

Webbasierte Anwendungen

Teil 2

Prof. Kristian Fischer

Dokumentation

Videothek News Manager

“Outtake”

Nico Löbbert 11081575
Oliver van der Bürie 11081751

Vorwort

Im 4. Semester des Studiengangs „Medieninformatik“ an der Fachhochschule Köln, Campus Gummersbach wurde in der Unterrichtung „Webbasierte Anwendungen 2“ für ein studienbegleitendes Projekt ein REST konformer Server inklusive XMPP Publish/Subscribe Funktion und die dazu zugehörigen Clientsoftware entwickelt. Dieses diente parallel zu den Vorlesungen als Praxisprojekt und ist als Gruppenarbeit zu Zweit ausgelegt. Ziel war es, sich mit modernen Webtechnologien wie RESTfull Webservices oder auch dem Jabber Protokoll auseinander zu setzen. Wir bedanken uns an dieser Stelle bei den wissenschaftlichen Mitarbeitern für die fachliche Unterstützung.

Inhalt

<u>1 Aufgabe</u>	Seite 4
<u>1.1 Aufgabenstellung</u>	Seite 4
<u>1.2 Problemfindung</u>	Seite 4
<u>2 Allgemeines</u>	Seite 5
<u>2.1 Umfang</u>	Seite 5
<u>2.2 Gesamtkonzeption</u>	Seite 5
<u>2.3 XML / XSB</u>	Seite 7
<u>2.4 JAXB</u>	Seite 9
<u>2.5 XMPP</u>	Seite 9
<u>3 REST</u>	Seite 10
<u>3.1 Allgemeines</u>	Seite 10
<u>3.2 Server</u>	Seite 12
<u>4 Admin Software “Outtake”</u>	Seite 14
<u>4.1 Viewport</u>	Seite 14
<u>4.2 Interface</u>	Seite 15
<u>4.3 Kommunikation</u>	Seite 16
<u>4.4 Umsetzung</u>	Seite 16
<u>4.5 Umzusetzen</u>	Seite 16
<u>5 XMPP</u>	Seite 19
<u>5.1 Asynchrone Kommunikation</u>	Seite 19
<u>5.2 Server</u>	Seite 19
<u>5.3 Publish / Subscribe</u>	Seite 19
<u>5.4 Publisher</u>	Seite 20
<u>5.5 Client</u>	Seite 21
<u>6 Schluss</u>	Seite 23
<u>6.1 Allgemein</u>	Seite 23
<u>6.2 Restserver</u>	Seite 23
<u>6.3 XML</u>	Seite 23
<u>6.4 JAXB</u>	Seite 24
<u>6.5 XMPP</u>	Seite 24
<u>6.6 Java-Swing</u>	Seite 24
<u>7 Quellen</u>	Seite 25

1 Aufgabe

1.1 Aufgabenstellung

Aufgabe war es, im Rahmen der Veranstaltung Webbasierte Anwendungen eine Problemstellung selbstständig zu identifizieren, eine entsprechende Lösung zu diesem Problem zu konzipieren und daraus ein Projekt zu entwickeln. Vorgegeben war die Programmiersprache JAVA. Zusätzlich sollte eine synchrone Server-Client HTTP Kommunikation, die auf Roy Fieldings REST Paradigma basiert, implementiert werden. Weiterhin sollte zusätzlich ein XMPP Server, basierend auf dem Openfire XMPP Server, aufgesetzt und ein XMPP Client für die "publish/subscribe" Eigenschaften des Servers mittels der Smack API entwickelt werden. Nebenher war eine vollständige Dokumentation des Projekts und ein persönliches Logbuch von den einzelnen Gruppenmitgliedern gefordert.

1.2 Problemfindung

Während des ersten Meetings fanden die Gruppenzusammenstellungen sowie die ersten Brainstromings zur Problemfindung und Aufgabengestaltung statt. Da wir bereits bei früheren Projekten zusammengearbeitet hatten, haben wir relativ schnell eine Gruppe gebildet und uns mit der Aufgabenstellung beschäftigt. Wir sind nach einer kleinen Skizze an der Tafel sehr schnell auf ein Video on Demand System gekommen. Unseren Schwerpunkt hatten wir dabei auf ein ganzheitliches System, welches fachübergreifend entwickelt werden sollte, gelegt. So sollten sowohl Aspekte aus der Android Programmierung als auch Aspekte der Datenbank Technologien, der Softwaretechnik und der Betriebssysteme Anwendung finden. Dazu hatten wir an eine Onlinevideothek mit Streaming-Funktionalität und Online-Verleih, nach dem Vorbild von "LoveFilm" oder "MaxDome", für kleinere Videotheken gedacht. Des Weiteren sollte noch eine Android App entwickelt werden, die ebenfalls alle diese Funktionalitäten der Clientsoftware enthält. Dieses haben wir jedoch frühzeitig aufgrund des zu hohen Aufwands bezüglich der Streamingserver auf das notwendige Minimum einer Videothek Management Software ohne Streaming Client reduzierten.

2 Allgemeines

2.1 Umfang

Während des ersten Pflichttermins wurde in den Gesprächen mit dem Betreuungspersonal der Schwerpunkt auf eine funktionierende, vollständige und logisch durchdachte Software gelegt. Zu diesem Zeitpunkt bemerkten wir, dass der Umfang für unser selbst gewähltes Projekt nur schwer realisierbar ist, weshalb wir bereits hier Abstriche am Konzept vornahmen. Bei jedem weiteren Termin wurden die Schwerpunkte auf nicht zielführende, fehlerhafte Teilespekte unserer Software gelenkt, was den Eindruck vermittelte, dass das eigentliche Ziel der Datenkommunikation zwischen Server und Client sekundär sei. Wir mussten dem Betreuungspersonal immer wieder klar machen, dass die Kommunikation und nicht die detaillierte Funktionalität (nicht verschlüsselt Passwörter, unlogische Programmstrukturen oder allgemein der Detaillierungsgrad der Software) im Vordergrund stehen. Somit hat unser Projekt ein riesiges Ausmaß angenommen, welches nur schwer neben dem Studium und den anderen Studienfächern realisierbar ist. Auch im Hinblick darauf, dass wir uns in alle neuen Komponenten (JAXB, Java HTTP Server, XML, XMPP, Smack, etc.) erst einlesen mussten. Bis heute ist daher lediglich nur die Kernfunktionalität der Server- und Clientsoftware, sprich „GET“, „PUT“, „POST“ und „DELETE“ einzelner Teilespekte umgesetzt. Sowohl die logische interne Programmstruktur als auch die grafische Oberflächen sind auf ein Minimum zur Bedienung o.g. Funktionalitäten und zur Erzeugung bzw. zum Empfangen von XMPP-Nachrichten reduziert worden. Fehlerroutinen mit „try-catch“ und Tests zur Qualitätssicherung nach ISTQB Standard wurden von uns erst gar nicht vorgesehen obgleich diese eine hinreichende Notwendigkeit hätten.

