JavaScript in professionellen Softwareprojekten: ein Erfahrungsbericht

Let's REST

In den letzten drei Monaten haben wir bei camunda erprobt, was es bedeutet, von JSF 2.0 auf REST und HTML5 umzusteigen. In diesem Beitrag ordnen wir die diversen Trends in diesem Bereich ein. Außerdem bieten wir eine knappe Einführung in REST und HTML5 und fassen unsere – überwiegend positiven – Erfahrungen mit diesem Architekturansatz zusammen.

von Andreas Drobisch, Nico Rehwaldt und Nils Preusker

Wir kennen die Fakten: Der Smartphone-Markt wächst, immer mehr Anwender verwenden primär Tablets statt Laptops, und wir können in den kommenden Jahren mit einem enormen Zuwachs beim Marktanteil von mobilen Endgeräten rechnen. Seitdem Mark Weiser in den frühen 90ern den Begriff „Ubiquitous Computing" prägte und als Chief Scientist bei Xerox PARC die ersten Tablets erprobte, sind diese mittlerweile wirklich „ubiquitous" – also allgegenwärtig – geworden. Und laut einer Studie von Cisco [1] wird der mobile Datenverkehr bis zum Jahr 2016 um das 18-fache steigen. Wir stehen also noch am Anfang eines Trends, den wir als Anbieter oder Entwickler von Webanwendungen nicht außer Acht lassen sollten.

Für uns als Java-Entwickler bedeutet das einerseits, dass immer mehr Arbeitgeber bzw. Kunden nach Anwendungen fragen werden, die auch auf mobilen Endgeräten bedienbar sind. Andererseits gewöhnen sich mehr und mehr Benutzer an den Bedienkomfort mobiler Apps und erwarten diesen künftig auch von geschäftlichen Webapplikationen, Firmenportalen und anderen typischen Einsatzgebieten von Java. Der User fordert, mit einem Wort, Usability.

Eine neue Webanwendungsarchitektur

Soviel zum Kontext, aber was bedeutet das technisch? Mittlerweile hat sich in der mobilen Welt und auch bei Desktopbrowsern HTML5 als Standard etabliert. Übrigens: **HTML5 = HTML + JavaScript + CSS3**. Struktur und Layout werden also mit HTML und CSS3 beschrieben, während wir das Verhalten mit JavaScript programmieren. Insofern hat sich mit HTML5 eigentlich nicht viel geändert, man versucht nur die Standards, die sich ohnehin mittlerweile etabliert haben, festzuschreiben und zu formalisieren.

Während im Frontend JavaScript dominiert, wird das Backend über REST (Representational State Transfer) angebunden, eine Technologie, die ja auch in der Java-Welt verbreitet und durch JAX-RS (Java API for RESTful Web Services) standardisiert ist. Um das Frontend wirklich im Griff zu haben und gleichzeitig leichtgewichtig zu bleiben, geht der Trend dahin, statt JavaScript zu generieren (wie bei JSF, GWT und Co.), die GUI-Logik in JavaScript selbst zu schreiben. Wer aber schon einmal mehr als 20 Zeilen JavaScript geschrieben hat, der weiß, dass die Codestruktur schnell zur Herausforderung wird. Abhilfe schafft da eine sich momentan herausbildende Kategorie von JavaScript-Frameworks, die häufig als Data-Binding- oder MVC-(Model-View-Controller-) Frameworks bezeichnet werden. Diese vereinfachen insbesondere die Erstellung so genannter Single-Page-Webapplikationen. Im Gegensatz zu klassischen Webseiten ähnelt der Aufbau dieser Anwendungen eher dem von Desktopanwendungen. Die Inhalte werden also zum größten Teil auf einer zentralen Seite dargestellt.

Auch in den Bereichen Frontend Design und Usability gibt es natürlich noch diverse interessante und wichtige Themen. Beispielsweise bieten Frameworks wie Twitter Bootstrap [2] einen guten Ausgangspunkt für die Entwicklung ansprechender Benutzeroberflächen, während sich Webdesigner mit Themen wie Responsive Design, CSS Grids und Styleguides beschäftigen.

