen werden Änderungen korrekt erkannt, aber intensive Tests sind notwendig um wirklich sicher zu sein.

2.2. Templating

In jeder Template-Sprache gibt es die Möglichkeit, repetitiven Code zu vermeiden. Im vorliegenden Beispiel bietet es sich z.B. an, den gelben Notizzettel auszulagern - damit dieser mehrfach verwendet werden kann.

Knockout

In Knockout kann man dies direkt über das `template`-Binding realisieren.

```
<form class="form_example">
```

```
    <label for="title">Title</label>
    <input id="title" data-bind="value: title">

    <label for="message">Message</label>
    <input id="message" data-bind="value: message">
</form>

<div data-bind="template: {
    name: 'sticky-note-template',
    data: {
        title: title,
        message: message
    }
}"></div>

<script type="text/html" id="sticky-note-template">
    <div class="sticky_note">
        <div>
            <h1 data-bind="text: title"></h1>
            <p data-bind="text: message"></p>
        </div>
    </div>
</script>
```

Demo

Angular

Eine gleichwertige Funktionalität kann man in AngularJs mit **Custom Directives** implementieren. Neben den bereits erwähnten Direktiven (z.B. `ngModel`) kann man durch einen einfachen Befehl eigene Direktiven spezifizieren. Bei der Gestaltungsfreiheit sind kaum Grenzen gesetzt, eine selbst erstellte Direktive kann auf einem DOM-Element, DOM-Attribut, einem CSS-Klassennamen oder einem Kommentar angewandt werden. Folgendes Beispiel verwendet ein DOM-Element "sticky-note", da der entstehende Quelltext so besonders einfach zu lesen ist.

```
<form class="form_example">
    <label for="title">Title</label>
    <input id="title" ng-model="model.title">

    <label for="message">Message</label>
    <input id="message" ng-model="model.message">
</form>

<sticky-note title="{{ model.title }}" message="{{ model.message }}"></s
```

```
.directive('stickyNote', function () {
```

```
    return {
        restrict: 'E',
        replace: true,
        scope: {
            title: '@',
            message: '@',
        },
        templateUrl: 'angular.tmpl.html'
    }
});
```

Die Direktive ersetzt alle Elemente welche "sticky-note" heißen und wendet hierbar das aus `templateUrl` stammende Markup an. Der Inhalt aus "angular.tmpl.html" entspricht dem zuvor verwendeten Code:

```
<div class="sticky_note">
    <div>
        <h1>{{title}}</h1>
        <p>{{message}}</p>
    </div>
</div>
```

Demo

Ist ein Wechsel möglich?

Der Wechsel von Knockout zu Angular sollte sich im Bezug auf exisistierende Ko-Templates relativ unproblematisch von Statten gehen. Es ist natürlich unerlässlich, existierende Ko-Bindungs auch hier zu portieren. Das Prinzip der Templates ist aber in beiden Frameworks vergleichbar und mit entsprechendem manuellen Aufwand ohne Überraschungen übertragbar. Im Vergleich zu den Templates von Knockout sind Angular-Direktiven viel flexibler, aber dennoch leicht und verständlich anzuwenden. Die guten Dokumentation zum Thema erleichert den Umstieg.

2.3. Modularer Code

Heutzutage sollte es Standard sein, Javascript-Code modular zu gliedern. Ein Modul kapselt zum einen Funktionalität und gibt zum anderen seine Abhängigkeiten bekannt. Module erlauben es, eine lose Kopplung zwischen Funktionalitäten zu erreichen - was allgemein als sauber Code gilt. Ein Modul-Loader kann dann diese Abhängigkeiten auflösen, sofern der Code das entsprechende Format implementiert. Im Browser hat sich das AMD (Asynchronous Module Definition) Format durchgesetzt, dessen

Referenzimplementierung stellt require.js dar. Weiter JavaScript-Loader sind unter anderem YepNope, $script.js, LABjs, headjs, der Loader vom Dojo Toolkit oder curl.js. Neben AMD, welches für Szenarien **im Browser** ausgelegt ist (asynchrones nachladen), ist CommonJS ein alternatives Format, welches vor allem von Node.js **auf dem Server** verwendet wird.

