Client-Server-Kommunikation mit Ajax

Kommunikation in den Zeiten von Ajax

VON JOHANNES LINK

Ajax-Techniken können auf unzählige Art und Weise dazu verwendet werden, herkömmliche, seitenflussorientierte Webanwendungen aufzupolieren. Wesentlich interessanter ist jedoch, dass immer mehr Teile des Web 2.0 zu so genannten SPAs (Single Page Applications) werden, d.h. zu Applikationen, die lediglich das Internet und dessen Protokolle benutzen, um am Ende etwas zu sein, das mehr unserer Vorstellung von Rich Clients als der von hyper-verlinkten Webseiten entspricht.

Diese SPA-Sicht der Dinge rückt die Verbindung zwischen Anwendungs- und Präsentationslogik ins Rampenlicht. Denn eine der schwierigsten Entscheidungen, die wir bei der Verteilung von Programmcode zwischen Client und Server treffen müssen, ist: Welchen Code wollen wir nur auf dem Server haben, und welchen Code entwickeln wir für – und deployen ihn auf – den Client? An den beiden äußeren Enden des denkbaren Lösungsspektrums stehen die SOA-Hardliner und die Lobbyisten der Composable Widgets. Die erste Gruppe setzt sich dafür ein, möglichst die ganze Integrationslogik auf dem Client zu haben und lediglich applikationsunabhängige Serviceaufrufe an den Server zu delegieren. Die zweite Gruppe hingegen erledigt sogar das HTML Rendering der Nutzeroberfläche auf dem Server – gewöhnlich begleitet von hinter den Kulissen generiertem JavaScript-Code – und verknüpft die Client Widgets mit ihren Gegenstücken auf dem Server. Typische Vertreter dieser Idee sind Ajax-befähigte JSF-Implementierungen. In der Praxis bringen beide Extreme einige Probleme mit sich:

- Die Erstellung von „Webbzwonull-Benutzererfahrungen“ mit HTML, CSS, JavaScript und DOM verträgt sich nicht nahtlos mit dem Ansatz, eine Menge grafischer Standardkomponenten – Widgets – zu einer Applikation zusammenzuklicken. Hinzu kommt, dass die enge Kopplung zwischen client- und serverseitigen Komponenten zu einer hochfrequenten und feingranularen Kommunikation führt, die zudem überwiegend synchron ist. Dies bringt uns neue Schwierigkeiten im Netzlatenz-Bereich, der serverseitigen Sessionverwaltung und dem eigtl. asynchronen XMLHttpRequest-Objekt.
- Wenn man jedoch „nur“ Dienste auf dem Server aufruft und deren Ergebnisse auf dem Client zusammenbaut, dann muss man große Teile des Domänenmodells und der Domänenlogik auf dem Client zur Verfügung stellen. Mit anderen Worten: Ein nichttriviales Mapping zwischen serverseitiger Sprache und JavaScript wird notwendig. Darüber hinaus stellen generische Services meist größere Informationsbrocken zur Verfügung, als für die Darstellung benötigt werden, d.h., der Bandbreitenbedarf wird größer. Auch ist interpretiertes JavaScript – zumindest zum heutigen Zeitpunkt – nicht für größere Berechnungen geeignet und giert nach den knappen Ressourcen des Clients.

Ein alter Hut

Die Schwächen der beiden Extremansätze legen die Suche nach einer Alternative nahe; eine Alternative, die es erlaubt, Präsentationsaspekte im Browser zu haben, die ganze Logik jedoch im Server auszuführen. Erreicht man dies, so gibt es gleich noch einen Bonus: die einfache Testbarkeit der Applikationslogik unabhängig von jeglicher Client-Technologie. Glücklicherweise muss man nicht lange nach einem entsprechenden Entwurfsmuster fahnden, denn schon lange versteckt der gewiefte Entwickler alle domänen- und applikationsspezifischen Berechnungen hinter einer Applikationsfassade (Application Facade). Zwei spezifischen Varianten der Applikationsfassade begegnet man immer wieder [1]: Presentation Model und Model View Presenter (MVP). (Martin Fowler unterscheidet neuerdings zwei Aspekte von MVP und nennt diese Patterns Supervising Controller und Passive View.) Beiden gemeinsam ist ein ausgeprägtes Verlangen nach Entkopplung von Benutzeroberfläche und Domänenlogik. Und beide können – in Theorie und

Praxis – für Ajax-basierte SPAs eingesetzt werden.