Was sind eigentlich die Beweggründe dafür, sich plötzlich mit einer Sprache wie JavaScript zu befassen? Was sind die Vorteile des oben genannten Architekturansatzes gegenüber dem herkömmlichen Modell, im Java-Ökosystem Webanwendungen zu bauen? Um diesen Fragen auf den Grund zu gehen, wollen wir uns zunächst kurz mit den Problemen der herkömmlichen Java-Webentwicklung befassen.

JSF 2.0 als Ausgangspunkt

Während der letzten Jahre haben wir mehrere Webanwendungen entwickelt und dabei bisher auf JSF („Java Server Faces") als Frontend-Technologie gesetzt. Das lag in erster Linie an unserem Erfahrungsschatz in der Backend-Entwicklung. Nach Möglichkeit wollten wir auf vorhandenes Know-how setzen. Da unsere Anwendungen weitestgehend auf Java EE 6 basieren und intensiven Gebrauch von CDI („Context and Dependency Injection") machen, lag JSF als Teil des Java-EE-Standards nahe. Der zunächst wie „Plain old HTML" anmutende XHTML/Facelets-Ansatz kam uns gut handhabbar vor, und Komponentenbiblio-

theken wie Primefaces versprachen gute Produktivität. Schließlich beeinflusste auch ein gewisses Maß an Skepsis gegenüber JavaScript und CSS die Entscheidung für JSF. Die Anwendung, die wir jetzt neu konzipieren, hat einige anspruchsvolle Anforderungen zu erfüllen:

- Single-Page-Anwendung mit vielen Dialogen, d. h. sehr viel Ajax-Kommunikation
- Cross-Browser-Support (IE ab Version 8)
- Anbindung einer externen Webanwendung über ein proprietäres JavaScript-API
- Deployment sowohl auf Servlet-Containern wie Tomcat als auch auf verschiedenen Java-EE6-Application-Servern (JBoss AS, GlassFish, WebSphere etc.)

Insbesondere die Unterstützung verschiedener Application-Server war für uns ursprünglich ein Argument für JSF, schließlich setzten wir damit auf einen erprobten Teil des Java-EE-Standards. Interessanterweise entpuppte sich aber gerade dieser Aspekt im Laufe der Zeit als problematisch.

Der Standard als Problem

Viele Bestandteile des Java-EE-Standards werden durch Subprojekte implementiert, wodurch unterschiedliche Application-Server unterschiedliche Implementierungen dieser Bestandteile mitbringen. So beispielsweise das Data Binding, also die Verknüpfung zwischen Model und View, die JSF durch EL-(Expression-Language-)Ausdrücke unterstützt. Die EL-Ausdrücke werden beim Rendern der Seiten mit Daten ersetzt.

Das Problem ist nun, dass beispielsweise Methodenaufrufe nicht von allen Implementierungen einheitlich unterstützt werden. In der Praxis führt das dazu, dass bestimmte Basisfunktionalitäten nicht auf allen Application-Servern unterstützt werden und man als Entwickler gezwungen ist, entweder serverspezifische Releases zu erzeugen oder eine eigene EL-Implementierung mit auszuliefern.

Ähnliche Probleme erlebten wir mit der JSF-Implementierung selbst. Auch hier können Implementierungsdetails zu Problemen führen. In der Praxis erwiesen sich sogar Versionssprünge derselben JSF-Implementierung als problematisch. So stellten wir beispielsweise nach einem Server-Upgrade fest, dass einige Primefaces-Komponenten nicht mehr richtig funktionierten. Nach aufwändiger Fehlersuche stellte sich heraus, dass die Probleme auf Änderungen in einer Micro-Version der JSF-Implementierung zurückzuführen waren.