Knockout

Knockout verlangt nicht die Verwendung von Modulen. Es steht dem Entwickler völlig frei, das ViewModel oder den Initialisierungs-Code nach eigenen Vorstellung zu strukturieren. (Dies führt leider dazu, das man viel KO-Code findet, der überhaupt nicht strukturiert ist.) Wird jedoch Knockout zu einem Zeitpunkt ausgeführt, an dem entweder der CommonJs oder ein AMD-Loader (wie z.B. require.js) ausgeführt wurden, so präsentiert sich Knockout als **entsprechendes Modul**. (KO verwendet eine Variation des UMD (Universal Module Definition) patterns)

In folgenden Beispiel sieht man, wie ein ViewModel als Abhängigkeit Knockout angibt. Dies funktioniert ohne spezielle Anpassungen:

```
require(['jquery', 'knockout', 'domReady!'], function ($, ko) {

    var ViewModel = function () {
        this.title = ko.observable('Remember');
        this.message = ko.observable('the milk');
    };

    var viewmodel = new ViewModel();
    ko.applyBindings(viewmodel);

});
```

Demo

Anders als AngularJS übernimmt Knockout aber nie die Führung. Es versteht sich selbst als eines von vielen Modulen einer Applikation und überlässt es dem Entwickler eine mehr oder weniger modulare Archtitektur zu gestalten.

AMD ist nur ein klein wenig Dependency Injection

Das AMD-Pattern thematisiert vor allem die Isolation von Code und das Nachladen von Code als definierte Abhängigkeit (Dependency). AMD gibt jedoch keine Vorgaben darüber, was der Inhalt eines Moduls ist. Es herrscht die gleiche Freiheit, wie bei allen anderen JavaScript-Objekten. Man kann z.B. einfache

Key-Value Pairs oder komplexe Objekte als "Typ" des Moduls definieren. Dies ist jedoch auch die Crux an AMD: Module sind jede Art von JavaScript Objekt. Es ist erst nach Studium der entsprechenden Dokumentation (oder des Quelltextes) klar, ob

- a) das zurück gelieferte Modul Properties besitzt, die direkt verwendet werden sollen (sich das Modul also wie ein Singleton verhält),
- b) es als Funktion aufgerufen werden soll oder
- c) es sich um eine z.B. Konstruktur-Funktion handelt, welche mit dem `new` Schlüsselwort aufgerufen werden soll.

Man kann AMD/Require.js als puristischen Service Locator verstehen. Das Austauschen von Dependencies zu Testzwecken ist zwar möglich (wie z.B hier beschrieben) aber doch ein wenig umständlich. Man kann die Arbeit mit AMD eher als "Dependency Injection"-Light bezeichnen. Schließlich bekommt man auch nicht immer fertig instanziierte Abhängigkeiten, sondern muss die Module oder Properties/Teile des Moduls ggf. selbst erst instanziieren. Es fehlen schlicht strikte Vorgaben im AMD-Format, die es erlauben würden, das Modul bzw. Teile des Moduls direkt durch den Modul-Loader instanziieren zu lassen.

Angular

Bereits in den vorherigen Beispielen wurde modularer AngularJS-Code verwendet. AngularJS verwendet ein eigenes Modul-Format, bei dem Angular-Module durch den Befehl `angular.module()` erzeugten werden. Man muss darauf achten, dass es hier zwei völlig andere Konzepte auf einander treffen:

.

> ***AMD / require.js*** *regelt das (asynchrone) Laden von JavaScript-Code, welcher im AMD-Format vorliegt. Dies geschieht vor allem einmal zum Start der Anwendung. Ein Modul wird auch bei mehrfacher Verwendung nur einmal geladen. Damit ist jedes AMD Modul effektiv ein Singleton.*

.

> ***AngularJS-Module*** *konfigurieren mithilfe der verschiedenen Methoden des $provide-service den $injector, welcher zur Laufzeit ein fertiges Objekt zusammenbauen kann. Hierzu kann der $injector Typen instanziieren, Methoden ausführen und auch weitere Module laden. Das fertige Objekt beinhaltet einen oder mehrere* ***Services****. Services sind stets Singletons.*

In AMD erhält man ein Modul. Es gibt keine weiteren Vorgaben.

Bei Angular erhält man ein Modul, das mehrere **benannte** Services enthalten kann. Durch diese einzuhaltende Konventionen, kann man eine vollwertige "Dependency Injecton" - besonders im Sinne der Testbarkeit - erreichen. Was hierbei der Unterschied der verschiedenen $provide-Methoden (Service, Factory & Provider) ist, wird im Developer Guide ausführlich beschrieben. Kurz gesagt, folgende drei Provider erzeugen jeweils einen Service, der eine HelloWorld-Methode besitzt.

```
angular.module('exampleApp', [])

    .service('helloWorldService', function() {
        this.sayHello = function() {
            return "Hello World!";
        };
    })

    .factory('helloWorldFactory', function() {
        return {
            sayHello: function() {
                return "Hello World!";
            }
        };
    })

    .provider('helloWorldProvider', function () {

        this.$get = function() {
            return {
                sayHello: function() {
                    return "Hello World!";
                }
            }
        };
    })
```

