KaffeeKlatsch
Das Magazin rund um Software-Entwicklung

ISSN 1865-682X
05/2011

Geh weg, bleib da!

Anbindung mobiler Clients mit jQuery und REST

von Werner Eberling

Geh weg, bleib da!

Anbindung mobiler Clients mit jQuery und REST

von Werner Eberling

Mobile Clients gewinnen in modernen Anwendungsszenarien mehr und mehr an Bedeutung. Dieser Artikel zeigt warum „klassische“ Web-Anwendungen für diese neue, heterogene Client-Landschaft nicht mehr ausreichen und wie mit Hilfe von jQuery und REST eine Anbindung an bestehende Server-Systeme erreicht werden kann, ohne sich tiefer mit den Spezifika der einzelnen Endgeräte auseinander setzen zu müssen.

Die Anbindung verschiedenartiger Clients ist zugegebenermaßen keine neue Herausforderung. Um unsere Software nicht für verschiedene Systeme anpassen zu müssen, haben wir hierfür schon seit langer Zeit eine Standard-Lösung: die Web-Anwendung. Waren es zur Anfangszeit Servlets und JSP-Seiten, haben wir uns inzwischen über Apache Struts zu Frameworks wie Java ServerFaces und Apache Wicket „hochgearbeitet“.[1]

Doch die heterogenen Clients von heute tragen nicht mehr die Bezeichnung Linux, Windows oder OS/2. Um den neuen Herausforderungen gerecht zu werden, müssen iPhone, iPad, Android-Phone bzw. -Tablet und natürlich Windows Mobile mit im Boot sein. Dabei sind es nicht die Namen, die die Anforderungen verändern, sondern vielmehr die Mobilität der neuen Endgeräte. War beim „klassischen“ Web-Client von einer ständigen Netzverbindung auszugehen, so ist dies bei den neuen Endgeräten grundsätzlich anders. So gut wie die Abdeckung mit WLAN oder UMTS (neu-deutsch: „3G“) auch sein mag, mit Funklöchern ist zu rechnen und der daraus resultierende Abriss der Server-Verbindung soll die Funktionsfähigkeit der Anwendung nicht oder zumindest nur so wenig wie möglich beeinflussen.

Offline-Fähigkeit

Der Betrieb der Anwendung ohne eine bestehende Verbindung zum Server, der sogenannte Offline-Betrieb, ist eine Anforderung deren Häufigkeit in den Anforderungskatalogen aktueller Client-Server-Projekte signifikant steigt. Die erste Antwort hierauf erscheint einfach: die Entwicklung eines Rich-Clients.

Soll eine Verarbeitung ohne Server möglich sein, muss die Logik nun einmal zurück auf den Client. Doch mit diesem Schritt schlägt die Plattformvielfalt in ihrer vollen Pracht zurück. Das iOS-basierte System hätte seinen Client gerne in Objective-C, die Android-Variante in Java und Windows Mobile bitteschön .Net-basiert. Soll nun eine Applikation für verschiedene Endgeräte verfügbar sein, bedeutet dies die Unterstützung der jeweils benötigten Plattformen. Um die so entstehende technische Vielfalt und die damit verbundene Komplexität zu reduzieren, wird ein „echter“ Web-basierter Rich-Client benötigt.[2]

Single-Page-Applikationen

Voraussetzung für die Bereitstellung eines Web-basierten Rich-Clients ist die Möglichkeit die Anwendung, während Netzwerkzugriff besteht, komplett auf das Endgerät zu überspielen. Dies beinhaltet die für die Darstellung notwendigen Informationen (den *View*) und die vom Client benötigte Verarbeitungslogik. Lediglich die darzustellenden bzw. zu verarbeitenden Daten werden im weiteren Verlauf noch zwischen Client und Server ausgetauscht. Um dies auf Basis von HTML zu erreichen, muss die gesamte Anwendung in Form einer einzigen

[1] Man möge mir an dieser Stelle den Fokus auf die Java-Welt verzeihen, aber einerseits ist diesem Artikel der Java-Einschlag nicht abzusprechen und andererseits bietet wohl kaum eine Technologie einen so bunten Strauß an unterschiedlichen Web-Frameworks wie die Java Enterprise Edition (aber das ist ein Thema für einen eigenen Artikel).

[2] Der Begriff der „Rich Internet Application“ (RIA) ist schon seit Jahren allgegenwärtig, aber sobald der Netzwerk-Stecker gezogen wird, ist es mit der „Richness“ der klassischen RIAs meist schnell vorbei.