Eine andere Besonderheit von JSF ist die intensive Nutzung von HTTP POST für die Synchronisation zwischen Back- und Frontend. Für die Umsetzung mancher Features sahen wir uns bald unsichtbare POST-Formulare manipulieren, um die Daten des integrierten JavaScript-API ans Backend zu senden. Teilweise mussten wir auch proprietäre Features der Komponentenbibliothek nutzen, um unsere clientseitigen Dialoge mit Daten aus dem Backend zu füllen. Während wir ursprünglich so wenig wie möglich mit JavaScript in Berührung kommen wollten, sahen wir uns plötzlich an den unschönsten Stellen an schlecht dokumentiertem JavaScript-Code herumdoktern.

Darüber hinaus wurde schnell klar, dass vorgefertigte Komponenten nur auf den ersten Blick die Produktivität erhöhen. In der Praxis wurden sie schnell zum Zeitgrab. Die grafische Anpassung an die Anwendung gestaltete sich schwieriger als erwartet. Das Verhalten vieler Komponenten war nicht erwartungskonform und schlecht anpassbar. Nebenbei besteht bei JSF immer eine enge Kopplung zwischen Front- und Backend, was sich insgesamt negativ auf die Anwendungsarchitektur auswirkt.

Unterm Strich hatten wir also in fast allen Aspekten der Anwendungen mit Problemen zu kämpfen. Während im Browser eine scheinbar moderne Webanwendung mit HTML, JavaScript und CSS lief, war die Entwicklung aufwändig, setzte auf nicht mehr zeitgemäße Konzepte und führte zu schwer wartbarem und teilweise schlecht strukturiertem Code.

Anzeige

REST als Alternative

Schließlich bot sich mit der Neukonzeption einer unserer Anwendungen die Gelegenheit, einen Spike, also eine kurze Probeimplementierung mit klar umrissenen Problemszenarien und verschiedenen Frameworks durchzuführen, um den Architekturansatz REST und HTML5 zu erproben. REST-Architekturen sind ressourcenorientiert, jedes von außen zugreifbare Geschäftsobjekt wird also zur Ressource, die durch eine eindeutige ID, zum Beispiel einen URL (Uniform Resource Locator), adressierbar ist. Ein Geschäftsobjekt namens „Bestellung" wäre also beispielsweise unter dem URL *http://mydomain/myapp/bestellungen/123* erreichbar.

Ein weiterer Aspekt ist die Verknüpfung von Ressourcen untereinander. Rufen wir eine Bestellung über REST auf, so liefert diese beispielsweise den URL zu einer Person, die wir dann mit einem weiteren HTTP-GET-Request abrufen können. Ändert sich der URL der Person, so muss im Idealfall nicht der gesamte Client angepasst werden, sondern nur der Einstiegspunkt der Anwendung. Auf diese Art und Weise erreichen wir eine sehr dynamische Architektur, in der Änderungen potenziell weniger Aufwand verursachen als bei herkömmlichen Ansätzen.

Weiterhin werden bei REST die HTTP-Standardmethoden, im Webanwendungskontext hauptsächlich GET, PUT, POST und DELETE, verwendet. Ein wichtiges Un-

Java Script ist die Sprache des Webs – höchste Zeit, sie zu lernen!

Welche Merkmale also machen JavaScript so spannend und was sind die typischen Anwendungsszenarien? Zunächst setzt JavaScript sowohl Paradigmen aus funktionalen Sprachen als auch objektorientierte Ansätze um. Closures verdeutlichen den funktionalen Charakter, der für OO-Entwickler manchmal ein wenig gewöhnungsbedürftig ist:

```
function helloClosure(name) {
  var value = "Hallo ";
  return function(endChar) {
    return value + name + " " + endChar;
  }
}
var helloWorld = helloClosure("Welt!");
alert(helloWorld("Hansi"));
```

