Einstieg in AngularJS

- DWX: 18.06.2015 09:00-17:00 Uhr

Sie planen ein neues Webprojekt? Stellen Sie sicher, dass AngularJS das richtige Framework für Ihren Einsatzzweck ist. In diesem intensiven eintägigen Workshop wird AngularJS anhand von praktischen Beispielen erläutert. Sie erfahren wertvolle Tipps und bekommen Best Practices für die Entwicklung einer Single-Page-Application auf Basis von AngularJS vermittelt.

Es werden unter anderem folgende Themen behandelt:

- MVC und Routing
- MVVM & Two-Way Binding
- Dependendy-Injection und Module
- Controller, Services und Direktiven
- Fehleranalyse und Debugging
- Zusätzliche Frameworks wie Bootstrap und Kendo UI

Lektion 1 - HelloWorld mit Modulen

Lektion 2 - Routing

Lektion 3 - Daten Laden und schick darstellen

Lektion 4... Ihr entscheidet!

(z.B. Unit Testing mit Jasmine/Karma oder JavaScript-Dateien per Bower?)

Rest API

Während des Workshops können wir Daten von folgenden REST-APIs beziehen:

http://workshop-angularjs.azurewebsites.net/

ODER

http://ex.extjs-kochbuch.de/help

Mini-Webserver:

Im Verzeichnis "*START*" befindet sich ein Express-Webserver für die Entwicklung.

Installation:

```
cd _START_
node -v
npm install
npm start
```

WICHTIG: Alle Quelltexte sowie dieses Dokument finden Sie unter:

https://github.com/JohannesHoppe/Workshop_AngularJS

Einstieg in AngularJS - Lektion 1: Hello World mit require.js

Hello World

AngularJS ist ein MVC Framework. Mittels der Directive `ngApp` wird hier das Modul "exampleApp" mit dem darin enthaltenen Controller "exampleController" ausgeführt. Hinter dem Befehl versteckt sich ein mehrstufiger Prozess, den AngularJS schlicht "Bootstrapping" nennt. Dies geschieht, sobald das HTML-Dokument komplett fertig geladen wurde (`DOMContentLoaded` Event).

Listing 1a -- HelloWorld.cshtml

```
<!DOCTYPE html>
<html>
<body ng-app="exampleApp">

    <div ng-controller="exampleController">
        <h1 ng-bind="model.text"></h1>
    </div>

    <script src="Scripts/angular.js"></script>
    <script src="Scripts/helloWorld.js"></script>
</body>
</html>
```

Listing 1b -- Die Datei helloWorld.js mit einem Angular-Modul

```
angular.module('exampleApp', [])
    .controller('exampleController', function($scope) {

        $scope.model = {
            text: 'Hello World'
        }
    });
```

Require.js

Um JavaScript-Dateien nicht mehr antiquiert über Script-Tags einbinden zu müssen, bedient man sich eines Modul-Loaders. Hierfür gibt es eine Reihe von Formaten und Frameworks. Als Defakto-Standard sollte man das "**Asynchronous** Module Definition (AMD)"-Format [1] kennen. Die Referenzimplentierung von AMD wird durch das Framework require.js [2] gestellt. Sollte das das eigene Projekt sowohl AMD als auch CommonJS-Module benötigen, so hilft curl.js [3] aus der Misere.

AMD ist schnell erklärt, da man prinzipiell nur zwei globale Methoden benötigt: `define` und `require`. Wie der Name vermuten lässt, definiert `define` ein AMD-Modul.

Listing 2a -- Die Datei myFirstModule.js im AMD-Format

```
define(['jquery'], function($) {
    var result = function() {
        $('body').text('Hello World');
    }
    return result;
});
```

Idealerweise befindet sich in einer JavaScript-Datei auch immer nur ein AMD-Modul. Folgt man dieser Konvention, so kann man ein anonymes Modul erstellen. Hier ergibt sich der Name des Moduls aus dem geladenen Dateinamen mit Pfad - bei dem Groß- und Kleinschreibung zu beachten sind!