Presentation Model

Ein kleines Beispiel soll das Grundprinzip verdeutlichen. Stellen wir uns eine Applikation vor, die anhand eines Such-Strings nach Büchern sucht und die Ergebnisliste anzeigt. Ein UML-Diagramm der Applikationsfassade und der Buchklasse sieht wie in Abbildung 1 aus.

Die Implementierungsdetails der eigentlichen Suche interessieren den Entwickler der Benutzeroberfläche nicht im Geringsten. Alles, was der grafische Applikationsteil tun muss, ist den Such-String aus einem Eingabefeld zu nehmen, die *searchTitles*-Methode aufzurufen und schließlich das Resultat (die Liste von *Book*-Objekten) in gewünschter Form anzuzeigen.

Diese Fassadenart nennt sich Presentation Model. Die View, d.h. der für die Präsentation zuständige Teil der Applikation, holt sich alle Daten, die sie benötigt, aus dem Model und schickt alle vom Anwender vorgenommenen Änderungen zurück; beides geschieht über das Fassaden-Interface (*BookSearchFacade*). Der Programm- und Datenfluss ist dabei sehr offensichtlich – abgesehen von einer kleinen Komplikation: Da die View Änderungen im Zustand des Models nicht mitbekommt, muss er entweder in regelmäßigen Abständen das Model danach fragen (Polling) oder selbst zum ständigen „Beobachter" werden. So kommt es zum bekannten Observer-Entwurfsmuster (Abb. 2).

Abb. 1: Presentation Model

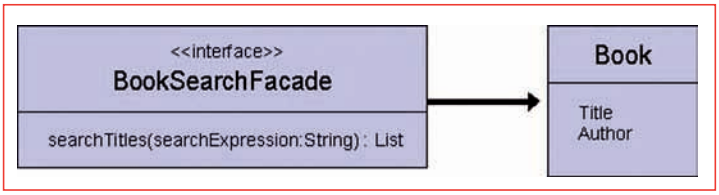


Abb. 2: Observer Pattern

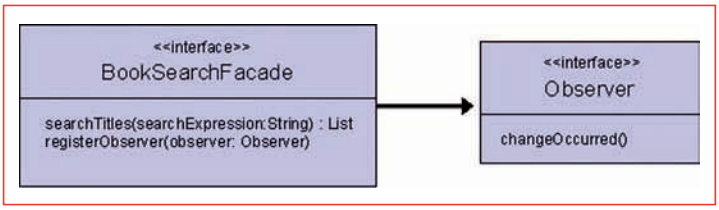
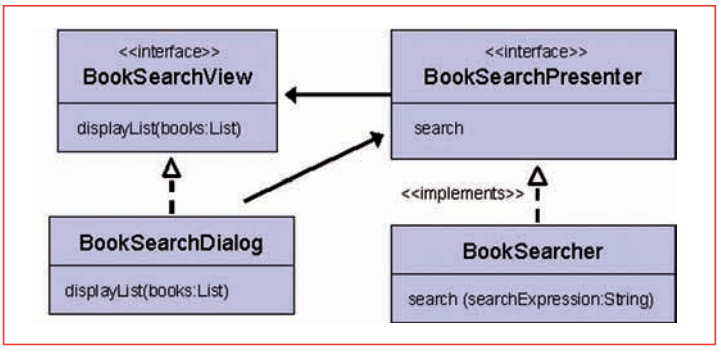


Abb. 3: Model View Presenter



Bei einer Zustandsänderung im Model wird die View mittels *changeOccurred* davon benachrichtigt und kann im Anschluss das Model wiederum nach den Details fragen; meist können sich beliebig viele Beobachter des Models registrieren (*registerObserver*). Das ist im Grunde alles, was man über das Presentation Model wissen muss – zumindest für den Augenblick.