Die Funktion *helloClosure* gibt selbst eine Funktion zurück, die auf die lokalen Variablen der erzeugenden Funktion zugreifen kann. Auf diese Weise lassen sich beispielsweise sehr gut *Callback*-Methoden implementieren, was beispielsweise für die asynchrone Behandlung von Events praktisch ist. Gleichzeitig lassen sich aber auch Klassenhierarchien erstellen:

```
function Animal(name){
  this.hungry = true;
}

Animal.prototype.eat=function(){
  this.hungry = false;
}

Dog.prototype = new Animal();

Dog.prototype.constructor=Dog;

function Dog(name){
  this.name=name;
}

Cat.prototype.bark=function(){
  return "Wuff";
}
```

Das Standard-Interface für JavaScript besteht aus einer Sammlung von Methoden und Klassen zum Zugriff auf den „DOM-Baum", der standardisierten Objektrepräsentation des geladenen HTML-Dokuments. Über dieses DOM-API bietet JavaScript Zugriff auf Ereignisse und Funktionen. So können wir beispielsweise auf Klicks reagieren, aber auch die aktuell geladene Seite manipulieren, um etwa neue Elemente hinzufügen:

```
<!doctype html>
<html>
  <head>
    <script type="text/javascript">
      function helloFromScript() {
        var greeting = document.createElement("p");
        greeting.innerHTML="Hallo Welt!";
        greeting.classList.add("greeting");
        document.body.appendChild(greeting);
      }
    </script>
  </head>
  <body onload="helloFromScript()">
  </body>
</html>
```

Nach dem Laden des HTML-Dokuments wird in das zunächst leere *body*-Element ein Absatz mit dem Text „Hallo Welt" und der CSS-Klasse *greeting* eingefügt. Es gibt übrigens viele Bibliotheken, die einerseits Standardfunktionen noch stark vereinfachen und andererseits ein browserübergreifend einheitliches API zur Verfügung stellen. So lässt sich die gleich Funktionalität mit jQuery [6] in einer Zeile ausdrücken:

```
$('body').append('<p class="greeting">Hallo Welt</p>');
```

Auch ein mit Standard-JavaScript-Mitteln eher umständlicher HTTP-Request ist mit jQuery schnell erzeugt:

```
$.ajax({
    type: "GET",
    url: "http://mydomain/myapp/bestellungen/123",
  }).success(function (data) {
    $('body').append('<p>' + data.name + '</p>');
  });
```

Hier wird eine Bestellung von einem REST-API angefordert und anschließend ein Paragraph mit dem Namen der Bestellung in das *body*-Element der HTML-Seite eingefügt.

terscheidungsmerkmal sind dabei die Sicherheit (safety) und Idempotenz (idempotence) der HTTP-Methoden [3]. GET beispielsweise ist sicher (safe), weil es den Zustand serverseitiger Ressourcen nicht verändert. PUT und DELETE sind nicht sicher, da sie serverseitige Ressourcen erzeugen, ändern oder löschen können, aber idempotent, da Mehrfachausführungen zum gleichen Ergebnis führen. POST hingegen ist laut HTTP-Spezifikation weder sicher noch idempotent. Die Mehrfachausführung eines POST Requests kann serverseitig zu Duplikaten von Ressourcen führen. POST z. B. für das Absenden eines Suchformulars zu verwenden, wie das bei JSF-Anwendungen häufig der Fall ist, ist also offensichtlich nicht im Sinne des Erfinders! Wir sollten uns darüber im Klaren sein, dass REST-Frameworks diese Eigenschaften der HTTP-Methoden nicht erzwingen. Die Verantwortung für die korrekte Umsetzung liegt also bei uns!

Ein weiterer Aspekt bei REST ist, dass Ressourcen in unterschiedlichen Repräsentationen auslieferbar sind. Der Client teilt dem Server mit, welche Repräsentationen er erwartet oder verarbeiten kann. Das kann zum Beispiel HTML- oder Plain-Text sein. Bei JavaScript-Webanwendungen bietet sich JSON (JavaScript Object Notation) an, da es kompakt ist und im JavaScript-Code leicht verarbeitet werden kann.

