Warum AngularJS auch für Java-Entwickler interessant ist

Brave New JS World

AngularJS bringt für die Entwicklung der Frontends moderner Webanwendungen ein neues Paradigma. Vorbei sind die Zeiten, in denen das UI auf dem Server berechnet wurde und wir uns im Browser ständig mit niederen Aufgaben wie DOM-Manipulationen herumschlagen mussten. Die Zukunft gehört RIAs, deren Frontend komplett im Browser läuft und die mit Techniken wie Data Binding, Model View Controller und Dependency Injection arbeiten. Genau diese Features bringt AngularJS in den Browser.

von Philipp Burgmer

Wie wurden Webanwendungen bisher entwickelt und warum soll das jetzt nicht mehr gut genug sein? Warum ein Paradigmenwechsel? Nehmen wir zum Beispiel zwei im Java-Umfeld weit verbreitete Frameworks: GWT und JSF. Die Erfahrung zeigt, dass sich mit beiden Technologien mächtige Webanwendungen entwickeln lassen. Allerdings haben aus heutiger Sicht beide auch Nachteile.

In GWT läuft das komplette Frontend im Browser, wird aber in Java entwickelt – sowohl die Logik als auch die Darstellung. Das ist gerade für erfahrene Java-Teams ein großer Vorteil. Mit den neueren Versionen ist es zudem möglich, mit UiBinder das UI auch deklarativ mittels XML zu beschreiben. Eines aber bleibt: Wir entwickeln unsere Webanwendung in der Java-Welt bzw. in einer vorgetäuschten Java-Welt und nicht nativ in der Webwelt. Das hat einen großen Nachteil: Um neue Funktionen der Browser und neue JavaScript-Bibliotheken nutzen zu können, sind wir auf eine Integration in GWT angewiesen (sei es durch GWT direkt oder durch eine Bibliothek), da wir eben nicht die vom Browser nativ unterstützten Sprachen verwenden, sondern diese weggekapselt werden. Jüngst konnte man das gut an der Integration von HTML5-Features beobachten, die in GWT lange auf sich warten ließ.

Bei JSF sieht die Situation ein bisschen anders aus: Auch hier wird HTML gemischt mit eigenen Komponenten zur Beschreibung des UIs verwendet, und die Logik wird in Java implementiert. Die Nachteile bei JSF sind aber andere: Das UI wird auf dem Server berechnet und im Browser nur dargestellt. Fast immer, wenn im UI etwas passiert (z. B. wenn der User etwas anklickt), wird eine Anfrage zum Server geschickt, um das UI zu aktualisieren. Das kann bei schlechten Internetverbindungen, bei Anwendungen, die auch auf mobilen Geräten verwendet werden oder auch bei vielen Änderungen im UI zu einem echten Problem werden. Hinzu kommt, dass Informationen über den Client auf dem Server gespeichert werden müssen. Wir benötigen also ein Session-Management, was wiederum Nachteile mit sich bringt. Zusätzlich zur erhöhten CPU-Last durch die vielen Anfragen, um das UI zu aktualisieren, benötigen wir dadurch auch deutlich mehr Arbeitsspeicher. Skalierung der Server, ob horizontal oder vertikal, wird so nicht nur schwieriger, sondern auch schneller notwendig. Die Rechen- und Speicherkapazitäten moderner Endgeräte, seien es Smartphones, Tablets oder Laptops, werden komplett verschenkt.

AngularJS verfolgt einen völlig anderen Ansatz: Das UI wird ebenfalls deklarativ mit HTML und CSS beschrieben, die UI-Logik wird aber in JavaScript umgesetzt. Das Frontend läuft komplett im Browser. Das Backend weiß bei diesem Ansatz nichts über die Darstellung im Client und dessen Zustand. Da es sich bei Angular um Single-Page-Anwendungen handelt, können diese ihren Zustand selbst verwalten. Das Backend hat dabei zwei Aufgaben, die allerdings nicht zwingend vom selben Server erledigt werden müssen. Zum einen liefert ein Server beim Aufruf der Anwendung oder bei Bedarf auch später statische Dokumente, die der Browser für das Frontend benötigt. Diese Dateien ändern sich selten und können daher von spezialisierten Servern ohne viel Overhead ausgeliefert und vom Client gut gecacht werden. Die zweite Aufgabe des Backends ist es natürlich, die Businesslogik auszuführen und die Anwendungsdaten zur Verfügung zu stellen, beispielsweise über ein REST API. Angular macht dabei keine Vorgaben, wie das Backend auszusehen hat. Es kann also mit beliebigen Technologien umgesetzt werden oder auch ein anderes Paradigma als REST verfolgen.