Mit `require` kann man dieses Modul wieder anfordern und dessen Rückgabewert weiter verwenden:

Listing 2b -- myFirstModule verwenden

```
require(['myFirstModule'], function (myFirstModule) {
    myFirstModule();
});
```

Der `require`-Befehl akzeptiert ein Array aus Modulnamen, welche alle vollständig geladen sein müssen, bevor die angegebene Callback-Funktion ausgeführt wird. Durch den Callback wird die Definition von Abhängigkeiten und deren tatsächliche Bereitstellung zeitlich voneinander getrennt und die gewünschte Asynchronität komfortabel zur Verfügung gestellt. Im vorliegenden Beispiel ist der Rückgabewert des Moduls eine einfache Funktion, welche "Hello World" im Browser ausgibt. Bemerkenswert ist die Tatsache, dass es für Verwender des Moduls nicht von Belang ist, welche weiteren Abhängigkeiten benötigt werden. Wie zu erkennen ist, hat das "myFirstModule" nämlich selbst eine Abhängigkeit zum Framework jQuery. Es ergibt sich ein Graph von Abhängigkeiten, welche require.js in der korrekten Reihenfolge auflösen wird. Viele Frameworks wie etwa jQuery, Underscore oder Knockout.js bringen AMD-Unterstützung bereits mit, andere Frameworks lassen sich durch ein wenig Konfiguration (so genannte "Shims") als Modul wrappen. Dank der breiten Unterstützung und der Möglichkeit von "Shims" kann man nun Objekten im globalen Gültigkeitsbereich (einer sehr schlechten Praxis) ganz und gar den Kampf ansagen und dennoch die Komplexität der Lösung gering halten.

AngularJS mit Require.js kombinieren

AMD-Module und Angular-Module sind somit zwei Konzepte, die unterschiedliche Schwerpunkte setzen. Mit ein paar kleinen Anpassungen lassen sich beide Welten

kombinieren.

Zuerst muss die Directive `ng-app` entfernt werden, da sonst das Bootstrapping zu früh beginnen würde. Man darf nicht mehr auf `DOMContentLoaded` warten, welches bereits dann feuern würde, wenn die wenigen synchron geladenen Scripte bereit stehen würden. Dies ist im folgenden Beispiel lediglich require.js selbst. Es wird weiterhin fast immer notwendig sein, ein paar Pfade anzupassen und Shims zu setzen. Dies erledigt man mit dem Befehl `requirejs.config`. Anschließend kann die AngularJs Anwendung mittels `require()` angefordert werden.

Listing 3 -- HelloWorld.cshtml wird um require.js ergänzt

```
<!DOCTYPE html>
<html>
<head>
    <title>Hello World AMD</title>
</head>
<body>

    <div ng-controller="exampleController">
        <h1 ng-bind="model.text"></h1>
    </div>

    <script src="Scripts/require.js"></script>
    <script>

        requirejs.config({
            baseUrl: '/Scripts',
            paths: {
                'jquery': 'jquery-2.1.1'
            },
            shim: {
                angular: {
                    exports: 'angular',
                    deps: ['jquery']
                }
            }
        });

        require(['examples/exampleApp']);
    </script>
</body>
</html>
```

Leider hat sich durch die Konfiguration und den `require`-Befehl die Anzahl der Codezeilen im Vergleich zu Listing 1 erhöht. Doch zum Glück unterstützt require.js die Angabe eines einzigen Moduls direkt im script-Tag. Es bietet sich an, an dieser zentralen Stelle zunächst die Konfiguration selbst nachzuladen (hier "require.config" genannt) und anschließend die Anwendung anzufordern. So erhält man eine Lösung, die im Vergleich mit einer Zeile weniger auskommt.