Möchte man nun die Funktionalität der Services nutzen, so kann man diesen Service als Funktionsparameter akzeptieren:

```
angular.module('exampleApp')

    .controller('exampleController', function ($scope, helloWorldService
        $scope.hello = helloWorldService.sayHello();
    });
```

Demo

Vor dem Aufruf des Kontrollers prüft der $injector die Signatur der Funktion

("Reflection" per .toString()) und "injected" dann den gewünschten Service. Es ist demnach sehr wichtig, auf die exakte Schreibweise der Funktionsparameter zu achten. Eine vergleichbare komfortable Verwendung kennt AMD nicht.

Ist ein Wechsel möglich?

Wie sich gezeigt hat, sind AMD-Module und Angular-Module zwei Konzepte, die unterschiedliche Schwerpunkte setzen. Mit ein paar kleinen Anpassungen lassen sich beide Konzepte kombinieren. So kann AMD-Code aus der Knockout-Welt in die Angular-Welt überführt werden. Dies ist dringend empfohlen, denn es sehr unpraktikabel bzw. fehleranfällig synchron und asynchron ausgelegten Code miteinander zu kombinieren. (Man setzt entweder ganz auf AMD oder gar nicht auf AMD!)

Man kann jedoch nicht mehr die `ng-app` Direktive verwenden, da diese im Zuge des automatischen Bootstrappings bereits beim Browser-Event `DOMContentLoaded` ausgewertet wird. Zu diesem Zeitpunkt sind die asynchronen Module aber noch gar nicht geladen.

Daher sollte man das Bootstrapping erst dann starten, wenn require.js alle Module geladen hat. Hierfür bietet sich das Standard-Modul `documentReady` an. Nun ist es ohne Probleme möglich, ein Angular-Modul durch ein AMD-Modul zu umhüllen:

```
// AMD Modul
define(['require', 'angular'], function (require, angular) {

    // Angular Modul
    angular.module('exampleApp', [])

        .directive('stickyNote', function () {
            return {
                restrict: 'E',
                replace: true,
                scope: {
                    title: '@',
                    message: '@',
                },
                templateUrl: 'angular.tmpl.html'
            }
        })

        .controller('exampleController', function ($scope) {

            $scope.model = {
                title: "Remember",
```

```
                message: "the milk"
            }

        });

    // manuelles Bootsrapping durch domReady Modul
    require(['domReady!'], function (domReady) {
        angular.bootstrap(domReady, ['exampleApp']);
    });

});
```

Demo

2.4. Routing

Knockout

Es fehlt noch ein Prinzip, welches für eine SPA unerlässlich ist: **Client-side Routing**.
"Routing" bedeutet, dass die Anwendung zwischen Ansichten wechseln kann und dabei die Browser-History aktualisiert. Es wird dadurch möglich, den "Zurück"- und "Vor"-Button des Browser wie gewohnt zu verwenden. Ebenso sollte das Routing sicherstellen, dass man zu eine beliebigen Ansicht springen kann, indem man die entsprechende URL im Browser aufruft. Ist das Routing gut implementiert, ist für den Anwender nicht mehr ersichtlich, ob es sich um eine "klassische" Anwendung mit mehreren HTML-Seiten oder eine SPA handelt (wobei natürlich die Vorteile von Single-Page, wie z.B. schnelle Ladezeiten erhalten bleiben sollten).

Knockout bietet kein Routing von Haus aus an. Es bietet sich an, entweder auf Durandal zu setzen oder das Prinzip mit einem der vielen zur Verfügung stehenden Routing-Libraries selbst zu implementieren. Es bieten sich z.B. Sammy.js, FinchJS oder Director an.

Wie ein solides Routing mit Knockout selbst implementiert werden kann, ist Bestandteil folgender Dokumentation.