Model View Presenter

Die andere Fassadenvariante, Model View Presenter, möchte dem GUI (alias *Dialog*) noch weniger Verantwortung übertragen. Übrig bleibt nur noch die Aufgabe, jegliche Benutzereingaben an die Fassade (alias *Presenter*) zu signalisieren. Der Presenter ist seinerseits dafür zuständig, die entsprechende Logik an und mit den Domänenobjekten durchzuführen, um anschließend zu entscheiden, wie – und ob überhaupt – die Anzeige (alias *View*) angepasst werden muss. Bei einer solchen Verteilung der Verantwortung sieht das Klassendiagramm ein wenig anders aus (Abb. 3).

Anzeige

Dieser Weg, View und Applikation voneinander zu entkoppeln, wirkt auf den ersten Blick umständlicher und komplizierter als ein geradliniges Presentation Model. Auf den zweiten Blick ergeben sich jedoch einige Vorteile:

- MVP stellt die größtmögliche Trennung zwischen jeglicher Logik – auch der Präsentationslogik – und der eigentlichen Dialogimplementierung her.
- Indem wir die View durch ein Dummy- oder Mock-Objekt ersetzen, können wir Applikationslogik und sogar einen Teil der Präsentationslogik automatisiert – beispielsweise mit JUnit – testen, ohne dabei einen Umweg über das echte User Interface gehen zu müssen.

JayJax

Bei JayJax handelt es sich um ein kleines Ajax-Framework, das sich sowohl auf Client- als auch auf Serverseite tummelt. Das Projekt lebt auf SourceForge [3] und hat vielfältige Ziele:

- Unterstütze ein MVP-basiertes Kommunikationsmodell zwischen Web-Client und (Java)-Server auf möglichst einfache Weise.
- Erlaube die Verwendung von Effekten, fertigen Komponenten und dem ganzen anderen coolen Ajax-Zeugs.ç
- Erlaube die *testgetriebene* Entwicklung von Client- und Servercode.
- Stütze dich dabei auf die zahlreichen guten, existierenden Frameworks und Bibliotheken.
- Mache so wenig Einschränkungen wie möglich betreffend der clientseitigen Umsetzung des Views; vom handgeschriebenem JavaScript-Code über die Verwendung einer fertigen Komponentenbibliothek bis hin zum Java-nach-JavaScript-Crosscompiler soll alles mit JayJax kombinierbar sein.
- Sei kompatibel zu allen (gängigen) Browsern.

JayJax verlangt eine Java 5-kompatible Servlet Engine (z.B. Tomcat 5.5). Das aktuelle Release (0.5.1) lässt natürlich noch Wünsche offen. MVP wird bereits gut unterstützt, die (getestete) Browser-Kompatibilität beschränkt sich bislang auf IE und Firefox. Als Basis der Implementierung dienen Prototype [4] und script.aculo.us [5], aber auch Dojo [6] liegt im zukünftigen Fokus des Projekts. Das JayJax-Team, repräsentiert durch den Autor dieses Artikels, freut sich über Unterstützung und Beteiligung jeglicher Art.

- Die View muss weder die Fassade „pollen“ noch muss sie Daten von ihr holen. Alles Nötige wird ihr von außen, vom Presenter, zugeschoben.
- Die Kommunikation zwischen View und Fassade ist fast vollständig asynchron. Die meisten oder gar alle Methoden an der Presenter-Schnittstelle haben keinen Rückgabewert; und der Presenter verschickt lediglich Nachrichten über gewünschte Anzeigeänderungen an die View.

Die beiden letztgenannten Argumente wiegen im Ajax-Szenario besonders schwer. In verteilten Applikation bedeutet der Verzicht auf Polling das Vermeiden unnötigen Netzwerkverkehrs. Darüber hinaus ist die Kommunikation in Ajax-Applikationen per definitionem asynchron. MVP scheint daher wunderbar geeignet, um die View-Implementierung (*BookSearchDialog*) im Browser zu betreiben und die Presenter-Implementierung (*BookSearcher*) dem Server zu überlassen.