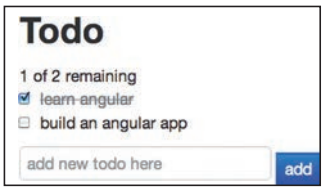

Abb. 1: Todo-List-Anwendung mit AngularJS und Bootstrap

Betrachten wir das Frontend als eine Art Rich Client, lassen sich HTML, CSS, JavaScript und auch Bibliotheken wie jQuery als Gestaltungsmittel für das UI ganz gut mit Java und SWT vergleichen: sehr low-level! Wer verwendet in Java nicht lieber eine höhere Abstraktion wie JFace oder gar die Eclipse RCP mit Funktionen wie Data Binding und Model View Controller? Genau hier springt AngularJS ein und bringt eine ganze Reihe von bewährten Konzepten in den Browser. AngularJS kapselt HTML und JavaScript nicht weg, sondern erweitert es um die Funktionen, die wir benötigen, um nicht nur statische Dokumente, sondern dynamische Anwendungen zu beschreiben. Und es bietet uns die Möglichkeit, HTML an unsere eigenen Bedürfnisse anzupassen.

Erste Schritte

Als „Hello World" der neu aufkommenden JavaScript-MVC-Frameworks wie AngularJS hat sich eine kleine To-do-List-App etabliert (**Abbildung 1** und [1]). Eine Angular-Implementierung wollen wir uns hier anhand des Codes in Listing 1 und Listing 2 etwas genauer anschauen. Für das Styling wird Bootstrap [2] verwendet. Der Code ist eine leicht angepasste Variante eines der vielen Beispiele auf der AngularJS-Website [3].

Als Erstes binden wir den nötigen JavaScript-Code ein, schön brav, als letzte Elemente im *body*-Tag (Zeile 15 und 16). Angular wartet, bis das Dokument fertig geladen ist und sucht dann nach dem Attribut *ng-app*, das in einem beliebigen Tag vorkommen kann, aber nur einmal pro Seite vorkommen sollte. Bei uns steht *ng-app* im *body*-Tag in Zeile 1. Angular verarbeitet nur den Teil des DOM, der durch dieses Attribut markiert ist. In diesem Element und allen seinen Kindern sucht Angular nach so genannten *Direktiven*. Das können Tags, Attribute oder CSS-Klassen sein – mehr dazu später. In unserem Beispiel sorgt das Attribut *ng-controller* in Zeile 1 dafür, dass für den *body* und seine Kinder eine Instanz des Controllers, der unter dem Namen *TodoCtrl* registriert wurde, verwendet wird (Listing 2, Zeile 2). Controller und View kommunizieren über eine Art View Model, in Angular *Scope* genannt, der dem Controller übergeben wird und auf dessen Properties von der View aus zugegriffen werden kann.