Schließlich spielt die Zustandslosigkeit bei REST eine wichtige Rolle, was auch gerne mit der Abkürzung HATEOAS („Hypermedia as the engine of application state“) beschrieben wird. Der Anwendungszustand wird also, anders als bei JSF, clientseitig verwaltet, während der Zustand von Geschäftsobjekten serverseitig gehalten wird. Zustandsänderungen werden durch Hypermedia, also Repräsentationen der Geschäftsobjekte, an den Server übermittelt.

Insbesondere der letzte Punkt erklärt, warum sich REST so gut für die Umsetzung von Webanwendungen eignet. Das clientseitige Halten des Zustands sorgt für deutliche bessere Skalierbarkeit, erlaubt entkoppeltes Testen des Backends und sorgt nach Ansicht der Autoren auch für weniger zustandsbedingte Bugs im GUI.

Mittlerweile ist REST übrigens auch in der SOA-Welt angekommen, wo es sowohl als Architekturkonzept als auch für die technische Implementierung von Services zunehmend an Bedeutung gewinnt. Mehr dazu in [4] und [5].

Auch wenn einige der oben genannten Punkte für die Entwicklung von Webanwendungen nicht direkt relevant erscheinen, lohnt es sich dennoch, sich der REST-Prinzipien bewusst zu sein. Wie aber setzen wir diese Prinzipien nun in Java um?

REST mit Java EE

Mit JAX-RS können wir aus POJOs (Plain Old Java Objects) mithilfe weniger Annotationen REST Services machen. Zur Laufzeit wandelt JAX-RS eingehende HTTP-Requests in Serviceaufrufe um und übersetzt den Payload im empfangenen Request Body in Java-Objekte und umgekehrt.

Seit Java EE 6 ist JAX-RS Teil der Java-EE-Spezifikation, was für eine gute Verzahnung mit Technologien wie JPA („Java Persistence API“), EJB 3.0 („Enterprise Java Beans“) und CDI sorgt. Anwendungsentwickler, die verschiedene Application-Server unterstützen müssen, seien an dieser Stelle jedoch gewarnt: Während JAX-RS nämlich die Schnittstelle standardisiert, ist die Umwandlung zwischen Java-Objekten und den clientseitig angeforderten Repräsentationen (beispielsweise JSON) nochmals durch Provider gekapselt. Das führt dazu, dass je nach Application-Server unterschiedliche Konfigurationen notwendig sind, um konstant das gleiche Format auszuliefern.

JavaScript-Frameworks und -Architektur

Zur Strukturierung und besseren Wartbarkeit des Codes haben sich MVC [7] und verwandte Entwurfsmuster, die häufig als MV*-Patterns bezeichnet werden, durchgesetzt. Dabei werden Daten (Model), Präsentation (View) und Geschäftslogik (Controller) innerhalb einer Anwendung sauber voneinander getrennt. Das vereinfacht später nebenbei auch das Testen.

Anzeige

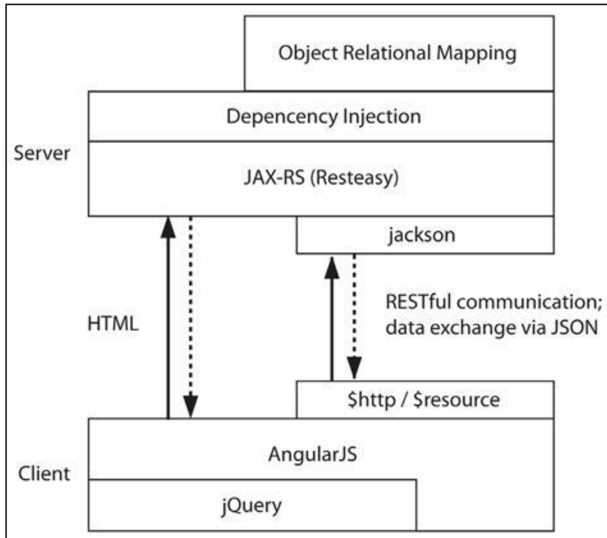

Abb. 1: Gesamtarchitektur der Webapplikation

Für JavaScript gibt es mittlerweile eine ganze Reihe von Bibliotheken, die bei der Anwendungsentwicklung auf MV*-Muster setzen. Dazu gehören Backbone.js, Knockout, Ember.js und AngularJS, um die bekanntesten zu nennen. Sie alle bieten eine bidirektionale Datenbindung zwischen JavaScript-Objekten und dem DOM.