2.2 Gesamtkonzeption

Im Wesentlichen sind von unserem Konzept folgende Punkte übriggeblieben:

- ein Webserver auf Java Basis mit REST Funktionalität
- ein XMPP Server auf Openfire Basis
- ein Admin Client zum Einstellen der Daten und publishen
- ein User Client zum Abruf der Filme und zum Verwalten des Accounts
- ein Mobile Client zum Subscriben und zum Empfangen der XMPP Nachrichten

Hierbei dient der Webserver dazu, den Kunden eine Repräsentation der in der Videothek verfügbaren Filme zu übermitteln. Der Service, den der Webserver bereithält, ist eine Applikation die nach dem REST Paradigma erstellt wurde und ebenfalls einen Warenkorb zum Ausleihen der einzelnen Filme enthalten soll.

Der XMPP Server dient dazu, XMPP Nachrichten auf den Mobile Client asynchron zu senden und somit dem Kunden einen Newsletter mit den aktuell hinzugekommenen Filmen bereit stellt. Der User Client sollte ein einfacher Web Browser sein, damit nicht jeder Kunde noch eine weitere Software auf seinem Computer installieren muss. Aufgrund der Kürze der Zeit konnten wir das allerdings nicht mehr zusätzlich realisieren, da der Server noch kein HTML ausliefert. Der Admin Client wird ebenfalls Browser gestützt sein, wurde jedoch auch erst mal als Java Client konzipiert. Dieser dient dazu um neue Filme in das System eintragen und die Kundenkonten administrieren zu können. Der Mobile Client wird eine für Android geschriebene Applikation, die den Newsletter empfangen und darstellen kann, sein. Im Rahmen unserer Möglichkeiten haben wir uns weiterhin dafür entschieden, alle Server auf ein und denselben Hardware zu installieren. Im produktiven Einsatz würden wir hier eine redundante, verteilte Konfiguration bevorzugen. Das Datenaustauschformat wird ausschließlich XML sein. Grundsätzlich würden wir für den Datenaustausch zwischen Server und Browser das JSON Format bevorzugen. Der Ablauf unseres verteilten Systems wird im folgenden Diagramm erläutert.

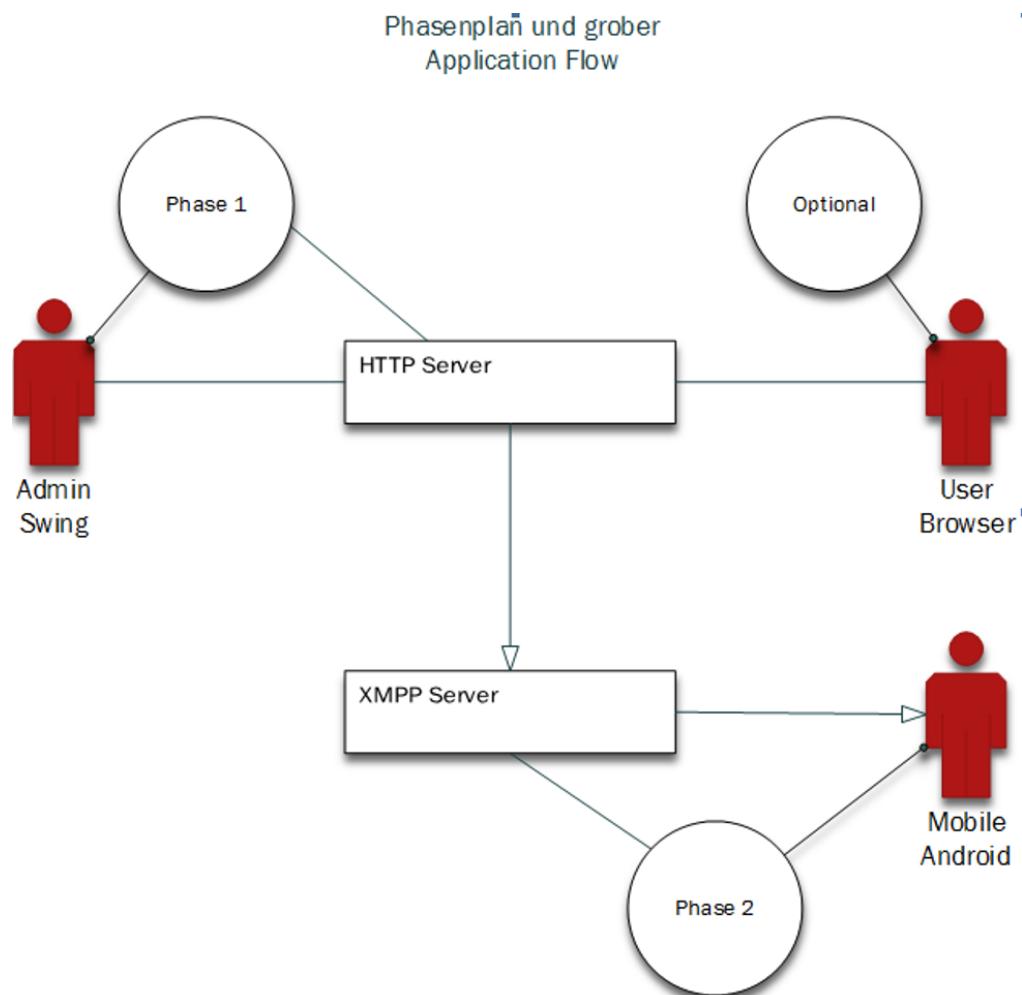


Abb: 2.1 Kommunikation

2.3 XML / XSB

XML ist als eine Untermenge von SGML anzusehen. Die Entwickler des W3C um Jon Bosak (Sun Microsystems) haben ca. 1996 festgestellt, dass SGML eine hoch komplexe, sehr mächtige aber auch sehr überladene Auszeichnungssprache ist. Daher haben sie eine Essenz aus dieser Sprache extrahiert, aus der XML entstanden ist.