Ein einfaches Programmiermodell

Das Leben eines Entwicklers wird jedoch nicht nur von den verwendeten Prinzipien und Entwurfsmustern bestimmt, sondern maßgeblich davon, wie einfach diese Prinzipien vom Programmiermodell seiner Umgebung (Bibliothek, Framework, Programmiersprache, IDE) unterstützt werden. Idealerweise möchte ich zur Realisierung der MVP-Ideen einige Dinge tun und andere nicht:

- Ich definiere die Presenter-Schnittstelle (z.B. als Java-Interface), implementiere sie auf dem Server und habe auf dem Client ein Proxy-Objekt zur Verfügung, das ich sehr einfach verwenden kann, z.B. *searcher.search('.*AJAX.*');*
- Ich definiere die View-Schnittstelle (z.B. als Java-Interface), implementiere sie für den Client und habe auf dem Server ein Proxy-Objekt zur Verfügung. Auch hier sollte die Verwendung möglichst einfach sein, z.B. *searchView.displayList(listOfRetrievedBooks);*.
- Die Übersetzung von Aufrufen an die Fassade in XMLHttpRequest-Objekte sowie die Entgegennahme und Verteilung entsprechender „Rückrufe“ soll vollständig unsichtbar hinter den Kulissen erfolgen. Als Applikationsentwickler möchte ich mich damit nicht herumschlagen.
- Das unvermeidbare (Un-)Marshalling und (De-)Serialisieren von Nachrichten, Ergebnisobjekten, Exceptions und DTOs soll automatisch geschehen und meine grauen Zellen nicht noch zusätzlich belasten.

Diese Wunschliste lässt sich unglücklicherweise – nach Kenntnisstand des Autors – mit existierenden Frameworks nicht so einfach erfüllen. DWR [2] – siehe auch Artikel auf Seite 44 – beispielsweise unterstützt zwar Methodenaufrufe inklusive Serialisierung und Deserialisierung von clientseitigem JavaScript ins serverseitige Java, doch schon der Rückruf zum Client ist ein recht neues Feature, und die MVP-artige Kopplung der Interfaces und Implementierung würde einiges an projektspezifischer Konfiguration und unter Umständen sogar Codierung erfordern.

Aus diesem Grunde lag die Entwicklung eines kleinen MVP-spezifischen Frameworks auf der Hand, zunächst lediglich als Technologiestudie, mittlerweile als eigenständige Bibliothek. JayJax heißt dieses tausend und erste Ajax-Rahmenwerk (Kasten). Die Umsetzung des obigen Beispiels mit JayJax ist einfach ...

Server-Seite

Das Ziel ist die Realisierung der in Abbildung 3 skizzierten Lösung und zwar derart, dass *BookSearcher* auf dem Server in Java implementiert wird, während unsere View – *BookSearcherDialog* – auf dem Browser in JavaScript umgesetzt wird. Konzentrieren wir uns zunächst auf den Server. Im ersten Schritt definieren wir beide Schnittstellen als einfache Java-Interfaces:

```
import jayjax.IFacade;
public interface BookSearchPresenter extends IFacade {
 void search(String searchString);
}

import java.util.List;
public interface BookSearchView {
 void displayList(List<Book> resultList);
}
```

Die Verwendung des parametrisierten Parameters *List<Book> resultList* dient dazu, die noch nicht existierende Implementierung der Presenter-Schnittstelle ein wenig typsicherer zu machen. In Listing 1 fügen wir die Suchfunktionalität hinzu.

Zu beachten ist, dass *BookSearcher* von *jayjax.AbstractAjaxFacade* abgeleitet wurde. Dies versetzt unser Framework in die Lage, den *BookSearcher*-Objekten den View-Proxy mitzugeben, auf den dann mittels *getView()* typsicher zugegriffen werden kann. JayJax kümmert sich selbstständig um die Weiterleitung aller Methodenaufrufe an den Client. Bei der Spezifikation der Schnittstellen gibt es eine Hand voll Einschränkungen:

- Solange man die Presenter-Methoden in JavaScript mit einem einfachen *presenter.method(par1, par2, ...)* aufrufen möchte, ist ein Überladen von Methoden nicht zulässig, da in JavaScript Methoden ausschließlich anhand ihres Namens unterschieden werden.
- Methoden der View-Schnittstelle können keinen Rückgabewert (oder Exceptions) haben bzw. dieser Rückgabewert wird nie an den aufrufenden Servercode zurückgeliefert, da die Kommunikation vom Server zum Client ausschließlich asynchron abläuft. Tatsächlich stellt dies jedoch im MVP-Kontext keinen Nachteil dar, da sich der Presenter (hier *BookSearcher*) nicht darum kümmern sollte, wie die View denn tatsächlich mit den Update-Nachrichten umgeht.
- Methodenaufrufe an den Presenter können Rückgabewerte liefern. Diese müssen jedoch JavaScript-typisch asynchron, d.h. in einer Callback-Methode verarbeitet werden – etwa so:

```
presenter.method(par1, par2, ... , {onSuccess:
                                    processResult});
function processResult(returnValue) {
 //do something with returnValue
}
```