Web-Seite zur Verfügung gestellt werden. Dieser Ansatz wird als Single-Page-Applikation oder auch als Single-Page-Interface bezeichnet.

Die Adressbuch-App

Als Beispiel-Anwendung soll im Rahmen dieses Artikels eine Adressbuch-Applikation dienen. Die Darstellung der kompletten Quellen würde den Rahmen dieses Artikels sprengen, weswegen darauf verzichtet wird.[3] Vielmehr wird hier nur auf grundlegende Ideen beim Aufbau der HTML-Quellen, Möglichkeiten und wichtige Regeln beim Einsatz von jQuery sowie die Schnittstelle des Backends und deren Aufruf via jQuery eingegangen.

Das Frontend

Um die gesamte Anwendung in einer einzigen HTML-Seite auszuliefern, müssen sämtlichen Client-Screens in einer HTML-Struktur untergebracht werden. Dies geschieht in Form einer DIV-Struktur, in der jeder Client-Screen von einem einzelnen DIV-Element definiert wird. Innerhalb dieses Elements erfolgt dann die Definition des Screens unter der Zuhilfenahme sämtlicher Möglichkeiten des HTML-Standards.

```
<HTML>
  …
  <BODY>
    <DIV id=“main“ class=“screen“>
    ...
    </DIV>
    <DIV id=“about“ class=“screen“>
   ...
    </DIV>
  </BODY>
</HTML>
```

Während der Interaktion des Users mit der HTML-Seite (sozusagen während der „Laufzeit“ der Anwendung) werden die benötigten Screens durch die Veränderung von CSS-Styles ein bzw. ausgeblendet. Leider bringt bereits ein so einfaches Detail wie das ein- und ausblenden von HTML-Elementen das Problem der Browser-Inkompatibilität mit sich. Reagieren die meisten Vertreter dieser Zunft auf den CSS-Style *display: none*, verlangt u.a. Opera ein *visibility: hidden* damit das Ausblenden von DIV-Elementen von Erfolg gekrönt ist.

Um sich nun nicht mit diesen oder ähnlichen Browser-Problemen herumschlagen zu müssen, kommt jQuery [1] zum Einsatz. Diese JavaScript-Bibliothek kapselt die technischen Details von Veränderungen in der Client-seitigen Repräsentation der HTML-Seite, wie beispielsweise das Ein- oder Ausblenden von Elementen in entsprechende Funktionsaufrufe und kümmert sich innerhalb dieser Funktionen um die Spezifika des jeweiligen Browsers.

```
    //Ausblenden der main-Screen
    $('#main').hide();

    //Einblenden der about-Screen
    $('#about').show();
});
```

Für diejenigen Leser, die es gerne etwas effektvoller mögen, bietet jQuery auch Möglichkeiten zum ein- und aussliden von Bildschirminhalten. Hierfür stehen u.a. die Methoden *slide()* und *animate()* zur Verfügung. Die Adressbuch-App nutzt dieses Feature zum Umschalten zwischen verschiedenen Screens:

```
function slideAllLeft(){
  $('.slide').animate({"left": "-=100%"}, "slow");
}

function slideAllRight(){
  $('.slide').animate({"left": "+=100%"}, "slow");
}
```

Struktur oder Chaos

Ein Client auf Basis von HTML und JavaScript – bleibt da nicht die Lesbarkeit auf der Strecke? Nicht unbedingt, wie bei jeder Technologie ist auch hier die Struktur im Aufbau der Anwendung und die Disziplin auf Seite des Entwicklers das Mittel zum Erfolg. Das Zauberwort heißt in diesem speziellen Fall *Unobtrusive JavaScript*[4] [2]. Die grundlegende Idee dieser Herangehensweise, die auf Ideen eines Artikels von Stuart Langridge zurück geht [3], ist die strikte Trennung von Inhalt (HTML), Präsentation (CSS) und Verhalten (JavaScript).

Im konkreten Anwendungsfall bedeutet dies kein JavaScript und keine Style-Information in das HTML-Markup zu integrieren. Vielmehr werden die entsprechenden JavaScript-Dateien und Stylesheets, die Verhalten bzw. Darstellungsinformationen definieren, in der HEAD-Sektion des HTML-Dokuments nur noch referenziert.

[3] Die vollständige Applikation steht im Internet zum Download zur Verfügung. Genauere Informationen hierzu sind am Ende des Artikels zu finden.