Beim Aufbau eines XML Dokumentes sollten folgende Dinge beachtet werden:

Jede XML Datei benötigt als erstes eine XML Deklaration <?xml version="1.0" ?>. Hiermit wird festgelegt, um welche XML Version es sich bei dem verwendeten Dokument handelt. Weitere Parameter, z.B. das Encoding Encoding="UTF-8", sind optional.

Danach kann optional eine DTD (Document Type Definition) in die Datei eingebunden werden. Diese kann entweder als eine externe Datei eingebunden oder direkt in der Datei definiert werden. Die Empfehlung ist, eine DTD immer einzubinden, denn ohne DTD können XML Dateien zwar wohlgeformt sein, aber sie können nicht gegen eine DTD validiert werden.

Nun muss ein Root-Knoten respektive ein Root-Tag erstellt werden, in das alle anderen Elemente eingehangen werden können. Man kann einen Vergleich zu dem von HTML bekannten <html></html> Tag ziehen.

Nun können weitere Elemente in das Root-Tag eingebunden und beliebig, jedoch unter folgenden Regeln verschachtelt werden:

- Elemente müssen eine geschlossene Hierarchie haben. <p></p> ist nicht erlaubt.
- Elemente müssen immer geschlossen werden. <p></p>
- Elemente sind Case sensitive. <p></P> führt zu einem Fehler.
- Leere Elemente müssen geschlossen werden.

- Attributwerte müssen in Anführungszeichen stehen.
 - <p c=1></p> = Falsch
 - <p c="1"></p> = Richtig
- Es gibt keine boolschen "Schalter". <option value="gm" checked>foo</option> ist nicht möglich. Wenn dann checked="checked".

Die XML Strukturen wurden von uns zuerst ohne Rücksicht auf mögliche Resourcen und Links entwickelt. Dieses sollte später noch zu einem Problem werden, durch das wir die gesamte XML Struktur nochmals überprüfen und komplett überarbeiten mussten. Anfangs haben wir uns nur auf die benötigten Daten in der Application / dem Service konzentriert, ohne die REST Komponente in die Betrachtung mit einzubeziehen. Daher bestand unser erster Entwurf rein aus Elementen. Im Rahmen einer ersten Überarbeitung haben wir das komplette XML auf Attribute umgestellt. Dieses hatte im Bezug auf den Overhead einige Vorteile. Allerdings wurde dadurch die Übersichtlichkeit in Mitleidenschaft gezogen. De facto könnte man unsere Applikation sowohl mit Elementen, als auch fast ausschließlich mit Attributen realisieren. Generell kann man allerdings sagen, dass wir in diesem Fall eher auf das JSON Format zurückgegriffen hätten. Dadurch würde man zwar einerseits eine gewisse Formatsicherheit verlieren, andererseits aber ein wesentlich kompakteres Datenformat für den Datenaustausch zwischen Client und Server erhalten. Nach dem zweiten Pflichttermin ist uns aufgefallen, dass die XML Vorlagen nicht REST konform sind, da sie keine Backlinks und auch keine Links auf Subressourcen enthielten. Und da REST bekanntlich ein zustandsloses Verfahren ist respektive auf dem zustandslosen HTTP aufsetzt, mussten wir diesen Fehler noch mühselig ausbessern, was uns einiges an Zeit gekostet hat. Diesen Fehler konnten wir dadurch leider noch nicht bei allen XML Dateien aufgrund des Zeitmangels vollständig ausbessern. Bei der Überprüfung der XML Schema Dateien ist uns ebenfalls aufgefallen, dass wir diese noch wesentlich restriktiver hätten schreiben können. Für eine erste Implementierung und Funktionalität der JAXB Klassen / Resourcen haben wir allerdings darauf verzichtet.

2.4 JAXB

Die Implementierung des JAXB als Java Bibliothek stellte uns nicht vor sonderlich große Probleme. Wir wurden allerdings vor allem durch die Komplexität des Marshalling / Unmarshalling überrascht. Was bei PHP ein simples “SimpleXMLElement()“ oder “simplexml_load_file()“ wäre, ist bei Java das komplette unmarshalling Verfahren. Es hat ein wenig Zeit in Anspruch genommen, das Konzept hinter dieser Technik zu verstehen. Ebenfalls hat uns irritiert, dass man nicht einfach ein einzelnes Element aus einem Marshal-Object herausziehen oder hinzufügen kann, ohne zumindest eine Factory Funktion oder ein händisches Marshalling für dieses Objekt zu kreieren. Dieses wäre unserer Meinung nach in anderen, jüngeren Programmiersprachen ebenfalls um einiges leichter gewesen. Da wir in unsere Applikation nicht eine gesonderte Datenbank mit einbinden wollten, obgleich das in einem produktiven Einsatz sicherlich sehr sinnvoll sein würde, haben wir versucht die XML Elemente/XML Dateien als Datenspeicher für unsere Applikation zu verwenden. Das Iterieren über die Elemente oder der Zugriff auf die Elemente stellten wir uns anfangs wie einen Zugriff auf eine Datenbank vor. Wir konnten jedoch relativ schnell feststellen, dass dieses in der Form nicht möglich ist. Diese Erkenntnis hat uns ebenfalls noch mal etwas Zeit gekostet, da wir relativ viel ausprobiert mussten.

2.5 XMPP

Im Gegensatz zu JAXB respektive den XML/XSB Dokumenten, mit welchen wir vorher schon gearbeitet hatten und einiges aus unserer beruflichen Erfahrung herausziehen konnten, stellte das XMPP Protokoll komplettes Neuland für uns dar. XMPP basiert zwar auf XML, jedoch kannten wir den kompletten Kommunikationsablauf zwischen Server und Client nicht. Weiterhin stellte sich uns die Frage, was es exakt mit dem Publish / Subscribe Verfahren auf sich hatte. Das Einarbeiten in dieses Protokoll inklusive dem Einarbeiten in die zugehörigen Java Klassen (hier Smack) hat uns enorm viel Zeit gekostet. Insbesondere hat es uns enorm viel Zeit gekostet, dass eine Methode innerhalb der Smack Klasse einen Fehler enthält, der außer in einem Forum für Softwareentwickler nirgends dokumentiert ist. Aufgrund dieses Fehlers haben wir versucht, die Software innerhalb von zwei Wochen mittels Trail & Error Programmierung zu der gewünschten Funktionalität zu bewegen. Dieses ist uns leider nicht gelungen. In einem produktiven Arbeitsumfeld würden wir an dieser Stelle den Fehler innerhalb der Smack Klasse suchen und diesen dort beheben, da wir davon ausgehen, dass wir wesentlich mehr Zeit für ein solches Projekt zur Verfügung hätten und Zeit, den Fehler zu suchen.