Listing 1

```
<form ng-controller="MyFormController">
  <input ng-model="name" type="text" />
  <input type="submit" ng-click="reset()" value="Reset" />
</form>
<script type="text/javascript">
  function MyFormController ($scope) {
    $scope.name = "Klaus";
    $scope.reset = function() {
      $scope.name = "Klaus";
    };
  }
</script>
```

Listing 2

```
var users = User.query(),
    user;
if (users.length) {
user = users[0];
user.name = "Klaus";
user.$save();
```

Listing 3

```
function MyController($scope, $http) {
  $http.get("app/name").success(function(data) {
    $scope.name = data.name;
  });
}
```

Listing 1 zeigt, wie Data Binding in AngularJS funktioniert. Nachdem die Bindung im HTML-Markup deklariert wurde (siehe *ng-model="name"*), ist das Formularfeld im Controller *"MyFormController"* über den *scope*-Parameter verfügbar. Damit besteht die bidirektionale Bindung, Änderungen am Text des Eingabefelds sind direkt im Controller verfügbar und Änderungen am *$scope.name*-Attribut werden automatisch ins DOM propagiert. Auf gleiche Weise können Formularfelder auch an Methoden im Controller gebunden werden (siehe *reset()* in Listing 1).

Die Bibliotheken unterscheiden sich in erster Linie darin, wie das Data Binding umgesetzt wird und wie viel JavaScript und HTML wir als Entwickler noch schreiben müssen. Einen guten Überblick darüber liefert das Projekt TodoMVC [3], das Implementierungen einer To-do-Liste mittels verschiedener JavaScript-MV*-Bibliotheken bereitstellt. Abgesehen vom persönlichen Geschmack spielen natürlich in erster Linie die Fragen nach Produktreife, Qualität und Umfang der Dokumentation, Testbarkeit, Richtlinien bezüglich der Anwendungsarchitektur und Modularisierung sowie zusätzliche Features wie clientseitiges Routing, Templating und eine Abstraktion für REST-Ressourcen eine Rolle.

Nachdem wir diverse JavaScript MV*-Bibliotheken im Hinblick auf diese Fragen evaluiert hatten, fiel unsere Wahl auf AngularJS. Neben deklarativer Model-View-Bindung setzt es einen starken Fokus auf Testbarkeit. Zusätzlich bringt Angular umfangreiche Mechanismen zur Anwendungsmodularisierung mit, bietet clientseitige Dependency Injection, Routing und eine REST-Ressourcen-Abstraktion (Listing 2). Nebenbei verfügt AngularJS über eine ausgezeichnete Dokumentation, bietet grundlegende Richtlinien für die Anwendungsarchitektur und erfreut sich einer sehr aktiven und nach

Listing 4

```
describe("MyController", function() {
  var $httpBackend, $rootScope, $controller;

  beforeEach(inject(function($injector) {
    $httpBackend = $injector.get('$httpBackend');
    $rootScope = $injector.get('$rootScope');
    $controller = $injector.get('$controller');
    $httpBackend.when('GET', 'app/name').respond({ name: 'Klaus' });
  }));

  it('Should fetch name and set it to scope', function() {
    $httpBackend.expectGET('app/name');
    var scope = $rootScope.$new();
    var ctrl = $controller(MyController, { $scope: scope });
    $httpBackend.flush();
  expect(scope.name).toBe('Klaus');
  $httpBackend.verifyNoOutstandingExpectation();
    $httpBackend.verifyNoOutstandingRequest();
  });
});
```

wie vor wachsenden Community. Die Gesamtarchitektur der Webapplikation ist in **Abbildung 1** zu sehen.