Und nun können wir Interfaces und Implementierung über Java-Annotations miteinander verknüpfen und dabei auch dem Presenter einen Namen zuweisen (hier: *searcher*), unter dem auf diesen aus JavaScript heraus zugegriffen werden kann:

```
@AjaxFacade(name="searcher",
 view=BookSearchView.class,
 implementation=BookSearcher.class)
public interface BookSearchPresenter extends IFacade {...}
```

Falls die beiden JayJax Servlets und die Fassade korrekt in der *web.xml* registriert wurden – dies ist detailliert in der JayJax-Dokumentation beschrieben –, dann kümmert sich das Framework um folgende Dinge:

- Es generiert „on the fly" JavaScript-Code, der die Presenter-Implementierung als einfaches JavaScript-Objekt auf dem Client verfügbar macht.
- Es generiert JavaScript-Code, der dafür sorgt, dass auf dem Client nichtimplementierte View-Methoden zu einer Fehlermeldung (als Alert-Box) führen.
- Es transportiert alle Methodenaufrufe vom Client zum Server und zurück. Da-

Listing 1

```
import java.util.*;
import jayjax.AbstractAjaxFacade;
public class BookSearcher
  extends AbstractAjaxFacade<BookSearchView>
  implements BookSearchPresenter {
 public void init() {
  getView().displayList(new ArrayList<Book>());
 }
 public void search(String searchString) {
  List<Book> result =
   calculateSearchResult(searchString);
  getView().displayList(result);
 }
 private List<Book> calculateSearchResult(
   String searchString) {
  List<Book> result = new ArrayList<Book>();
  // Retrieve list and add book objects to result
  ...
  return result;
 }
}

public class Book {
 private String title, author;
 /** Needed for deserialization */
 public Book() {}
 public Book(String title, String author) {
  this.title = title;
  this.author = author;
 }
 public String getAuthor() {return author;}
 public String getTitle() {return title;}
}
```

bei kümmert es sich um die notwendige Serialisierung und Deserialisierung; ebenso ist es möglich, auf unterschiedliche Probleme bei der Kommunikation oder bei der serverseitigen Abarbeitung zu reagieren.

Gegenwärtig kommt JayJax mit allen primitiven Typen und den Standardklassen wie *String*, *java.util.List* und *java.util.Date* zurecht. Darüber hinaus lassen sich Objekte, die Javas Bean-Konvention entsprechen, ohne zusätzlichen Programmieraufwand über die Leitung schieben. Für Spezialfälle gibt es zusätzlich die Möglichkeit, eigene Serialisierungs- und Deserialisierungsmechanismen zu programmieren und ins Framework einzuhängen. Hinter den Kulissen arbeitet JayJax mit einem XML-basierten Serialisierungsformat; eine Umstellung auf JSON (JavaScript Object Notation), eine einfache JavaScript-basierte Möglichkeit, Objekte zu serialisieren, wäre jedoch mit wenig Aufwand möglich.

Listing 2

```
public class BookSearcherTest extends TestCase {
 private List<Book> books = null;
 private BookSearcher searcher;
 class DummyView implements BookSearchView {
  public void displayList(List<Book> resultList) {
   books = resultList;
  }
 }
 protected void setUp() throws Exception {
  searcher = new BookSearcher();
  searcher.setView(new DummyView());
 }
 public void testInit() throws Exception {
  searcher.init();
  assertTrue(books.isEmpty());
 }
 public void testSimpleSearch() throws Exception {
  searcher.search("topic");
  //The current implementation always displays 3 books
  assertEquals(3, books.size());
 }
 public void testEmpytSearch() throws Exception {
  searcher.search("");
  assertTrue(books.isEmpty());
  searcher.search(" ");
  assertTrue(books.isEmpty());
  searcher.search(null);
  assertTrue(books.isEmpty());
 }
}
```