3 REST

3.1 Allgemeines

REST Ist kurz gesagt ein Architektur-Stil und eine Abstraktion für verteilte Hypermedia-Systeme. Als solches ist REST nicht Service orientiert sondern Resourcen orientiert. REST ist hingegen keinen Technologie, kein Protokoll und kein zweites SOAP. Der Begriff REST trat erstmals im Jahre 2000 auf und wurde durch Roy Fielding in seiner Dissertation eingeführt.

REST Prädigma

Ressourcen eindeutig identifizierbar.

Eine Resource ist ein Objekt, welches ein Entwickler als "eindeutig identifizierbar" in einen Service implementieren will. Das heißt konkret, dass eine Resource sowohl ein statisches HTML Dokument, als auch ein dynamischer Webservice, ein Datensatz, eine Funktion/Methode oder gar ein ganzer Service sein kann. Eine Resource kann durch diverse Requests abgerufen, angelegt, verändert oder gelöscht werden. Die eindeutige Identifikation der Resource wird durch die URI (Uniform Resource Identifier) sichergestellt.

Verwendung von Hypermedia

Hypermedia bezeichnet die Verlinkung von Dokumenten untereinander. Im Sinne von REST würde man eher die Verlinkung von Resourcen verstehen. Eine Resource kann wiederum Unterressourcen haben und verweist auf diese. Somit entsteht ein Resourcen-Baum durch den sich ein User oder auch ein Computer iterieren kann.

Verwendung standardisierter Methoden

Nach der Dissertation von Roy Fielding sind diese Methoden vom Protokoll HTTP abgeleitet / übernommen. Diese sind hauptsächlich GET, POST, PUT, DELETE und HEAD und werden von jedem Browser verstanden.

Mehrere Repräsentationen

Eine Repräsentation stellt das Datenformat einer Resource dar, welches vom Server im Response zurückgeliefert wird. Dabei kann eine einzelne Resource mehrere Repräsentationen haben. Zum Beispiel kann die Beschreibung eines Autos (Marke = Tatra, Farbe = Grün, etc.) sowohl als vollständiges HTML Dokument, als auch im XML, im JSON oder in sonstigen Formaten zurück gegeben werden. Um eine bestimmte Form der Repräsentation abzurufen wird entweder ein Accept: <Format> HTTP Header Field beim Request mit übertragen. Danach kann der Server die passende Representation zurückliefern. Eine weitere Möglichkeit besteht darin, die Representation als Dateiendung in den Request zu verpacken. Bsp: localhost/auto.html oder localhost/auto.xml.

Statuslose Kommunikation

Statuslose Kommunikation bedeutet, dass der Server keine Angaben über den Clientstatus speichert. Der Server "trennt" die Verbindung sofort nach erfolgreichem Abarbeiten eines Requests respektive nach erfolgtem senden des Response. Er kann den Client allerdings dennoch identifizieren. Z.B. durch Cookies. Diese Technik hat jedoch nichts mit dem Begriff statuslos zu tun. Statuslos ist zum Beispiel auch das HTTP Protokoll.

3.2 REST Server

Der Server sollte ein in JAVA geschriebener HTTP Server mit REST Funktionalität sein. Laut Projekt.- und Aufgabenbeschreibung sollten wir einen Grizzly HTTP Server implementieren. Auf Nachfrage, ob es denn tatsächlich ein Grizzly Server sein muss, wurde uns der HTTP Server freigestellt. Wir haben uns dann für den in Java eingebundenen HTTP Server entschieden, da dieser mit sehr geringem Aufwand einzubinden war. Somit haben wir uns das Einarbeiten in die Eigenheiten des Grizzly HTTP Server erspart und konnten diese Zeit in die Entwicklung der Ressourcen und das Einarbeiten in die Eigenheiten der Jaxb Klassen investieren. Grundsätzlich funktioniert die in Java eingebundene HTTP Server Funktionalität genau wie die des Grizzly-Servers. Mittels spezieller Annotation werden innerhalb des Programmcode einzelne Resources benannt. Diese Annotationen werden grundsätzlich vor einzelne Funktionen in der Klasse selber geschrieben respektive einmalig vor die Klassendefinition und enthalten kein Schließen des Semikolon. So wird ein Pfad zu einer Ressource mittels folgender Annotation deklariert:

```
@Path("messages")
```

Nach einem Aufruf des Pfades „/messages“ unter der Domäne, auf der der Server läuft, wird somit diese Funktion innerhalb der Java Klasse angesprochen. Um die einzelnen Ressourcen noch weiter spezifizieren zu können, werden folgende Annotationen gebraucht. Für die einzelnen REST konformen Operationen die Annotationen:

```
@GET  
@POST  
@PUT  
@DELETE
```

Diese Annotationen werden zusätzlich zu der Pfad Annotation vor die einzelnen Funktionen innerhalb der Klasse geschrieben. Somit kann eine Funktion mehrerer Annotationen enthalten. Weiterhin sind möglich:

```
@Consumes(MediaType.APPLICATION_XML)  
@Produces(MediaType.APPLICATION_XML)
```

Diese dienen dazu um das akzeptierte und zurückgegebene Datenformat, den sogenannten „Content-Type“, das durch den HTTP Header übergeben wird, zu spezifizieren.

Um auf die einzelnen Parameter des Pfades oder der Query zuzugreifen, können innerhalb eines Pfades Platzhalter definiert werden. Innerhalb der Pfadannotation werden diese Platzhalter durch geschweifte Klammern repräsentiert. Um innerhalb der Funktion auf diese Platzhalter zugreifen zu können, werden die Annotationen `@PathParam` für Pfad Parameter sowie `@QueryParam` für Query Parameter benötigt. Diese werden vor der Typen Deklaration der Parameter Variablen notiert. Ein Beispiel:

```
@Path("{date}")
@PathParam("date") String date)
```

Konkret kann dann eine vollständige Funktion wie folgt aussehen:

```
@GET
@Path("{date}")
@Produces(MediaType.TEXT_HTML)
public Response get(@PathParam("date") String date){
    return Response.status(Response.Status.OK).entity("Hello World").build();
}
```

Innerhalb der einzelnen Funktionen können dann sämtliche Java Objekte, in unserem Fall JAXB Elemente, genutzt werden. Die Rückgabe erfolgt durch eine Response Klasse. Diese hält Methoden und Funktionen vor, mit denen sowohl der Rückgabestatus als auch der Rückgabe-Body einfach gesetzt werden können.

