Legacy Web-Apps mit AngularJS pimpen

WEIGLEWILCZEK

thecodecampus</>

Über uns

- Jan Blankenhorn und Philipp Burgmer
- Software Developers
- w11k.com / thecodecampus.de —> Esslingen / Stuttgart
- Schulungen, Projekt-Kickoff
- Consulting, Softwareentwicklung

Was ist dein Problem?

Probleme

- Pflege und Weiterentwicklung alter Webanwendungen
- Kunden verwöhnt von modernen Anwendungen
  —> wollen ähnliche Features
- Entwickler genervt von alten Technologien
- Zunehmend schwieriger Entwickler für alte Technologien zu finden

Beispiele Anforderungen

- Autovervollständigung bei Suche
- Schnelle Rückmeldung auf Eingaben / Validierung
- Schnellere Reaktionszeiten der Anwendung
- Website als Anwendung nicht als Website

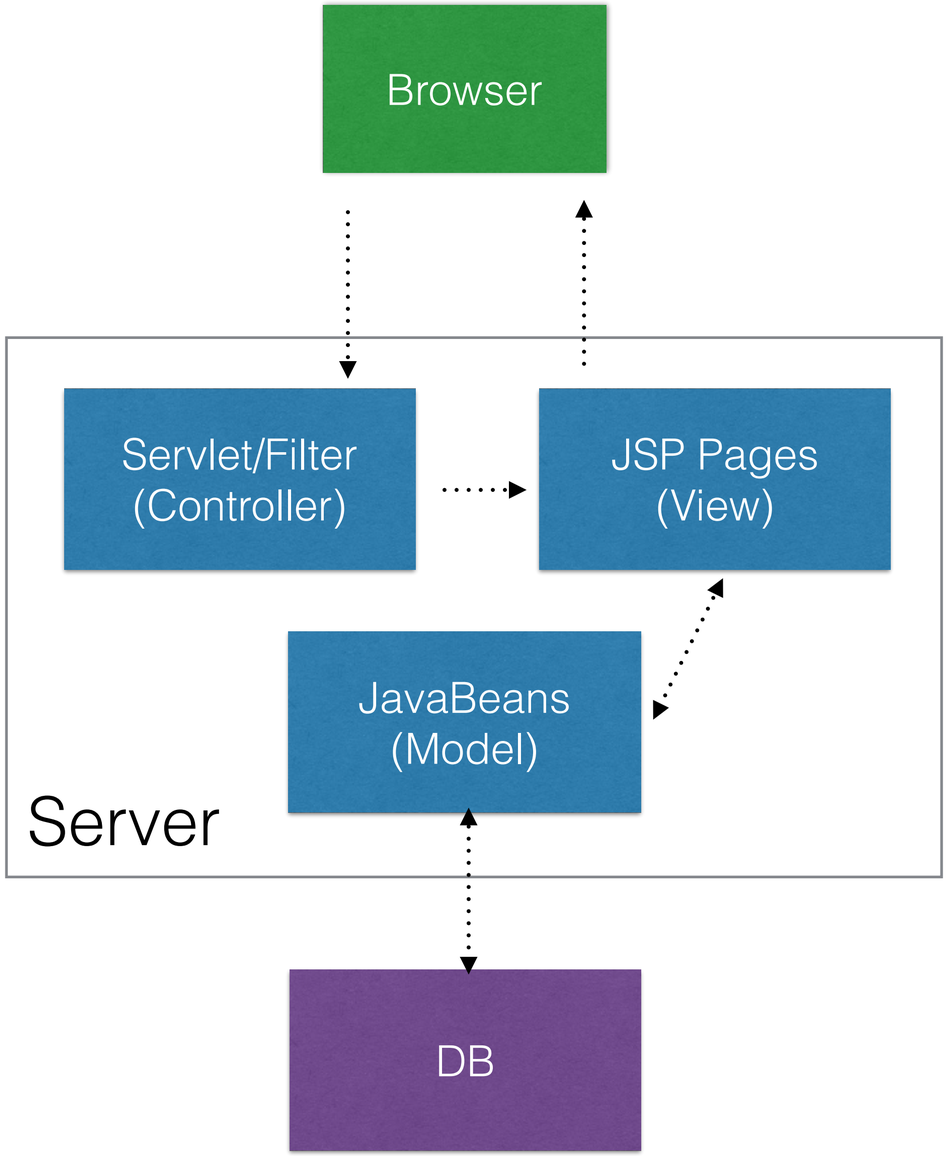
Lösungsansatz

- Neu Implementieren: Oft zu teuer und zu gefährlich
- Bleibt nur: alte und neue Technologien verbinden