In Listing 1, Zeile 3 sehen wir zwei One-Way Bindings mittels der Notation {{}}. Im ersten Binding wird die Funktion *remaining* verwendet, im zweiten die Länge des Array *todo* abgefragt. *todo* ist genauso eine vom Controller im Scope definierte Property wie *remaining*, mit dem Unterschied, dass *remaining* eine Funktion ist (Listing 2, Zeilen 3 und 12). Da Funktionen in JavaScript Objekte erster Klasse sind, können sie wie Werte und andere Objekte als Referenz übergeben werden. Außerdem können Objekte dynamisch um Properties erweitert werden, wie wir es hier mit dem *Scope*-Objekt machen. In Listing 1, Zeile 5 wird mittels *ng-repeat* über alle Einträge in *todo* iteriert, um für jeden Listeneintrag mit einer Checkbox und einem Text auszugeben. Für jede Iteration legt Angular einen neuen Child Scope an und speichert den aktuellen Eintrag in der von uns definierten Property, hier *todo*. Da Scopes in AngularJS ausgehend vom Root Scope einen Baum bilden, der über JavaScripts Prototype-Mechanismus abgebildet wird, kann auch immer transparent auf Properties der Parent Scopes zugegriffen werden. In Listing 1, Zeile 10 wird ein Formular definiert, über das sich neue Todo-Einträge erfassen lassen. Das Textfeld ist mittels *ng-model* an die Property *todoText* gebunden. Bei *ng-model* handelt es sich um ein Two-Way Binding. Am Formular selbst wird mit *ng-submit* definiert, welche Funktion im Scope

Listing 1

```
<body ng-app="myModule" ng-controller="TodoCtrl">
  <h2>Todo</h2>
  <span>{{remaining()}} of {{todos.length}} remaining</span>
  <ul class="unstyled">
    <li ng-repeat="todo in todos">
      <input type="checkbox" ng-model="todo.done">
      <span class="done-{{todo.done}}">{{todo.text}}</span>
    </li>
  </ul>
  <form ng-submit="addTodo()">
    <input type="text" ng-model="todoText">
    <input class="btn-primary" type="submit" value="add">
  </form>
  <script src="angular.js"></script>
  <script src="listing2.js"></script>
</body>
```

Listing 2

```
angular.module('myModule', [])
.controller("TodoCtrl", function($scope) {
  $scope.todos = [
    {text:'learn angular', done:true},
    {text:'build an angular app', done:false}];

  $scope.addTodo = function() {
    $scope.todos.push({text:$scope.todoText, done:false});
    $scope.todoText = '';
  };

  $scope.remaining = function() {
    var count = $scope.todos.reduce(function(sum, todo) {
      var value = todo.done ? 0 : 1;
      return sum + value;
    }, 0);
    return count;
  };
});
```

aufgerufen werden soll, wenn das Formular abgesendet wird. Die Funktion *addTodo* (Listing 2, Zeile 7), die durch Drücken der Enter-Taste im Textfeld oder durch Klick auf den ADD-Button aufgerufen wird, liest den Wert aus *todoText* aus und erzeugt damit einen neuen Eintrag im *todos*-Array. Durch diese Änderung wiederum wird automatisch das UI aktualisiert. Zu beachten ist dabei, dass im gesamten Controller auf kein DOM-Element zugegriffen wird. Der Controller bzw. die Funktionen im Scope kennen die View nicht.

Module und Dependency Injection

Die Bedeutung und die Vorteile von Modularisierung und Dependency Injection sollten allen Entwicklern bekannt sein. Martin Fowlers Artikel zu DI [4] gehört meiner Meinung nach zur Pflichtlektüre. In Listing 3 sehen wir ein Beispiel sowohl für die Definition von Modulen als auch für DI in AngularJS. Module werden über *angular.module* registriert (Zeile 1) und können, wie am zweiten Parameter zu sehen, von anderen Modulen abhängen. Angular bietet dann über Method Chaining die Möglichkeit, Services, Controller, Direktiven usw. zu einem Modul hinzuzufügen (Zeile 2). Unser Service *usefulService* hat eine Abhängigkeit zum Service *$window*, den AngularJS mitbringt und der das globale *window*-Objekt per DI zur Verfügung stellt. Dieser Service wird bei der Erstellung unseres Services per DI bereitgestellt – genauso wie *usefulService* in *myModule* und alle Module, die von *myModule* abhängen. Unser Service besteht dabei lediglich aus einer Funktion (*return* in Zeile 3), die beim Aufruf das Fenster schließt (Zeile 4); sehr nützlich also.