4 Admin Software “Outtake”

Ist eine simple administrative Clientsoftware für den Restserver, die die Standartfunktionalitäten „GET“, „PUT“, „POST“ und „DELETE“ bedient. Sie ist nicht für den Mehrnutzerbetrieb gedacht, kann aber nach dem Prinzip „Last Save Wins“ auch dafür missbraucht werden. Eine Authentifizierung erfolgt über das Betriebssystem und wird nicht erneut von der Software gefordert, damit entfällt auch eine Protokollierung von Änderungen.

4.1 Viewport

Um eine schnelle und einfache Portierung auf Android zu ermöglichen ist „Outtake“ nach dem Paradigma von Computerspielen umgesetzt. Gemeint ist die Ausgabe eines aktuell sichtbaren Level-Abschnitts in einem Viewport, vergleichbar mit einem Bild im Bilderrahmen. Hierfür beherbergt ein „JPanel“ alle interagierenden Komponenten und wird lediglich, innerhalb eines „JFrames“, horizontal oder vertikal verschoben. Diese Methode zwingt uns dazu die Software wie einen Eingabeassistenten zu konzipieren, was die Bedienung der Oberfläche vereinfacht und somit selbsterklärend wirkt. Darüber hinaus werden die Bewegungsabläufe von Smartphones imitiert, was wiederum bei einer möglichen Portierung von Vorteil ist. Sollte es tatsächlich zu einer Portierung der Software auf Android kommen, muss lediglich der Viewport neu entwickelt werden und der Rest kann mit geringfügigen Änderungen übernommen werden.

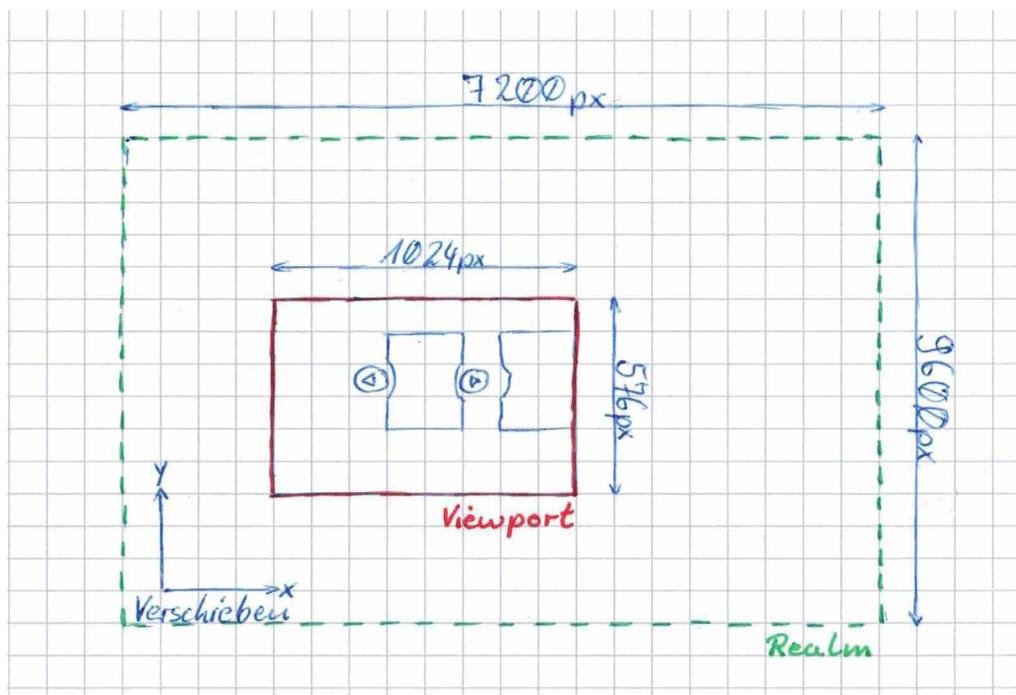


Abb. 4.1 Viewport

4.2 Interface

Da die Gestaltung von „Outtake“ an Eingabeassistenten orientiert ist, haben wir uns eine interessante Metapher als grafisches Highlight überlegt. Angedacht ist eine optische Aufbereitung der einzelnen Ein- und Ausgabemasken als Puzzle-Teile, die sich in vier Himmelsrichtungen aneinanderreihen. Jedoch macht uns die Java-Swing-Umgebung einen Strich durch die Rechnung und weigert sich penetrant unsere entworfenen Grafiken richtig darzustellen.

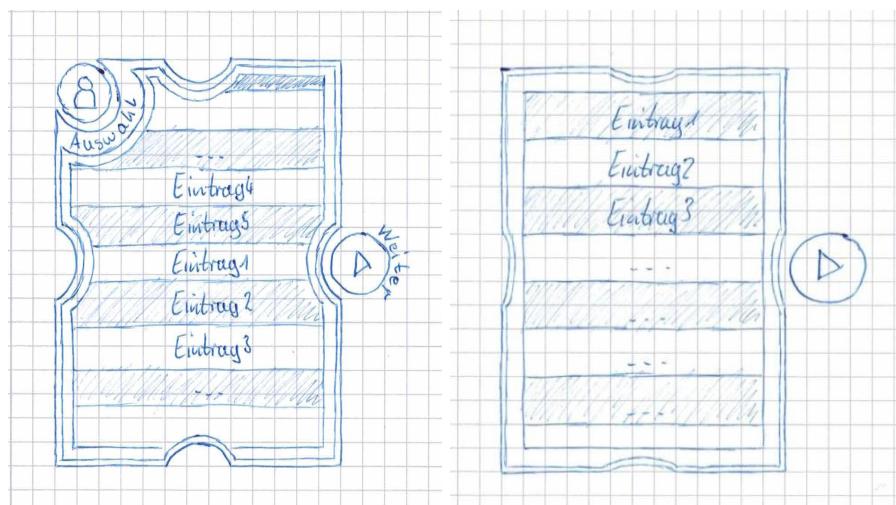


Abb. 4.2 Erstentwurf

Abb. 4.3 Zweitentwurf

Nach zwei Wochen vergeblicher Entwicklungsversuche konnten wir den Fehler auf die grafischen Ausgaberoutinen „paint“, „paintComponent“ und „paintComponents“ einkreisen. Bei einer Interaktion mit der grafischen Oberfläche, wird der interagierende Bereich neu ausgegeben. Dabei überschreibt die Komponente alle anderen grafischen Darstellungen, was zur einer Missgestaltung der Oberfläche führt. Dieser Fehler ließe sich lediglich mit einer gezielten Aufforderung aller Komponenten zur erneuten Ausgabe beheben. Weil jedoch die „paint“ Methoden lediglich auf das übergeordnete Objekt zugreifen können, kommt es an dieser Stelle zu einer endlosen Rekursionsschleife.