[4] Unobtrusive {adj.} – engl.: unauffällig, unaufdringlich, zurückhaltend.

```
<HTML>
  <HEAD>
    <LINK
      rel="stylesheet" href="css/addressbuch.css" type="text/css" />
    <SCRIPT
      type="text/javascript" src="js/jquery-1.5.2.js" />
    <SCRIPT
      type="text/javascript" src="js/json2.js" />
    <SCRIPT
      type="text/javascript" src="js/addressbuch.js" />
  </HEAD>
...
</HTML>
```

Die so erhaltene klare Aufteilung der einzelnen Anwendungsbestandteile ermöglicht es auch bei grösseren Applikationen den Überblick zu behalten.

Verhalten an HTML-Elemente binden

Mit CSS-Klassen die Präsentation einer HTML-Struktur zu definieren ist nichts wirklich spannend Neues, weswegen hier auf weitere Details zu diesem Thema verzichtet wird. Doch wie sieht es mit dem Binden von Aktionen an ausgewählte HTML-Elemente aus? Hier kommt wieder jQuery zum Einsatz.

Das Vorgehen ist dabei vergleichbar mit der Definition und Registrierung von *ActionListenern* im Rahmen einer JavaSwing-Anwendung. JQuery bietet die Möglichkeit, Funktionen als Callbacks für JavaScript-Events von HTML-Elementen zu hinterlegen. Anders ausgedrückt: Sobald z. B. ein HTML-Link oder ein Button gedrückt wird, kann eine vom Entwickler definierte Funktion ausgeführt werden.

Da die Bindung der Aktionen während der Initialisierung der Webseite erfolgen muss, sollte dies geschehen, wenn der Browser die HTML-Seite vollständig geparsed und den der Seite entsprechenden DOM-Baum erstellt hat. JQuery bietet hierfür den *document.ready*-Event an, für den, wie bei den Callback-Events der HTML-Elemente, eine Funktion hinterlegt werden kann, die vom Browser nach der Erstellung des DOM-Baums ausgeführt wird.

```
//Durchführung von Aktionen nach dem
//Aufbau des DOM-Baums im Browser
$(document).ready(function(){

  $('#reloadBtn').click(load);

  $('#saveBtn').click(
    function(){
      addAdresse();
      clearNewAdresse();
    });
  ...
});
```

Das Backend

Nach der Betrachtung der grundlegenden Ideen zum Aufbau des Clients stellt sich nun noch die Frage, wie die Server-Seite des Adressbuchs aussieht. Um der jQuery-Anwendung eine leichtgewichtige, sprachneutrale Schnittstelle anzubieten sind hierfür RESTful Web-Services eine gute Wahl. Diese werden im JEE-Umfeld durch den JAX-RS-Standard abgebildet.

Die grundlegende Idee des RESTful-Ansatzes ist die Bereitstellung bzw. Manipulation von Ressourcen (in diesem Fall: Adressen) auf Basis eindeutiger Identifikatoren (URLs) und einem Standardsatz von Methoden (*GET, PUT, POST, DELETE*) auf diesen Ressourcen. Im Rahmen dieses Artikel soll diese, recht kurzgefasste Beschreibung des REST-Ansatzes genügen. Für eine tiefer gehende Betrachtung sei hier auf [4] bzw. [5] verwiesen.

Das Modell

Wie sehen nun die Ressourcen im Falle des Adressbuchs aus? Einfach gesagt, wie man es in Java erwarten würde. Eine Adresse wird als JavaBean repräsentiert, die einen Default-Konstruktor und Getter und Setter für alle für den Client sichtbaren Attribute besitzt. Nachdem Adressen auch in die Datenbank gespeichert werden sollen, wird die Klasse noch zusätzlich als JPA-Entity annotiert und mit einer technischen Id versehen.

```
@javax.persistence.Entity
@javax.xml.bind.annotation.XmlRootElement
public class Adresse {
  @javax.persistence.Id
  private int id;
  private String name;
  // weitere Attribute

  public Adresse() {
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
  // weitere Getter und Setter
}
```

Die Adresse als Server-seitiges, persistentes Objekt

Als weitere Besonderheit fällt die Deklaration der Klasse als *XMLRootElement.* auf. Diese Annotation wird benötigt, um die Serialisierung bzw. Deserialisierung durch die JAX-RS-Implementierung zu ermöglichen. Denn was auf Server-Seite noch als Java-Objekt zur Verfügung steht, soll ja technologie-neutral auf den Client transportiert werden. Eine mögliche Variante ist hierbei die Umsetzung in XML (auf Basis von JAX-B) oder in das deutlich leichtgewichtigere JSON[5]. Beide Ansätze serialisieren Objektgraphen in Textdarstellung bzw. aus dieser wieder zurück in Java-Objekte. Um diese Umwandlung korrekt vorzunehmen, muss die Wurzel des Datenmodells gekennzeichnet werden. Genau dies wird mit der Annotation *@XMLRootElement* erreicht.