Listing 4a -- HelloWorld.cshtml refactored

```
<!DOCTYPE html>
<html>
<head>
    <title>Hello World AMD</title>
</head>
<body>

    <div ng-controller="exampleController">
        <h1 ng-bind="model.text"></h1>
    </div>

    <script src="Scripts/require.js" data-main="Scripts/init"></script>
</body>
</html>
```

Listing 4b -- Die Datei init.js im AMD-Format

```
require(['require', 'require.config'], function (require) {
    require(['examples/exampleApp']);
});
```

Es fehlt zur Vervollständigung des Beispiels jene Datei für die Anwendung selbst. Laut Quelltext ist diese auf dem Webserver unter dem Pfad "/Scripts/examples/examplesApp.js" aufrufbar, beinhaltet ein AMD-Modul mit dem Namen "examples/exampleApp" sowie darin enthalten ein AngularJS-Modul mit dem Namen "exampleApp". Wie Sie sehen, müssen Namen in den beiden Modul-Welten nicht überein stimmen. Es liegt an Ihnen, für die Benennung und Verzeichnisorganisation passende Konventionen zu finden.

Listing 4c -- Die Datei exampleApp.js im AMD-Format mit Angular-Modul

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

    var app = angular.module('exampleApp', [])
        .controller('exampleController', function ($scope) {

            $scope.model = {
                text: 'Hello World'
            }
        });
```

```
    // bootstrap Angular after require.js and DOM are ready
    angular.element(document).ready(function () {
        angular.bootstrap(document, ['app']);
    });

    return app;
});
```

Ungelöst ist immer noch das Bootstrapping, welches nicht mehr über `ng-app` realisiert werden kann. Da ab dem Require-Callback alle notwendigen Dateien geladen sind, ist es an der Zeit das AngularJS-Bootsrapping mittels `angular.bootstrap` zu beginnen. Et voilà - die ersten beiden Zutaten aus unserem Technologiemix sind angerichtet!

Aufgaben

1. Die Anwendung zeigt lediglich "HelloWorld" an. Definiere ein Array mit Kunden-Objekten. (Werte: CustomerId, FirstName, LastName, Mail)
   Zeige diese Daten mittels ngRepeat [4] an!
2. Verwende ngClick [5] um den Namen eines Kunden zu ändern.

---

[1] AMD.Format: https://github.com/amdjs/amdjs-api/wiki/AMD
[2] Require.js: http://requirejs.org/
[3] Curls.js: https://github.com/cujojs/curl
[4] ngRepeat: https://docs.angularjs.org/api/ng/directive/ngRepeat
[5] ngClick: https://docs.angularjs.org/api/ng/directive/ngClick

---

Einstieg in AngularJS - Lektion 2: Routing

Routing mit angular-route

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).

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[1]

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

konfiguriert entsprechend den `$routeProvider`:

```
angular.module('app', ['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) {

        /* logik */
    })

    .controller('detailController', function ($scope, $routeParams) {

        /* logik */
        console.log($routeParams.id);
    });
```

Direktiven

Direktiven sind Marker im HTML, welche dem HTML compiler (`$compile`) von AngularJS Instruktionen geben. Es wird dadurch eine sehr deklarative Beschreibung der Applikation möglich.

Aufgabe

1. Erstelle eine Detailansicht für den Kundenmanager mittels Routing.
2. Vereinfache deinen Code mittels einer Direktive!

---

[1] ngView: https://docs.angularjs.org/api/ngRoute/directive/ngView

---

Einstieg in AngularJS - Lektion 3: Daten Laden und schick darstellen

Die Geschäftslogik

Alle Beispiele basieren auf einer simplen Geschäftslogik mit zwei Entitäten. Es gibt somit die Entität "Kunde", welche eine beliebige Anzahl an Rechnungen besitzen kann.

```
GET http://example.org/api/Customers
```

Dieser Web API Controller lässt sich über den `$http`-Service von AngularJS aufrufen. Der Service akzeptiert einen String oder ein Konfigurations-Objekt. Der Rückgabewert der Methode ist ein "promise"-Objekt, welches die Methoden "success" und "error" besitzt. Über diese beiden Methoden lassen sich Callbacks für einen erfolgreichen bzw. fehlerhaften Aufruf registrieren. Das Listings 1c zeigt den vollständigen Code, um Daten per `$http` zu laden.

Listing 1c -- listing1controller.js: AngularJS Controller fragt Daten per GET ab