Architekturvergleich

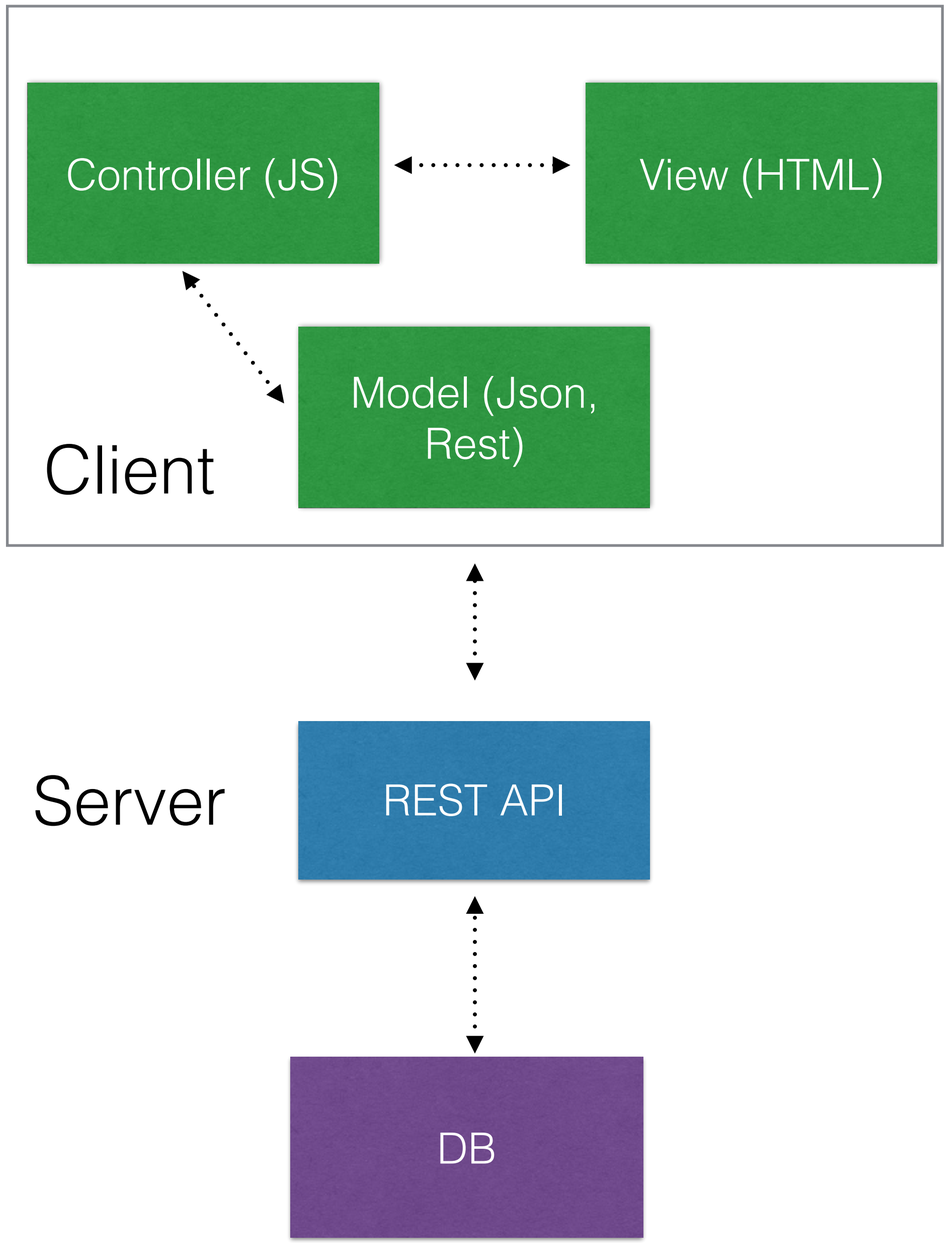
Klassische Java Webanwendungen (JSP / JSF)