Die Schnittstelle

Um die Speicherung von Adressen transaktionssicher zu gestalten, wird das Adressbuch als Enterprise JavaBean, genauer gesagt als *Stateless SessionBean*, realisiert. Diese bietet Methoden zum Anlegen neuer Einträge, zur Abfrage bestehender Einträge sowie zur Löschung einzelner Einträge an. Das nachfolgende Code-Stück zeigt das lokale Business-Interface der Adressbuch-EJB. Auf konkrete Implementierungsdetails wird an dieser Stelle verzichtet, da das Augenmerk auf der Implementierung einer REST-Ressource mittels JAX-RS gelegt werden soll.

```
@javax.ejb.Local
@javax.ws.rs.Path("adressen")
public interface Adressbuch {

  @javax.ws.rs.GET
  @javax.ws.rs.Produces("application/json")
  public List<Adresse> getAll();

  @javax.ws.rs.GET
  @javax.ws.rs.Path("{id}")
  @javax.ws.rs.Produces("application/json")
  public Adresse getAdresse(
    @javax.ws.rs.PathParam("id") int id);

  @javax.ws.rs.POST
  @javax.ws.rs.Consumes("application/json")
  @javax.ws.rs.Produces("application/json")
  public Adresse addAdresse(Adresse adresse);

  @javax.ws.rs.DELETE
  @javax.ws.rs.Path("{id}")
  public void removeAdresse(
    @javax.ws.rs.PathParam("id") int id);
}
```

Schnittstelle des Adressbuchs mit JAX-RS-Annotationen

[5] JavaScript Object Notation.

Alternativ wäre auch die Implementierung als einfache Java-Klasse (neu-deutsch: POJO) möglich, in diesem Fall wären die JAX-RS-Annotationen direkt in der Klasse platziert worden.

Client-Zugriff auf die Web-Services

Wie erfolgt nun der Zugriff aus der Web-Seite auf die bereitstehende Schnittstelle? Für jQuery existieren eine Reihe von Plugins, die den Zugriff auf RESTful Web-Services erleichtern, wie z. B. das *JSON-Rest* Plugin [6]. Der Einsatz solcher Plugins bringt aber unter Umständen Einschränkungen bezüglich der Auswahl der eingesetzten jQuery-Version mit sich. So ist das genannte Plugin unter der aktuellen jQuery-Version 1.5 nicht lauffähig. Hier heißt es entweder auf eine ältere Version von jQuery zurückgehen oder auf den Komfort des Plugins verzichten. Da das Absetzen eines AJAX-Calls mit jQuery keine wirklich komplizierte Aufgabe darstellt und die in [6] enthaltene JSON-Library problemlos unter jQuery 1.5 funktioniert, ist der „selbstgestrickte" Web-Service-Aufruf hier das Mittel der Wahl.

```
$.ajax({
  type: "GET",
  url: "adressen",
  success: function(data){
    adressen = data;
    initAdressbuch();
  }
});
```

Web-Service-Aufruf via jQuery

Neben der Angabe der HTTP-Methode und der relativen URL wird beim AJAX-Call eine Callback-Methode angegeben, die nach erfolgreichem Web-Service-Aufruf aufgerufen wird. Diese Methode ist notwendig, da AJAX-Calls asynchron ablaufen, der aufrufende Thread also nicht blockiert und man somit auch das Ergebnis nicht direkt erhält.

Offline-Fähigkeit

Sollte der Web-Service nicht erreichbar sein, müssen Änderungen, die im Client vorgenommen werden zwischengespeichert und später zum Server geschickt werden. Hierzu wird eine weitere Callback-Methode des AJAX-Calls verwendet, die es ermöglicht, auf Fehler zu reagieren:

```
$.ajax({
   type: "POST",
   url: "adressen",
   contentType: "application/json",
   data: JSON.stringify(data),
   error: function(){
   updateCache.push(data);
   }
});
```

Einfache Fehlerbehandlung im Web-Service-Aufruf

Sollte die Anlage einer neuen Adresse fehlschlagen, werden die Adressdaten innerhalb eines Update-Caches auf Client-Seite vorgehalten. Der Inhalt dieses Caches wird in regelmäßigen Abständen mit dem Server abgeglichen und das Update somit zum nächstmöglichen Zeitpunkt auf den Server überspielt.

```
window.setInterval(replayUpdates, 10000);
```

Im Rahmen dieses Artikels soll die Implementierung des Caches als einfaches, transientes Array ausreichen. Soll der Update-Cache auch nach einem Neustart des Clients noch verfügbar sein, muss er auf Client-Seite persistiert werden. Hierzu bietet jQuery – über entsprechende Plugins – eine Reihe von Möglichkeiten wie z. B. die Speicherung in einem erweiterten Cookie (DOM-Storage) oder die Speicherung innerhalb einer Flash-Datei an. Bei Interesse sei hier auf das *jquery-jstore*-Plugin [7] verwiesen.