Da JavaScript eine dynamisch typisierte Sprache ist, werden auch für Funktionsparameter keine Typen angegeben. Woher weiß AngularJS dann, was beim Funktionsaufruf bzw. bei der Service-Instanziierung injiziert werden muss? Angular verwendet hierfür Namen, sowohl bei der Registrierung als auch bei der Referenzierung. Dieser Mechanismus hat jedoch ein Problem: Codekompressoren ersetzen gerne die Namen von Funktionsparametern durch möglichst kurze, um die Dateigröße zu verringern (Zeile 2 und 4, Parameter *a*, eigentlich *$window*). Um dennoch die benötigte Information, den Namen der Abhängigkeit, zu bewahren, kann man statt der Funktion ein Array übergeben. Dieses Array enthält einen String für jeden Funktionsparameter sowie als letztes Element die Funktion selbst. Da die Namen der Abhängigkeiten jetzt als Strings im Code stehen (Zeile 2, erstes Array-Element), gehen sie bei der Kompression nicht verloren. Nachteil ist, dass man jeden Namen zweimal schreiben muss.

Testbarkeit

UI-Tests sind noch immer ein schwieriges Thema. Durch die Kombination von MVC und DI erreichen wir mit AngularJS bei Webanwendungen aber ein neues Niveau an Testbarkeit unseres UI-Codes. Schauen wir uns dafür noch einmal unseren *usefulService* in Listing 3 an. Wollen wir diesen Service testen und der Test läuft nicht in einem Browser, sondern in einer headless JavaScript-Testumgebung, steht höchstwahrscheinlich kein *window*-Objekt bereit. Da wir aber nicht das global verfügbare *window*-Objekt, sondern den AngularJS-Service *$window* verwenden, kann dieser sehr einfach durch einen Mock ersetzt werden. Dazu müssen wir lediglich im Modul *ng* den Mock unter dem Namen *$window* registrieren. Anschließend können wir beim Injector unseren *usefulService* erfragen, der dann den Mock nutzt. Ähnlich sieht es beim Testen von Controllern und somit der eigentlichen UI-Logik aus. Durch die strikte Vermeidung der Nutzung globaler Variablen und Service-Lookups oder gar hart kodierter Abhängigkeiten kann fast alles durch Mocks ersetzt werden. Da Controller zudem keinen Zugriff auf DOM-Elemente enthalten (sollten), sondern mit der View nur über den Scope kommunizieren, können auch diese ohne echten Browser und ohne UI getestet werden. Der Testcode kann dann den Zustand des Scope-Objekts und Aufrufe bei den Service-Mocks überprüfen. Zusätzlich kann natürlich weiterhin z. B. mit Silenium [5] das tatsächlich gerenderte UI samt dessen Verhalten getestet werden. Der notwendige Umfang dieser Tests und damit die Fehleranfälligkeit der Tests im Allgemeinen dürfte jedoch stark abnehmen. Außerdem können die Tests deutlich schneller ausgeführt werden, am besten gleich ständig während des Entwickelns.

Direktiven

Schauen wir uns zum Schluss noch ganz kurz die bereits mehrfach erwähnten Direktiven an. Direktiven können in AngularJS für zwei Arten von Aufgaben verwendet werden: UI-Komponenten und DOM-Manipulationen. Ein einfaches Beispiel für beide Aufgaben sehen wir in Listing 4 und Listing 5. Zwar wird wohl niemand den *blink*-Tag aus den frühen Zeiten von HTML ernsthaft vermissen, zumal man Text auch per CSS blinken lassen kann. Dennoch wollen wir *blink* nachbauen, um zu zei-

Listing 3

```
angular.module('myServiceModule', ['moduleOfOtherLibrary'])
.service('usefulService', ['$window', function(a) {
  return function() {
    a.close();
  }]
};
```

Listing 4

```
<body ng-app="app">
  <p>This is a paragraph with a <blink speed="500">blinking</blink> text inside.</p>
  <script src="angular.js"></script>
  <script src="listing5.js"></script>
</body>
```

Abb. 2: einfache Tabs mit eigener Direktive

gen, wie unsere Direktive jegliche DOM-Manipulation kapselt und wir HTML an unsere Bedürfnisse anpassen können. Blinkender Text ist gedruckt schwierig zu präsentieren, das Beispiel ist aber unter [6] live zu bewundern.