Testing

Java mit einem REST-Interface auf dem Server, JavaScript, HTML und CSS für den Client: Dieser Ansatz erlaubt es uns, die clientseitige Funktionalität entkoppelt vom Backend zu entwickeln. Es vereinfacht und beschleunigt nicht nur die Entwicklung von Features, sondern führt auch zu einer architektonischen Verbesserung. Während Geschäftslogik und Daten im Backend gekapselt sind und ihr Zustand auch dort verwaltet wird, kümmern wir uns im Frontend um den Anwendungszustand („application state").

Bisher ließ sich nur die Backend-Funktionalität entkoppelt vom Gesamtsystem testen, während das GUI nur mit aufwändigen End-to-End-Tests – etwa mit Frameworks wie Selenium – testbar war. Dank der sauberen Trennung von Logik und Präsentation wird nun auch die Clientlogik automatisiert und isoliert testbar. Listing 3 und 4 zeigen das anhand von AngularJS und Jasmine (einer populären JS-Testbibliothek). Dabei werden die Backend-Services (*$http*, *$resource*) durch Mocks [8] ersetzt, was das serverunabhängige Testen der Controller ermöglicht.

Für die Ausführung von JavaScript-Tests im Rahmen des Build-Prozesses existieren Plug-ins. Für Maven beispielsweise das *jasmine-maven-plugin*.

Ausblick

Während JSF für einfache formularbasierte Anwendungen durchaus eine gute Lösung sein kann, bieten REST und HTML5 gerade bei Single-Page-Webanwendungen ein vielversprechendes neues Architekturkonzept. REST erfreut sich in letzter Zeit sowohl in der Webentwicklung als auch in Architektenkreisen zunehmender Popularität. Dieser Ansatz lässt sich hervorragend in Java umsetzen und mit JavaScript-Bibliotheken zur clientseitigen Entwicklung kombinieren. Dabei sollten wir als professionelle Softwareentwickler die architektonischen Grundkonzepte von REST beherzigen, JavaScript als Sprache ernst nehmen und Vorteile, die dieser Architekturansatz mit sich bringt, wie die hervorragende Testbarkeit, voll ausschöpfen. Für diejenigen, die jetzt neugierig geworden sind, wie das konkret aussehen kann, beleuchten wir die hier beschriebenen Konzepte in einer der kommenden Ausgaben mit einer Beispielanwendung und viel Code.

Andreas Drobisch ist Entwickler bei der camunda services GmbH und beschäftigt sich momentan mit dem aktuellen „Server to Client" Shift bei Webanwendungen und den verschiedenen Deployment-Szenarien.

Nico Rehwaldt ist Entwickler bei der camunda services GmbH. Hin- und hergerissen zwischen Java und modernen clientseitigen Webtechnologien interessiert er sich vor allem dafür, wie sich das Beste aus beiden Welten kombinieren lässt.

Nils Preusker ist Berater und Trainer bei der camunda services GmbH. Neben dem Einsatz von Open-Source-Technologien für BPM- und SOA-Projekte interessiert er sich für die Zukunft der Mensch-Maschine-Interaktion.

Links & Literatur

[1] http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.html

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

[3] http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.1

[4] Stefan Tilkov: REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien, 2. Auflage, dpunkt.verlag, 2011.

[5] Thomas Erl, Benjamin Carlyle, Cesare Pautasso, Raj Balasubramanian: SOA with REST: Principles, Patterns & Contraints for Building Enterprise Solutions with REST, Prentice Hall, 2012.

[6] http://jquery.com

[7] http://addyosmani.com/resources/essentialjsdesignpatterns/book/#detailmvcmvp

[8] http://martinfowler.com/articles/mocksArentStubs.html

Anzeige

Anzeige