- Rendering Template -> HTML geschieht auf dem Server
- Zustand jedes Benutzers liegt auf dem Server
  —> Skaliert schlecht
- Komplette Page Requests pro Interaktion
- JavaScript nur für kleine Aufgaben


Browser
Servlet/Filter
(Controller)
JSP Pages
(View)
JavaBeans
(Model)
Server
DB

JavaScript Apps

- Client hat den Zustand und UI-Logik
- Server hat keinen Zustand —> Bessere Skalierung
- Server liefert
  - statische Ressourcen wie Templates und JavaScript Code
  - Daten via REST/JSON
- Weniger Redundanz bei Übertragung


Controller (JS)
View (HTML)
Model (Json, Rest)
Client
Server
REST API
DB

Szenarien

Nur Client

Client + Server

Rest Backend

Nur Client

Szenario 1a

Ausgangssituation

- Klassische Web-Anwendung z.B. mit Struts
- Keine Änderungen an Server Architektur möglich
- Server liefert weiterhin fertiges HTML

Lösungsansatz

- Client per JavaScript erweitern
- Direkte Interaktion bieten
- HTML bzw. DOM nutzen
- Höheres Level als jQuery

AngularJS

- JavaScript-Framework zur Entwicklung von Rich Browser Applikationen
- Bringt grundlegende UI Konzepte wie z.B. MVC in den Browser
- Erweitert HTML anstatt zu abstrahieren
- HTML kann nach den gegebenen Bedürfnissen erweitert werden

AngularJS

- Eigentlich für Single-Page-Anwendungen gedacht
- Leichtgewichtig, schnelle Initialisierung
- Kann ruhig bei jedem Page-Reload geladen werden
- JavaScript Code in Dateien auslagern -> Caching
- Auch auf Teil des DOM anwendbar

Beispiel

- Formular Validierung mit AngularJS
- Server generiert HTML mit speziellen Attributen
- AngularJS verwendet Attribute zum Validieren
- Client zeigt Fehlermeldungen sofort an
  (mitgeliefert vom Server im HTML)

```
<div ng-app>
  <form name="userForm" novalidate post="createUser.do">
    <label for="userForm.email">E-Mail:</label>
    <input type="email" id="userForm.email" ng-model="user.email" name="email" required>
    <div ng-messages="userForm.email.$error">
      <div ng-message="required">Please enter your email address</div>
      <div ng-message="email">Please enter a valid email address</div>
    </div>

    <button type="submit" ng-disabled="userForm.$invalid"></button>
  </form>
</div>
```

Validatoren

- Standard HTML Attribute
  - min, max, required
  - type mit email, date, time, number, url
- AngularJS Attribute
  - ng-min-length und ng-max-length
  - ng-pattern
- Eigene Validatoren per Attribut und JavaScript Code

Ajax mit DWR

Szenario 1b

Ausgangssituation

- Klassische Web-Anwendung z.B. mit Struts
- Keine Änderungen an grundlegender Server Architektur möglich

Lösungsansatz

- Server liefert weiterhin fertiges HTML
- Beliebige JavaScript Frameworks oder VanillaJS im Client
- Ajax Kommunikation mittels DWR

DWR

- Servlet, dass Ajax-Requests verarbeitet
- DWR erstellt JavaScript Stubs für Java Klassen und übernimmt Client - Server Kommunikation
  - Eine Art “Remote Procedure Call”
- Spring / Guice / Struts Integration
- http://directwebremoting.org/dwr/index.html

Beispiel

- Dynamisches Anzeigen einer Liste
- Klassisch: jeweils ein voller Page Request nötig
- DWR Lösung:
  1. AJAX Request zum Laden der Daten
  2. JavaScript: Anzeige der Daten

Konfigurieren

```
<dwr>
<!-- Nur Klassen in <allow> werden konvertiert -->
<allow>
    <!-- Definieren der Klasse die freigegeben werden soll
         Erstellt wird die Klasse von Struts
  -->
  <create creator="struts" javascript="AjaxService">
      <!-- auflisten der Methoden -->
    <include method="getAllElements"/>
  </create>
</allow>
</dwr>
```

