ANGULARJS UND ASP.NET, TEIL 1

Am Anfang war das Modul

AngularJS und der Microsoft Web Stack ergänzen sich ideal. Eine Auswahl von Entwurfsmustern und Frameworks hilft bei der Integration in die .NET-Anwendung.

Erste Schritte mit AngularJS sind schnell gemacht. Die Grundlagen dafür hat Golo Roden schon in der dotnetpro gelegt [1, 2]. Im zweiten Artikel verwendete der Autor auf dem Server Node.js mit dem Express-Framework. Der durchgängige Einsatz von JavaScript im Browser und auf dem Server ist jedoch nicht notwendig. AngularJS lässt sich ebenso gut mit Microsoft-Technologien kombinieren. Dies hat den Vorteil, dass vorhandenes Wissen sowie vorhandene Infrastruktur weiter verwendet werden können. Es kann auch sinnvoll sein, bestehende Webanwendungen auf Basis von ASP.NET Web Forms oder ASP.NET MVC mithilfe von Angular zu modernisieren. In den nächsten drei Ausgaben der dotnetpro zu AngularJS sei daher der Fokus wieder mehr auf die .NET-Welt gerichtet, wobei die folgenden Schwerpunkt-Themen betrachtet werden:

- das Laden von JavaScript-Dateien und das Verwalten von Abhängigkeiten
- asynchrone Datenübertragung per OData/Web-API
- Unit-Tests auf dem Server und im Client

Hierzu werden neben AngularJS drei weitere JavaScript-Frameworks vorgestellt:

- der Modul-Loader RequireJS
- das Ajax-Framework Breeze.js
- das Unit-Test-Framework Jasmine

In diesem Artikel wird das Framework RequireJS anhand einer fiktiven Webanwendung beleuchtet, die auf ASP.NET MVC mit Razor basiert. Alle Beispiele dazu lassen sich ohne großen Aufwand auch auf die ASPX-View-Engine oder mit ASP.NET Web Forms anwenden.

Modulares AngularJS

In der Oktober-Ausgabe von dotnetpro hat Golo Roden das modulare Prinzip von AngularJS vorgestellt [1]. Das folgende Beispiel in der Datei *HelloWorld.cshtml* knüpft daran an:

```
<!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>
```

Die in *HelloWorld.cshtml* eingebundene Datei *helloWorld.js* mit einem Angular-Modul:

```
angular.module('exampleApp', [])
    .controller('exampleController',
    function($scope) {
  $scope.model = { text: 'Hello World' }
});
```

Mittels des Attributs *ng-App* im Element *body* wird 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 (dies wird über das *DOMContentLoaded*-Ereignis angezeigt).

Modulares JavaScript

Das Einfügen von JavaScript-Dateien durch *<script>*-Elemente funktioniert tadellos. Ebenso verhält es sich mit der „Bundling and Minification"-Funktionalität aus dem *System.Web.Optimization*-Namensraum von ASP.NET MVC. Der Browser wird alle angegebenen JavaScript-Dateien oder Bundles synchron laden und das *DOMContentLoaded*-Ereignis auslösen, sobald alle Dateien und das Objektmodell, das DOM, verfügbar sind. Leider hat dieser klassische Ansatz eine Reihe von Nachteilen. Ein Nachteil ist, dass auch der gewünschte Output erst dann erscheint; in einer größeren Anwendung kann dies eine ganze Weile dauern. Die notwendige Reihenfolge der Skripte ist nur durch technisches Hintergrundwissen zu bestimmen – einer JavaScript-Datei ist nämlich nicht sofort anzusehen, welche Abhängigkeiten auf anderen Dateien bestehen. Aus demselben Grund ist es aufwendig und umständlich, ausschließlich nur die Skripte zu laden, die tatsächlich benötigt werden. Zu allem Überfluss macht das Einbinden von Skripten aus einem Content Delivery Network die Lösung noch komplexer.

Um JavaScript-Dateien nicht mehr antiquiert über *<script>* einbinden zu müssen, gibt es die sogenannten Modul-Loader, zum Beispiel Browserify, der das CommonJS-Format von Node.js verwendet [2]. CommonJS-Module sind jedoch nicht primär für eine asynchrone Verwendung im Browser ausgelegt. Man ist gut beraten, wenn man bei seinen Skripten von Anfang an eine asynchrone Verwendung vorsieht. Hierfür gibt es eine Reihe von Formaten und Frameworks. Als Defacto-Standard gilt das „Asynchronous Module Definition (AMD)"-Format [3], das Framework RequireJS von AMD als Referenzimplementierung [4]. Sollte das eigene Projekt so-