Auf dem Client

Um den Client MVP-tauglich zu machen, müssen lediglich einige JavaScript-Bibliotheken und generierte Code-Seiten eingebunden werden:

```
<script type="text/javascript" src="js/prototype.js">
                                                </script>
<script type="text/javascript" src="js/scriptaculous/
                                    scriptaculous.js"></script>
<script type="text/javascript" src="js/jayjax.js"></script>
<script type="text/javascript" src="searcher.gjs">
                                                </script>
<script type="text/javascript" src="all-beans.gjs">
                                                </script>
```

Die Dateiendung *.gjs* weist dabei auf den vom Framework generierten JavaScript-Code hin. Nun lassen sich alle Methoden der Presenter-Schnittstelle auf dem Client in denkbar einfacher Weise verwenden, z.B. *searcher.search($('searchString').value)*;, um die Suche auf dem Server mit dem Wert aus dem Input-Feld mit ID *searchString* zu starten. (Der $-Operator von Prototype liefert das *Dom*-Objekt bei Übergabe einer ID oder des *Dom*-Objekts selbst.) Außerdem müssen noch die Callback-Methoden der View-Schnittstelle implementiert werden:

```
searcher.view.displayList = function(bookList) {
 jayjax.Dom.clear('books');
 bookList.each(function(book){
  var tr = "<tr><td>" + book.title + "</td><td>" +
                                    book.author + "</td></tr>";
  new Insertion.Bottom('books', tr);
 });
}
```

Im Beispiel wird unter Zuhilfenahme einer JayJax- und zweier Prototype-Bibliotheksfunktionen zunächst die HTML-Anzeigetabelle für die Bücherliste gelöscht und anschließend mit allen anzuzeigenden Büchern zeilenweise wieder aufgefüllt. Auf die Properties der *Book*-Instanzen kann dabei über einfache Punkt-Notation zugegriffen werden, während auf Java-Seite die Bean-Konvention (*get/set/is*) zur Anwendung kommt. Fertig! Alle weiteren teuflischen Details des Beispiels lassen sich dem *war*-File mit Beispielanwendungen entnehmen, das der JayJax-Distribution beiliegt.

Testen

Damit die Rede vom testgetriebenen Entwickeln nicht nur eine Floskel bleibt, in Listing 2 noch ein Beispiel im JUnit 3.8-Stil, welche das Testen des Presenters *BookSearcher* mithilfe eines Dummy Views verdeutlicht. Statt der inneren Dummy-Klasse könnte natürlich ebenso gut ein dynamisches Mock-Objekt à la EasyMock [7] zum Einsatz kommen.

Auf JavaScript-Seite sind automatisierte Unit-Tests noch nicht weit verbreitet, dennoch gibt es auch hier mittlerweile einige vielversprechende Ansätze, die sich beispielsweise bei Ajaxian unter dem Stichwort Testing finden lassen [8]. Bei der Entwicklung von JayJax kommt zurzeit das in script.aculo.us enthaltene Test-Framework zum Einsatz.

Fazit

Art und Granularität der Kommunikation sind bei Ajax-basierten Applikationen wichtige Grundsatzfragen, da von den Entscheidungen in diesem Bereich nicht nur das Programmiermodell maßgeblich beeinflusst wird, sondern auch Volumen und Häufigkeit des Netzwerkzugriffs. Der Model-View-Presenter-Ansatz bietet sich für so manche Single Page Application an, denn er erlaubt nicht nur die vollständige Trennung von Logik und Präsentationsaspekten, sondern unterstützt gleichzeitig das in Ajax bevorzugte asynchrone Kommunikationsmuster.

Johannes Link ist Softwareentwickler, -coach und -aktivist, Bücherleser und Buchschreiber. Sein Webtagebuch findet sich auf jlink.blogger.de.

Links & Literatur

[1] Martin Fowler: Development of Further Patterns of Enterprise Application Architecture: www.martinfowler.com/eaaDev/

[2] DWR (Direct Web Remoting): getahead.ltd.uk/dwr/

[3] JayJax: sourceforge.net/projects/jayjax/

[4] Prototype: prototype.conio.net

[5] script.aculo.us: script.aculo.us

[6] Dojo: dojotoolkit.org

[7] EasyMock: easymock.org

[8] Testen auf Ajaxian: ajaxian.com/by/topic/testing/