Auch per Annotations konfigurierbar

Einbinden

```
<script .. src="/lib/static/dwr/2.0/engine_and_util.min.js"/>
<script .. src="/dwr/interface/AjaxService.js"/>
```

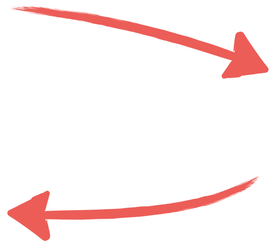
Benutzen

Webbrowser

Server

```
/**
* Java Script Code
**/
AjaxService.getAllElements(populateList);

function populateList(data) {
  //etwas mit den Daten machen
  console.log(data);
}
```



```
public List<String> getAllElements() {
  return elements;
}
```

Client + Server

Szenario 2b

Ausgangssituation

- Klassische Web-Anwendung z.B. mit Struts
- KEINE saubere Trennung zwischen Business-Logik und Web-Schnittstelle
- Änderungen am Server in begrenztem Umfang möglich
- Anwendung soll schrittweise erneuert werden

Lösungsansatz

- Server liefert nur noch Daten und Temples (getrennt)
- State kann im Server bleiben
- AngularJS im Client setzt Daten und Templates zusammen

Probleme

- Server ist eigentlich gedacht fertig gerenderte HTML-Seiten auszuliefern -> Hack ;)
- Java Daten müssen serialisiert werden

Template & Daten

- Laden der Seite in 2 Requests aufteilen
- Normaler“ Struts Request liefert eine JSP Seite mit dem HTML Template und Code für AngularJS Anwendung
- AngularJS Anwendung lädt dann die Daten als JSON über zusätzliche Requests vom Server

```java
/*
 * Struts Action
 */
public ActionForward doExecute(...) throws Exception {
    final String acceptType = request.getHeader("Accept");

    // Abfragen des Accept Types
    // 1. Call
    if (false == acceptType.startsWith("application/json")) {
        // JSP Seite zurückgeben.
        // Enthält die JavaScript Anwendung
        return mapping.findForward("template");
    }
    // 2. Call
    else {
        // Daten erstellen und serialisieren
        final Object data = buildData(request, response, form);
        final String json = serializeData(data);
        request.setAttribute("jsonResponse", json);

        // antworten mit JSON
        return mapping.findForward("jsonResponse");
    }
}
```

```java
/*
 * Beispiel: Daten mittels GSON zu JSON sterilisieren
 */
private String serializeData(final Object data) {
  final GsonBuilder builder = new GsonBuilder();
  builder.serializeNulls();
  final Gson gson = builder.create();

  final String json = gson.toJson(data);

  return json;
}
```

https://code.google.com/p/google-gson/

```
<!-- JsonResponse.jsp
  Minimale JSP Seite, in die das Json Eingebunden wird
   -->
<%@ page contentType="application/json; charset=UTF-8" pageEncoding="UTF-8"%>
${jsonResponse}
```

```
<!-- JSP Seite mit AngularJS Anwendung -->
<script type="text/javascript">
angular.module("List", []);

angular.module('List').controller("ListCtrl", function ($scope) {
  $scope.data = [];

  $scope.search = function() {
    var requestConfig = {
      searchText: $scope.searchText
    };

    // Aufruf der Server REST Schnittstelle
    $http.get('http://localhost/showData.do', requestConfig).then(function (response) {
      // Verarbeitung der Daten
      $scope.data = response.data;
    });
  };

  $scope.search();
});
</script>
```

```
<div ng-app="List" ng-controller="ListCtrl">
  <div class="header">
    <input type="text" ng-model="searchText" placeholder="Full Text Search">
    <button ng-click="search()">Search</button>
  </div>

  <table>
    <thead>
      <tr>
        <th ng-click="sortDataBy('name')">Name</th>
        <th ng-click="sortDataBy('description')">Description</th>
      </tr>
    </thead>
    <tbody>
      <tr ng-repeat="entry in data">
        <td><a ng-href="{{entry.link}}">{{entry.name}}</a></td>
        <td>{{entry.description}}</td>
      </tr>
    </tbody>
  </table>
</div>
```