Mangels Projektzeit können wir uns nicht weiter mit diesem Problem beschäftigen, weshalb wir uns entschlossen haben eine abgewandelte, nicht so ausgereifte und detaillierte Version der grafischen Oberfläche umzusetzen. Ihr Vorteil liegt darin, dass es zu keinerlei Überschneidungen mit den Java-Swing-Objekten kommt und somit der o.g. Fehler umgangen wird. Der Nachteil ist ganz klar darin zu sehen, dass eine Menge an Informationen, die der Selbsterklärung der Software dienen, verloren gehen.

4.3 Kommunikation

Es werden mittels der Java-Standard-Funktion „`HttpURLConnection`“ synchronisierte Anfragen an den Restserver gesendet. Dies bedeutet, dass der Client solange die Verbindung zum Server aufrecht erhält bis seine Anfrage mittels „`OutputStream`“ versendet und die Antwort mittels „`InputStream`“ empfangen ist. In dieser Zeit wird die Software blockiert und der Benutzer kann nicht weiterarbeiten. Nach unserem Dafürhalten würde hierbei eine teils asynchrone Kommunikation durchaus wesentlich mehr Sinn machen. Da wir uns bei dem Client allerdings vorerst auf eine reine REST Funktionalität beschränkt haben, haben wir den synchronen Ansatz gewählt, da dieser wesentlich einfacher zu implementieren war. Bei unserer Weiterentwicklung der Applikation würde das Ganze dann so aussehen, dass alle Funktionalitäten und Design-Elemente innerhalb der Applikation unabhängig von der Kommunikation ausgeführt werden können.

4.4 Umsetzung

Zwecks Arbeitersparnis besteht der Client lediglich aus einem „`JFrame`“, einem „`JPanel`“ und einer Vielzahl an „`JLabel`“, „`JButtons`“ bzw. „`JListens`“. Alle Interaktionen werden durch die Listen- und Button-Events gesteuert, was die ganze Umsetzung um ein vielfaches erleichtert.

4.5 Umzusetzen

Es gibt keinerlei Sicherungs- und Fehlerbehandlungs routinen, noch ist der korrekte interne Arbeitsablauf umgesetzt. Diesbezüglich fallen noch folgende Arbeiten an:

- Realisierung der Verleih-Funktionen beim Erzeugen von Rechnungen
- Interne Verwaltung von Nachrichten (zeitgesteuertes Löschen etc.)
- Sicherungsroutinen bei Verlust der Serververbindung
- Sicherheitsabfragen beim Löschen von Einträgen
- Fehlerkorrekturen für Benutzereingaben