```
define(['angular'], function(angular) {

    return angular.module('listing1', [])
        .controller('listing1Controller', [
            '$scope', '$http', function($scope, $http) {

                $scope.customers = [];

                $http.get('/api/Customers').success(function(data) {
                    $scope.customers = data;
                });
            }
        ]);
});
```

Die empfangenen Daten werden anschließend mittels `ng-repeat` und dem CSS-Framework Bootstrap [1] tabellarisch dargestellt (siehe Listing 1d).

Listing 1d -- listing1.html: AngularJS Template rendert Daten als Tabelle

```
<div class="table-responsive">
    <table class="table table-striped">
        <thead>
            <tr>
                <th>#</th>
                <th>FirstName</th>
                <th>LastName</th>
                <th>Mail</th>
            </tr>
        </thead>
        <tbody>
            <tr ng-repeat="customer in customers">
                <td ng-bind="customer.Id"></td>
                <td ng-bind="customer.FirstName"></td>
                <td ng-bind="customer.LastName"></td>
                <td><a ng-href="mailto:{{customer.Mail}}" ng-bind="customer.Mail"></a></td>
            </tr>
        </tbody>
    </table>
</div>
```

#	FirstName	LastName	Mail
1	James	Red	0@example.com
2	Harry	Black	1@example.com
3	Joseph	Magenta	2@example.com
4	Megan	Indigo	3@example.com
5	Ellie	Purple	4@example.com
6	Hannah	Indigo	5@example.com
7	Hannah	Indigo	6@example.com
8	William	Red	7@example.com
9	Lewis	Myrtle	8@example.com
10	George	Magenta	9@example.com

[Abb. 2] Die Tabelle aus Listing 1d im Bootstrap-Design

Tabellarische Daten mit OData anzeigen

So wie der Web API Controller aus Listing 1b implementiert wurde, wird ein Aufruf der Ressource ohne weitere Parameter eine Liste aller Entitäten zurückgeben. Es wird hierbei tatsächlich der gesamte Inhalt der Datenbank-Tabelle verwendet! Je mehr Daten vorhanden sind, desto unbrauchbarer wird dieser Ansatz. Das OData Protokoll [2] ist gut geeignet, um die Datenmenge enzuschränken. Die notwendigen Parameter heißen `$top` und `$skip`. `$top` gibt *n* Elemente der Ergebnismenge zurück. `$skip` überspringt *n* Elemente in der Ergebnismenge. Möchte man z.B. die Kunden mit der fortlaufenden Nummer 3 bis 7 abrufen, so verwendet man folgenden Aufruf:

```
GET http://example.org/odata/CustomersApi?$top=5&$skip=2
```

Weitere Query-Parameter sind unter anderem `$filter`, `$orderby`, `$count` oder `$search`.

Beispiele

- Metadaten Dokument abrufen: **/odata/$metadata**
- Paging: /odata/Customers?$top=5&$skip=2
- Paging mit Count: /odata/Customers?$top=5&$skip=2&$inlinecount=allpages
- Filtern: /odata/Customers?$filter=FirstName eq 'Hans'

Der Controller unterstützt nun eine seitenweise Ausgabe, Sortierung und Filterung. Diese Fähigkeiten direkt mit AngularJS umzusetzen wäre ein großer Aufwand. Es bietet sich an, ein fertiges Tabellen-Control ("Grid") zu verwenden. Auf dem Markt finden sich eine Reihe von freien und proprietären Grids, welche mit AngularJS kompatibel sind. Ein bekanntes und weit verbreitetes Framework ist Kendo UI von Telerik [3]. Listing 2c und Listing 2d zeigen die Verwendung des Kendo UI Grids im Zusammenspiel mit AngularJS und OData.

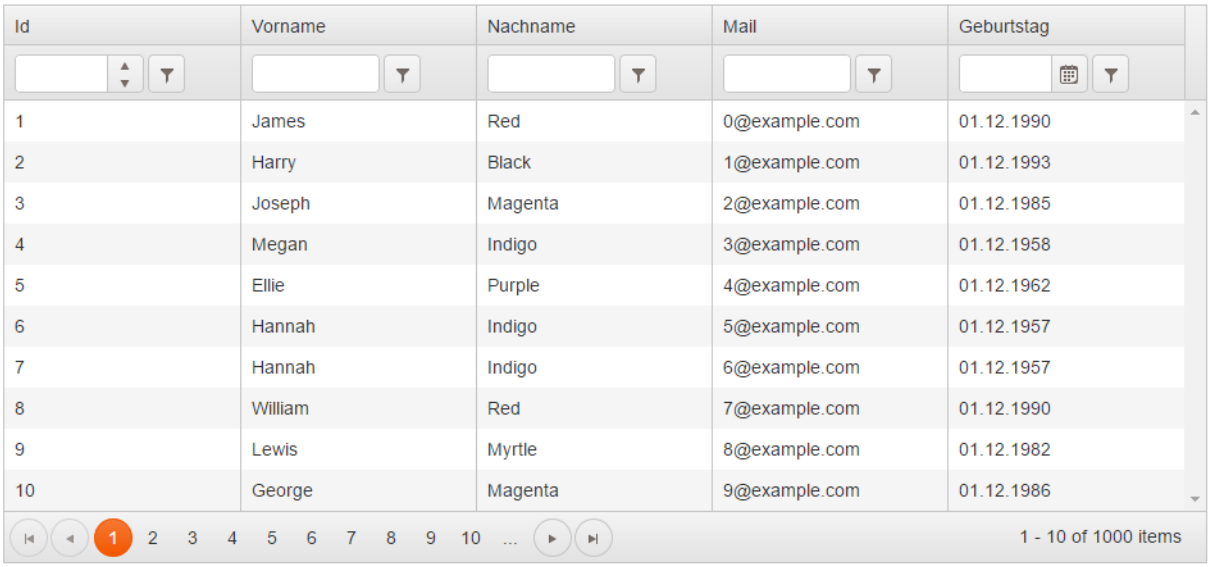
Listing 2c -- listing2controller.js: Die Datenquelle des Grids muss konfiguriert werden

```
define(['angular', 'kendo'], function(angular) {

    return angular.module('listing2', ['kendo.directives'])
        .controller('listing2Controller', [
            '$scope', function($scope) {

                $scope.customerDataSource = new kendo.data.DataSource({
                    type: 'odata',
                    transport: {
                        read: {
                            type: 'GET',
                            url: '/odata/Customers',
                            dataType: 'json'
                        }
                    },
                    schema: {
                        data: function (data) { return data.value; },
                        total: function (data) { return data['odata.count']; },
                        model: {
                            id: 'Id',
                            fields: {
                                Id: { type: 'number' },
                                FirstName: { type: 'string' },
                                LastName: { type: 'string' },
                                Mail: { type: 'string' },
                                DateOfBirth: { type: 'date' }
                            }
                        }
                    },
                    serverPaging: true,
                    serverSorting: true,
                    serverFiltering: true,
```

```
                pageSize: 10
            });
        }
    ]);
});
```

Listing 2d -- listing2.html: Eine AngularJS Direktive wrappt das KendoUI Grid-Control