Wie in Listing 4, Zeile 2 zu sehen, stellt unsere Direktive einen Tag bereit. Dafür registrieren wir sie in Listing 5, Zeile 2 mit dem Tag-Namen an unserem Modul. Damit die Direktive als Tag verwendet werden kann, geben wir in Zeile 4 ein E (wie *Element*) an. Die Property *replace* sorgt dafür, dass unser Template den *blink*-Tag ersetzt. *transclude* und *ng-transclude* im Template sorgen dafür, dass der Inhalt des *blink*-Tag in das Template übernommen werden. In unserer Link-Funktion lesen wir zunächst das *speed*-Attribut aus (Zeile 9–15). Anschließend werden per *$timeout* zwei Funktionen zum Ein- bzw. Ausblenden abwechselnd aufgerufen. Fertig ist unser *blink*-Tag!

Ein sinnvolleres, aber für diesen Artikel leider zu langes Beispiel einer Direktive ist in **Abbildung 2** zu sehen. Bootstrap verwendet mehrere geschachtelte *div*-Tags mit bestimmten CSS-Klassen, um Tabs und deren Titel und Inhalt zu beschreiben. Das ist nicht nur schwierig zu merken, sondern auch unleserlich. Einfacher wäre es doch, wenn wir unsere Tabs wie in Listing 6 beschreiben könnten. Mit AngularJS und einer passenden Direktive ist das kein Problem.

Listing 5

```
1  angular.module('app', [])
2  .directive('blink', function($timeout) {
3    return {
4      restrict: 'E',
5      replace: true,
6      transclude: true,
7      template: '<span ng-transclude></span>',
8      link: function(scope, element, attrs) {
9        var speed = 1000;
10       if (attrs.speed !== undefined) {
11         var speedAttr = parseInt(attrs.speed, 10);
12         if (angular.isNumber(speedAttr)) {
13           speed = speedAttr;
14         }
15       }
16
17       $timeout(hide, speed);
18
19       function show() {
20         element.css("visibility", "visible");
21         $timeout(hide, speed);
22       }
23
24       function hide() {
25         element.css("visibility", "hidden");
26         $timeout(show, speed);
27       }
28     }
29   };
30 });
```

Listing 6

```
<tabs>
  <tab title="Tab 1">
    Just some conntent
  </tab>
  <tab title="Tab 2">
    Some other content
  </tab>
</tabs>
```

Fazit

Die im ersten Teil des Artikels aufgeführten Nachteile von GWT und JSF sind natürlich kein Grund, gleich alles über Bord zu werfen und mit AngularJS neu umzusetzen. Angular bringt aber viele, gerade im Java-Bereich etablierte Konzepte in den Browser und stellt somit einen Funktionsumfang bereit, mit dem sich erstaunlich schnell auch komplexere Frontends entwickeln lassen. Somit stellt es gerade für neue Projekte eine ernstzunehmende Alternative dar.

Da Angular nur den mit *ng-app* markierten Bereich verarbeitet, lässt es sich aber auch sehr gut mit anderen Bibliotheken kombinieren und sehr gezielt in ältere Webanwendungen einbauen. Bei der WeigleWilczek GmbH setzen wir AngularJS beispielsweise erfolgreich in einer recht alten JSP-/Struts-Anwendung ein.

In diesem Artikel konnte ich von den Konzepten und Möglichkeiten, die AngularJS bietet, natürlich nur einen Bruchteil vorstellen. Wessen Interesse ich damit wecken konnte, dem sei die AngularJS-Website [3] empfohlen, die eine sehr gute Dokumentation mit vielen direkt ausführbaren Beispielen enthält.

Philipp Burgmer arbeitet bei der WeigleWilczek GmbH im Bereich Software Development. Er beschäftigt sich überwiegend mit modernen Webanwendungen, neuen Frameworks wie AngularJS und User Interfaces im Allgemeinen.

Links & Literatur

[1] http://todomvc.com/

[2] http://twitter.github.com/bootstrap/

[3] http://angularjs.com

[4] http://martinfowler.com/articles/injection.html

[5] http://seleniumhq.org/

[6] http://jsbin.com/ekevip/latest/edit