- WDC Kompakt 2013 - SPA Workshop - Part 3: Knockout Basics
- WDC Kompakt 2013 - SPA Workshop - Part 4: Knockout Bindings und Formularverarbeitung
- WDC Kompakt 2013 - SPA Workshop - Part 5: SPA Architektur mit Knockout (inkl. Routing)
  (Quelltext als Download)

Es läuft darauf hinaus, das man ein Modul entwickelt, welches den State hält (hier die Datei "appState.js" mit Sammy.js) und damit das Routing kapselt. Diese Routing-Datei kann dann eine weiteres Modul instruieren, je nach Route einen View und ein Viewmodel zu laden und diese mit Knockout auszuführen. Jenes Modul heißt hier schlicht "app", man könnte diese Modul auch "controller" nennen. So sieht diese zentrale Routing-Datei im vorliegenden Beispiel aus:

```
// singlePage/appState.js
define(['singlePage/app',
        'jquery',
        'sammy'], function (app, $, sammy) {

    var sammyApp;

    var init = function() {

        // Client-side routes
        sammyApp = sammy(function () {

            this.get('#/', function () {
                app.loadView('index');
            });

            this.get('#:viewId/:param', function () {
                app.loadView(this.params.viewId, this.params.param);
            });

            this.notFound = function() {
                app.loadView('page404');
            };

        }).run('#/');
    };

    var changeState = function (newViewId, newParam) {

        var newLocation = !newParam ? "#" + newViewId :
                                      "#" + newViewId + "/" + newParam;
        sammyApp.setLocation(newLocation);
    };

    return {
        init: init,
        changeState: changeState
    };
});
```

(siehe appState.js)

Angular

Das größte Problem am vorliegenden Knockout-Beispiel ist die unnötig hohe Komplexität und fehlende Standardisierung. Es wäre viel vorteilhafter, wenn das vorhandene MVVM-Framework bereits alle notwendigen Funktionalitäten anbietet. Genau dies bietet AngularJS.

.

> *Man könnte argumentieren, dass **Durandal** (das bekannteste SPA-Framework für KO) die notwendige Standardisierung liefert, aber Durandal wird Zugunsten von AngularJS nicht mehr weiter entwickelt!*

Angular setzt hier auf den $routeProvider, welcher die History überwacht und bei Bedarf ein Template lädt und den passenden Controller aufruft. Die Verwendung ist schnell ersichtlich. Man definiert eine `ngView` Direktive

```
<body ng-app="exampleApp" class="example">
    <div ng-view></div>
</body>
```

konfiguriert entsprechend den `$routeProvider`:

```
var hardcodedData = [
    { id: 1, title: "Remember", message: "the milk" },
    { id: 2, title: "DWX", message: "one great weak!" },
    { id: 3, title: "MDC kompakt", message: "one great day!" }
];

angular.module('exampleApp', ['ngRoute'])

    .config(function($routeProvider) {

        $routeProvider.when('/list', {
            templateUrl: 'templates/list.html',
            controller: 'listController'
        });

        $routeProvider.when('/detail/:id', {
            templateUrl: 'templates/detail.html',
            controller: 'detailController'
        });

        $routeProvider.otherwise({ redirectTo: '/list' });
    })

    .controller('listController', function ($scope) {
```

```
        $scope.listOfNotes = hardcodedData;
    })

    .controller('detailController', function ($scope, $routeParams) {
        var detail = _.find(hardcodedData, function(d) { return d.id ==
        $scope.detail = detail;
    });
```

und definiert eine passendes Listen-Template (templates/list.html):

```
<ul>
    <li ng-repeat="note in listOfNotes">
        <a href="#/detail/{{ note.id }}">{{ note.title }}</a>
    </li>
</ul>
```

Demo

3. Fazit

Ein Fazit fällt schwer. Es ist offensichtlich, dass Knockout allein sehr wenig Funktionsumfang bietet. Doch es ist durchaus möglich, das Sie bereits eine exzellente Architektur mit Knockout als MVVM-Engine besitzen. In diesem Fall sollte man bei Knockout bleiben, denn Knockout wird stetig weiter entwickelt und gepflegt. Von einem "Upgrade" auf Durandal würde ich seit Neuestem abraten, da Aufgrund der Konvergenz von Durandal und AngularJS keine neuen Innovationen bei Durandal zu erwarten sind.

Sollten Sie keine exzellente Architektur besitzen ("historisch gewachsen"), dann bietet es sich an auf AngularJS zu wechseln. Die Macher von AngularJS haben viele architektonischen Entscheidungen bereits für Sie gefällt. Durch die modulare Architektur und die gute Testbarkeit wird die neue Erzeugung von "schlechtem Code" hinreichend vermieden!

4. Downloads & Links

Sie finden dieses Dokument auf: http://bit.ly/Ko2NgDocs
Die Präsentation finden Sie hier: http://bit.ly/Ko2NgSlides

Die Beispiele sind in voller Länge auf Github verfügbar:
https://github.com/JohannesHoppe/FromKnockout2Angular
(Ordner: /Slides/examples/)