```
<div kendo-grid
     k-data-source="customerDataSource"
     k-sortable="true"
     k-pageable="true"
     k-columns="[
         { field: 'Id' },
         { field: 'FirstName', title: 'Vorname' },
         { field: 'LastName', title: 'Nachname' },
         { field: 'Mail' },
         { field: 'DateOfBirth', title: 'Geburtstag', format: '{0:dd.MM.yyyy}' }]"></div>
```



Id	Vorname	Nachname	Mail	Geburtstag
1	James	Red	0@example.com	01.12.1990
2	Harry	Black	1@example.com	01.12.1993
3	Joseph	Magenta	2@example.com	01.12.1985
4	Megan	Indigo	3@example.com	01.12.1958
5	Ellie	Purple	4@example.com	01.12.1962
6	Hannah	Indigo	5@example.com	01.12.1957
7	Hannah	Indigo	6@example.com	01.12.1957
8	William	Red	7@example.com	01.12.1990
9	Lewis	Myrtle	8@example.com	01.12.1982
10	George	Magenta	9@example.com	01.12.1986

1 2 3 4 5 6 7 8 9 10 ... 1 - 10 of 1000 items

[Abb. 3] Das Kendo UI Grid aus Listing 2d

Im Kern ist Kendo UI ein Framework, welches aus diversen jQuery-Plugins besteht. Normalerweise ist die Integration von jQuery-Plugins in AngularJS mit Aufwand verbunden. Doch der Hersteller liefert über das AngularJS Modul `kendo.directives` gleich passende Direktiven für AngularJS mit. Die Datenquelle "customerDataSource" beschreibt das Modell und die Fähigkeiten des OData Services im Detail. Um die Übersichtlichkeit zu erhöhen, wurde die Datenquelle nicht im Markup konfiguriert. Man könnte übrigens in einem künftigen Refactoring-Schritt die Datenquelle in einen eigenen AngularJS Service auslagern.

Optional: Daten mit OData und Breeze.js abrufen

Nun gilt es, mithilfe von Metadaten und URL-Konventionen die Entwicklung eigener Funktionalitäten zu vereinfachen. Weder die Low-Level API von `$http`, noch das Angular-Modul `ngResource` sind dafür gut geeignet. Man benötigt ein Framework, welches die Komplexität von OData auf ein verständliches Niveau abstrahiert.

Die gesuchte Abstraktion bietet das Open-Source Framework "Breeze.js" an [4]. Für die OData Integration wird wiederum auf das Framework "data.js" [5] zurück gegriffen. Breeze.js verwendet zudem den internen Promise-Service `$q` von AngularJS, was Unit-Tests entscheidend vereinfacht. .NET Entwicklern wird Breeze.js sehr vertraut vorkommen. Das Framework ist stark vom Entity Framework und LINQ inspiriert. Das verwendete Modell ergibt sich stets aus den Metadaten. Konzepte wie "Change Tracking", das Unit of Work Pattern ("Batched saves"), "Navigation Properties" oder einen internen Speicher für Entitäten ("Client-side caching") sind aus dem Entity Framework bestens bekannt. Listing 4 zeigt, wie man alle Kunden mit dem Vornamen "Jack" komfortabel abfragt.

Listing 4 -- listing4controller.js: OData Service mit Breeze.js abfragen

```
define(['angular', 'breeze.angular'], function(angular) {

    return angular.module('listing4', ['breeze.angular'])
        .controller('listing4Controller', [
            '$scope', 'breeze', function($scope, breeze) {

                breeze.config.initializeAdapterInstance('dataService', 'webApiOData', true);
                var manager = new breeze.EntityManager('/odata');

                new breeze.EntityQuery()
                    .using(manager)
                    .from("Customers")
                    .orderBy("FirstName")
                    .where("FirstName", "eq", "Jack")
                    .execute()
                    .then(function(data) {
                        $scope.customers = data.results;
                    });
            }
        ]);
});
```

Ein interessantes Feature ist die Unterstützung von Navigation-Properties mittels "$expand". Folgendes Beispiel demonstriert, wie man den Kunden Nr. 42 und gleichzeitig all seine Rechnungen mit einem Aufruf lädt:

Listing 5 -- listing5controller.js: Verwendung von Navigation-Properties in Breeze.js (Ausschnitt)

```
new breeze.EntityQuery()
    .using(manager)
    .from("Customers")
    .where("Id", "eq", 42)
    .expand("Invoices")
    .execute()
    .then(function(data) {
        $scope.customer = data.results.length ? data.results[0] : null;
    });
```

Aufgaben

1. Verändere das Theme von Bootstrap!
2. Füge weitere Kendo-UI Controls zur Anwendung hinzu.

---

[1] Bootstrap: http://getbootstrap.com/
[2] OData Version 4.0 - URL Conventions - http://docs.oasis-open.org/odata/odata/v4.0/odata-v4.0-part2-url-conventions.html
[3] Kendo UI - http://www.telerik.com/kendo-ui1
[4] Breeze.js - http://www.breezejs.com/
[5] Data.js - http://datajs.codeplex.com/