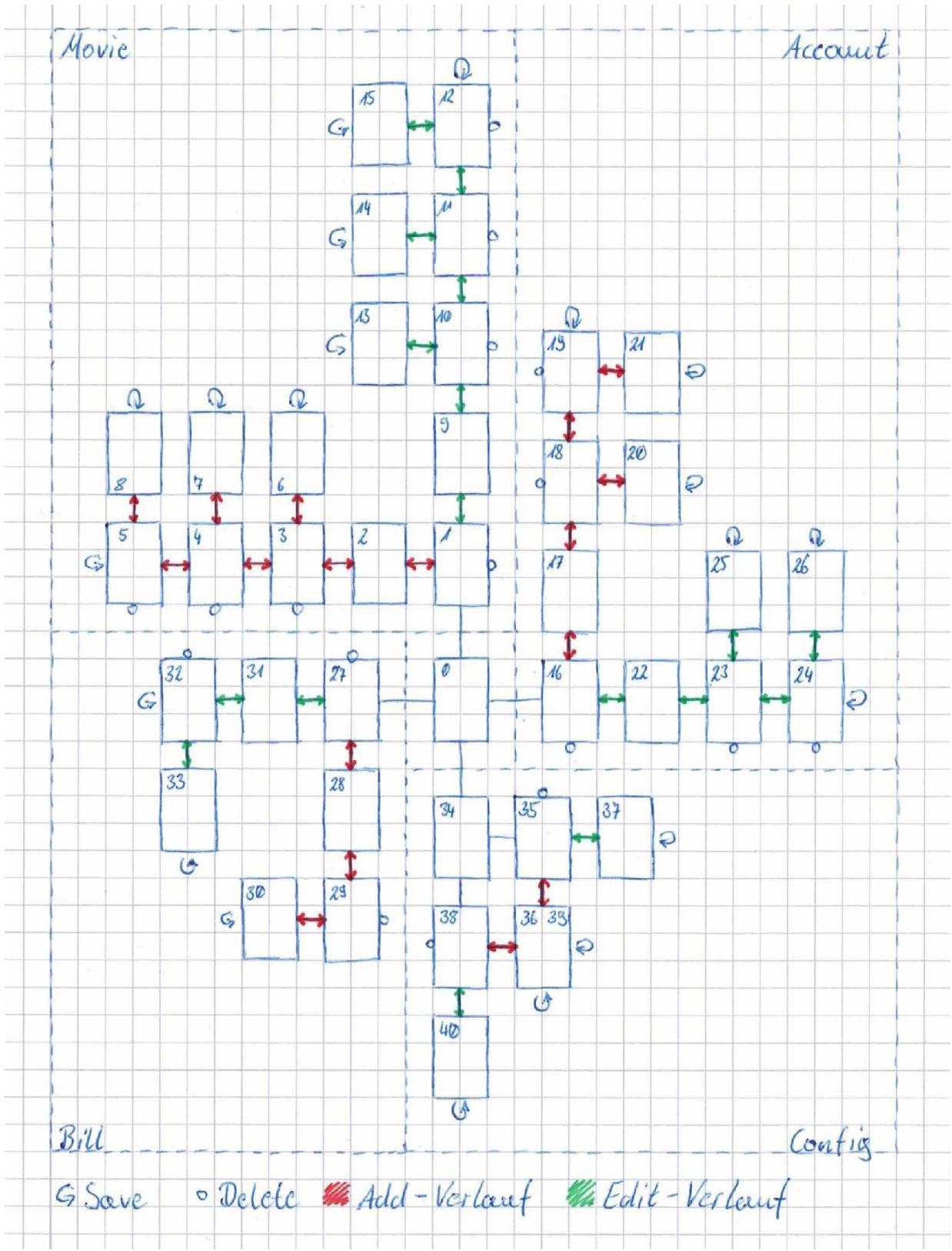


Abb. 4.4 Navigation

- 0: Hauptmenü
- 1: Ausgabeliste „<movies>“
- 2: Hinzufügen – Eingabemaske „<movies><movie>“
- 3: Hinzufügen – Ausgabeliste „<movies><movie><stocks>“
- 4: Hinzufügen – Ausgabeliste „<movies><movie><genres>“
- 5: Hinzufügen – Ausgabeliste „<movies><movie><actors>“
- 6: Hinzufügen – Eingabemaske „<movies><movie><stocks><stock>“
- 7: Hinzufügen – Auswahlliste „<genres>“ erzeugt „<movies><movie><genres><genre>“
- 8: Hinzufügen – Auswahlliste „<actors>“ erzeugt „<movies><movie><actors><actor>“
- 9: Bearbeiten – Editiermaske „<movies><movie>“
- 10: Bearbeiten – Ausgabeliste „<movies><movie><stocks>“
- 11: Bearbeiten – Ausgabeliste „<movies><movie><genres>“
- 12: Bearbeiten – Ausgabeliste „<movies><movie><actors>“
- 13: Bearbeiten – Editiermaske „<movies><movie><stocks><stock>“
- 14: Bearbeiten – Auswahlliste „<genres>“ erzeugt „<movies><movie><genres><genre>“
- 15: Bearbeiten – Auswahlliste „<actors>“ erzeugt „<movies><movie><actors><actor>“
- 16: Ausgabeliste „<accounts>“
- 17: Hinzufügen – Eingabemaske „<accounts><account>“
- 18: Hinzufügen – Ausgabeliste „<accounts><account><genres>“
- 19: Hinzufügen – Ausgabeliste „<accounts><account><wishes>“
- 20: Hinzufügen – Auswahlliste „<genres>“ erzeugt „<accounts><account><genres><genre>“
- 21: Hinzufügen – Auswahlliste „<movies>“ erzeugt „<accounts><account><wishes><movie>“
- 22: Bearbeiten – Editiermaske „<accounts><account>“
- 23: Bearbeiten – Ausgabeliste „<accounts><account><genres>“
- 24: Bearbeiten – Ausgabeliste „<accounts><account><wishes>“
- 25: Bearbeiten – Auswahlliste „<genres>“ erzeugt „<accounts><account><genres><genre>“
- 26: Bearbeiten – Auswahlliste „<movies>“ erzeugt „<accounts><account><wishes><movie>“
- 27: Ausgabeliste „<bills>“
- 28: Hinzufügen – Auswahlliste „<accounts>“ erzeugt „<bills><bill>“
- 29: Hinzufügen – Ausgabeliste „<bills><bill><movie>“
- 30: Hinzufügen – Auswahlliste „<movies>“ erzeugt „<bills><bill><movie>“
- 31: Bearbeiten – Editiermaske „<bills><bill>“
- 32: Bearbeiten – Ausgabeliste „<bills><bill><movie>“
- 33: Bearbeiten – Auswahlliste „<movies>“ erzeugt „<bills><bill><movie>“
- 34: Konfigurationsmenü
- 35: Ausgabeliste „<genres>“
- 36: Hinzufügen – Eingabemaske „<genres><genre>“
- 37: Bearbeiten – Editiermaske „<genres><genre>“
- 38: Ausgabeliste „<actors>“
- 39: Hinzufügen – Eingabemaske „<actors><actor>“
- 40: Bearbeiten – Editiermaske „<actors><actor>“

5 XMPP

5.1 Asynchrone Kommunikation

Zur Aufgabenstellung gehörte die Entwicklung eines asynchronen XMPP Clients unter Verwendung der Openfire Smack Library. Als Server sollte der Openfire XMPP Server dienen. Weiterhin sollte das System für ein Publish-Subscribe Verfahren konzipiert werden. Grundsätzlich kann gesagt werden, dass das XMPP Protokoll ein Protokoll für Instant Messaging, Chats, Voice.- und Video-Calls ist. Das von uns benötigte Publish / Subscribe Verfahren ist nur eine Unterkategorie von XMPP respektive im Server ein Plugin. Alle anderen Funktionalitäten des XMPP Protokolls wurden von uns nicht benötigt. Ebenfalls benötigten wir die Funktionen der Nutzerverwaltung nur rudimentär.

5.2 Server

Das Aufsetzen des Openfire XMPP Servers geschieht nach den Vorgaben in der Openfire Dokumentation. Bei der Konfiguration haben wir uns dafür entschieden, die Nutzer des Servers über die Weboberfläche per Hand einzutragen. Ursprünglich war angedacht, dass sich neue User selbst an dem XMPP Server einen Account erstellen und diesen verwalten können. Aufgrund von massiven Problemen mit dem sogenannten „publish/subscribe“, die wir im Schwerpunkt behandeln wollten, haben wir diese Funktionalität in unserem Prototyp vorerst ad acta gelegt. Weiterhin haben wir für unser Projekt die in Openfire implementierte Datenbank verwendet. Es wäre zwar ein leichtes gewesen, eine externe Datenbank in den Server einzubinden. Allerdings haben wir uns diesen zusätzlichen Aufwand absichtlich erspart, da wir davon ausgegangen sind dass dieses nicht die Hauptaufgabe des Projekts ist. In einem Produktiv-System würden wir selbstverständlich auf andere Datenbank Technologien ausweichen.