Anwendungsfälle

- Listen-Ansicht mit Sortierung, Filtern und Volltextsuche
- Detail Ansicht mit dynamischem Nachladen von Daten (z.B. in Tabs oder Popups)
- Mini-Single-Page-App auf einer Unterseite (CRUD)

Client + sauberer Server

Szenario 2b

Ausgangssituation

- Klassische Web-Anwendung mit z.B. Struts
- Saubere Trennung zwischen Business-Logik und Web-Schnittstelle
- Änderungen am Server in begrenztem Umfang möglich

Lösungsansätze

- Ersetzen der Web-Schnittstelle durch REST API
- Ausliefern der Temples als statisches HTML
- Client wird wieder mit AngularJS umgesetzt

REST (Jersey)

```java
/**
 * wird unter dem Pfad "resource" bereitgestellt
 */
@Path("resource")
public class Resource {

    /**
     *
     * Methode verarbeitet HTTP GET requests.
     * Das Resultat wird als "text/plain" gesendet
     */
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getIt() {
        return "Got it!";
    }
}
```

https://jersey.java.net/

REST Backend

Szenario 3

Ausgangssituation

- Etwas modernere Web-Anwendung
- REST Backend
- Flex Client

Lösungsansatz

- Client schrittweise portieren
- Flex Anwendung anpassen und in neues HTML Grundgerüst integrieren
- Flex nur bei Bedarf anzeigen
- Kommunikation Flex <-> JavaScript

Probleme

- Flash —> display: none —> display: block —> neue Initialisierung
- JS Code ruft ActionScript zu früh auf
- ActionScript ruft JS Code auf —> global, kein Angular Kontext

Lösung: w11k-flash

- Open Source
- Github: http://github.com/w11k/w11k-flash
- Blog Artikel mit ausführlicher Erklärung: http://blog.thecodecampus.de/migration-von-flex-zu-angularjs

```
<div ng-controller="TestCtrl">
  <div w11k-flash="flash.config"
       w11k-select-visible="flash.visible"
       >
  </div>
</div>
```

```
angular.module('app').controller('TestCtrl', function ($scope) {
  $scope.flash = {
    config: {
      swfUrl: 'assets/test.swf',
      callback: function (readyPromise) {
        $scope.flash.ready = readyPromise;
      }
    }
  };

  $scope.talkToMe = function (message) {
    $scope.message = message;
    $scope.response = 'Hello! My name is AngularJS.';
    return $scope.response;
  };

  $scope.talkToFlex = function () {
    if (angular.isDefined($scope.flash.ready)) {
      $scope.flash.ready.then(function (flash) {
        $scope.message = 'Hello! My name is AngularJS. What is your name?';
        $scope.response = flash.talkToMe($scope.message);
      });
    }
  };
});
```

```
protected function application_creationCompleteHandler(event:FlexEvent) :void {
  this.angularjs = AngularJSAdapter.getInstance();

  // initialize flex application
  this.currentState = defaultState.name;
  ExternalInterface.addCallback("talkToMe", talkToMe);

  this.angularjs.fireFlashReady();
}

protected function talkToMe(message :String) :String {
  this.message = message;
  response = 'Hello! My name is Flex.';
  return response;
}

protected function talkToAngularJS() :void {
  message = 'Hello! My name is Flex. What is your name?';
  response = this.angularjs.call('talkToMe(message)', { message: message });
}
```

Fazit

Herausforderungen

- Transaktionen, da asynchrone Kommunikation
- Sicherheit: Client vs. Server
- Validierung: Client + Server
- Integration in bestehendes Build System
- Sehr viele Technologien = Polyglott

Aber

- Macht Spaß
- Meist schnell und recht einfach möglich
- Produktivitätsschub für neue Features
- Generiert beim Kunden Lust auf mehr

Legacy Web-Apps mit AngularJS pimpen

WEIGLEWILCZEK

thecodecampus</>

Jan Blankenhorn
blankenhorn@w11k.com

Philipp Burgmer
burgmer@w11k.com

github.com/w11k
blog.thecodecampus.de

WEIGLEWILCZEK

thecodecampus</>