Paketierung und Deployment

Nachdem Client und Server-Seite nun vollständig sind, stellt sich zum Abschluss noch die Frage nach der Paketierung. Um die Adressbuch-Applikation als ein Artefakt deployen zu können, werden der statische Web-Client und das EJB-Backend gemeinsam in einem Web-Archive paketiert. Diese Deployment-Variante steht seit JEE 6 als sogenanntes *WAR-Deployment* zur Verfügung. Hierbei werden EJB-Klassen, JPA-Entitäten und die benötigten Deskriptoren im Verzeichnis *WEB-INF/classes* abgelegt und das Web-Archive wie gewohnt deployed.

Wird die JAX-RS-Implementierung *RESTEasy* von JBoss verwendet [8], sind keine weiteren Konfigurationen notwendig. In diesem Fall muss das Archiv nur noch in einen JBoss ApplikationServer ab Version 6.0.0 *final deployed* werden. Andere Implementierungen, wie beispielsweise die Referenzimplementierung Jersey [9] benötigen noch zusätzliche Konfigurationsinformationen im Web-Deskriptor (web.xml).

Fazit

Um mehrere Plattformen mit einer Implementierung zu bedienen, bleiben Web-Anwendungen das Mittel der Wahl. Die Herausforderung durch den Siegeszug der mobilen Endgeräte ist die Offline-Fähigkeit dieser Anwendungen. Hier bietet das Zusammenspiel von jQuery auf dem Client und REST auf der Server-Seite eine gute Basis zum Aufbau leichtgewichtiger mobiler Anwendungen. Speziell in einfachen Szenarien ist dieses Duo eine ernstzunehmende Konkurrenz gegen „native" Implementierungen. Doch auch für komplexere Applikationen werden JavaScript-Frameworks immer öfter ins Spiel gebracht. Hier wird es interessant sein, zu beobachten, ob es jQuery über kurz oder lang nicht in die Liga der Technologien für die Implementierung von Enterprise-Applikationen schafft, zumindest wenn es um die Implementierung von Web-Clients geht.

Die komplette Adressbuch-App

Die in diesem Artikel vorgestellte Adressbuch-Anwendung steht im Internet unter [10] zum Download zur Verfügung.

Referenzen

[1] jQuery *jQuery is a new kind of JavaScript Library.*
http://jquery.com

[2] Wikipedia *Unobtrusive JavaScript*
http://de.wikipedia.org/wiki/Unobtrusive_JavaScript

[3] Langridge, Stuart *Unobtrusive DHTML, and the power of unordered lists*
http://www.kryogenix.org/code/browser/aqlists

[4] Fielding, Roy T. *Architectural Styles and the Design of Network-based Software Architectures*
http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[5] Tilkov, Stefan *REST und HTTP,* dpunkt verlag 2009
http://rest-http.info

[6] jQuery *JSON-REST Plugin*
http://plugins.jquery.com/project/Rest

[7] jQuery *jStore Plugin*
http://plugins.jquery.com/project/jStore

[8] JBoss *RESTEasy*
http://www.jboss.org/resteasy

[9] Glassfish *Jersey*
http://jersey.java.net

[10] Eberling, Werner *Adressbuch-Anwendung*
http://www.werner-eberling.de/download/AddressbuchJQueryREST.zip

Kurzbiographie

Werner Eberling (werner.eberling@mathema.de) ist Principal Consultant und Technical Lead bei der MATHEMA Software GmbH in Erlangen. Seit 1999 beschäftigt er sich mit verteilten Systemen, sein Fokus liegt dabei auf CORBA und der JEE. Neben seiner Projekttätigkeit hält er Technologie-Trainings in Deutschland und der Schweiz und ist als Sprecher auf nationalen und internationalen Konferenzen anzutreffen. Er ist u. a. Autor des Buches *Enterprise JavaBeans 3.1 Das EJB-Praxisbuch für Ein- und Umsteiger*, erschienen im Hanser Verlag.