5.3 Publish / Subscribe

Beim Publish / Subscribe Verfahren handelt es sich um eine asynchrone Kommunikation bzw. einem asynchronen zur Verfügung stellen von Informationen. Zur Vorbereitung des Verfahrens müssen auf dem Server sogenannte „Nodes“ erstellt werden. Diese Elemente unterscheiden sich in „CollectionNodes“ und „LeafNodes“. Mittels „CollectionNodes“ können verschiedene Elemente, ähnlich einer Verzeichnis oder Baum Struktur, gruppiert werden. Sie können sowohl weitere „CollectionNodes“ als auch „LeafNodes“ enthalten. „LeafNodes“ dienen als Speicher für die eigentlichen Informationen. Sie können keine weiteren „Nodes“ Elemente enthalten und

bilden, wie der Name schon sagt, die Blätter des Baumes bzw. die untersten Elemente. Um Information zu verbreiten können in eine „LeafNode“ sowohl leere „Items“ als auch komplett XML Strukturen als „PayLoad“ hinzugefügt werden. Im Rahmen unseres Projekts haben wir ausschließlich mit „PayLoadItems“ gearbeitet, da unsere XML Strukturen aus den XML Daten des REST Server mittels JAXB generiert werden. Beim „publish“ werden die Informationen durch einen Nutzer auf dem Server veröffentlicht. Und diese Informationen abrufen zu können, müssen sich andere Nutzer auf dem Server für die entsprechende „Node“ eintragen („subscribe“). Sobald sich ein Nutzer auf dem Server eingetragen hat, versendet der Server automatisch die entsprechenden Informationen, wenn ein neues Item veröffentlicht wurde. Hierbei ist es egal, ob der entsprechende Client zur Zeit Online ist oder nicht. Damit ein Nutzer eine Informationen veröffentlichen oder empfangen kann, benötigt er eine Verbindung zum Openfire Server. Diese Verbindung wird über eine Connection Klasse hergestellt. Damit dieses erfolgreich ist, benötigt der Nutzer einen freigeschalteten Account auf Server-Seite sowie seine Login Daten auf Clientseite. Um sich in eine LeafNode einschreiben zu können, wird weiterhin die sogenannten JID („Jabber ID“) benötigt. Diese beläuft sich in unserem Fall auf <Nutzername>@<Domäne>. Mit diesen Informationen kann sich ein Nutzer wegen jede Node einschreiben.

5.4 Publisher

Der XMPP Publisher ist ein in Java geschriebenes Programm, welches in den entwickelten REST Server eingebunden wird. Es wurde unter Zuhilfenahme der Java Openfire Smack Library programmiert. Der Publisher dient dazu, neue Nachrichten in den XMPP Openfire Server einzutragen bzw. zu veröffentlichen („publish“). Das Einbinden erfolgt mittels einer Funktion innerhalb der Ressource „Message“ („MessageResource.java“) des REST Servers. Mit Aufruf der Ressource „Message/{date}/send“ marshaled diese Funktion die zuletzt in den REST Server eingetragenen Nachrichten mit dem Selektor {date} und veröffentlicht diese auf dem XMPP Openfire Server unter der LeafNode „MovieNews“ als neues Item. Wir sind den Umweg über einen expliziten Selektor für die einzelnen Id's der neuen Messages und einer expliziten Ressource „/send“ gegangen, um einzelne Nachrichten versenden zu können. Ursprünglich war angedacht, dass auch mehrere Items zur gleichen Zeit veröffentlicht werden können. Durch die fehlerhafte Smack Klasse wäre es zwar möglich gewesen, dieses so durchzuführen, allerdings würde durch den Server nur immer das letzte Element an die Nutzer gesendet. Problematisch bei der Erstellung des XMPP Publishers war u.a. ein Fehler, der von uns auf Anhieb nicht gefunden werden konnte. Die Übertragung der XML Daten mittels der XMPP Smack Api erfordert, dass die XML Daten ohne die dazugehörige XML Deklaration vor dem Root-Knoten an die Methode zum Publishen übergeben werden. Sollte eine XML Deklaration noch vor dem Root-Knoten vorhanden sein, so werden die Daten zwar an dem Server gesendet, werden dort aber nicht gespeichert und der Server wirft eine Exception. Da wir unsere XML Daten allerdings durch die JAXB Klassen haben generieren lassen, wurde auch jedes Mal die XML Deklaration

hinzugefügt. Diesen Umstand haben wir erst nach mehrstündiger Fehlersuche feststellen und beheben können.

5.5 Client

Der XMPP Client dient einerseits dazu, sich auf dem Openfire XMPP Server einschreiben („subscribe“) zu können. Andererseits dient er zum Empfang der veröffentlichten Nachrichten. Angedacht war die Umsetzung als Android App mit der grafischen Gestaltung von „Outtake“ für den Nutzer. Dieses ist jedoch mangels Projektzeit verworfen worden. Unterdessen haben wir ein kleines Java Programm unter Verwendung der Swing Library entwickelt, welches zumindest die empfangenen XMPP Nachrichten auf dem Desktop anzeigen kann. Bei der Entwicklung des XMPP Client traten die größten Probleme des Gesamtprojekts auf. Wir hatten das Szenario entwickelt, dass neue Informationen und Items veröffentlicht werden, während ein Nutzer längerer Zeit nicht online ist. Damit ein Nutzer letztendlich alle Informationen empfangen kann, war angedacht, dass beim nächsten Login sämtliche noch nicht abgeholteten Nachrichten empfangen werden. Wir konnten dann mittels des Publishers mehrere Items auf dem Server veröffentlichen. Nachdem der Nutzer sich eingeschrieben hatte bzw. wieder online gekommen ist, wurde allerdings regelmäßig nur das letzte Item zurückgegeben. Wir haben die verschiedensten Methoden zur Fehlerbehebung ausprobiert. Wir haben versucht mit sämtlichen Formen der Java Collections über die bestehenden Items zu Iterieren. Wir haben die gesamte Konfiguration der Node geändert, von persistenten Items über die Einstellung „DeliverPayloads“ bis hin zu dem AccessModel. Wir haben insgesamt etwas mehr als zwei Wochen an einem kleinen Code Schnipsel von ungefähr 20 Zeilen gearbeitet. Letztlich haben wir durch Zufall in einen Internetforum für Programmierer einen Beitrag gefunden, der besagt, dass es zu Schwierigkeiten zwischen den einzelnen Versionen des Server und der Smack API kommen kann. Hierbei wurde berichtet, dass die Smack API nach einem veralteten Verfahren den Server anspricht. Ansonsten gibt es über dieses Verhalten keinerlei offizielle Dokumentation. Jedoch berichteten Kommilitonen über ähnliche Probleme und auch die Betreuer hatten auf diese Problematik keine konkrete Antwort. Bei dem XMPP Client bestand ebenfalls die Schwierigkeit, die empfangenen Daten zu parsen und die essentiellen Informationen zu extrahieren. Unser erster Ansatz war dieses Mittels einer speziellen JAXB Klasse zu realisieren. Aus Zeitgründen haben wir dann auf einen einfachen, in der JSL eingebauten XML Parser zurückgegriffen, um zumindest die passenden Daten in der GUI anzeigen zu können.

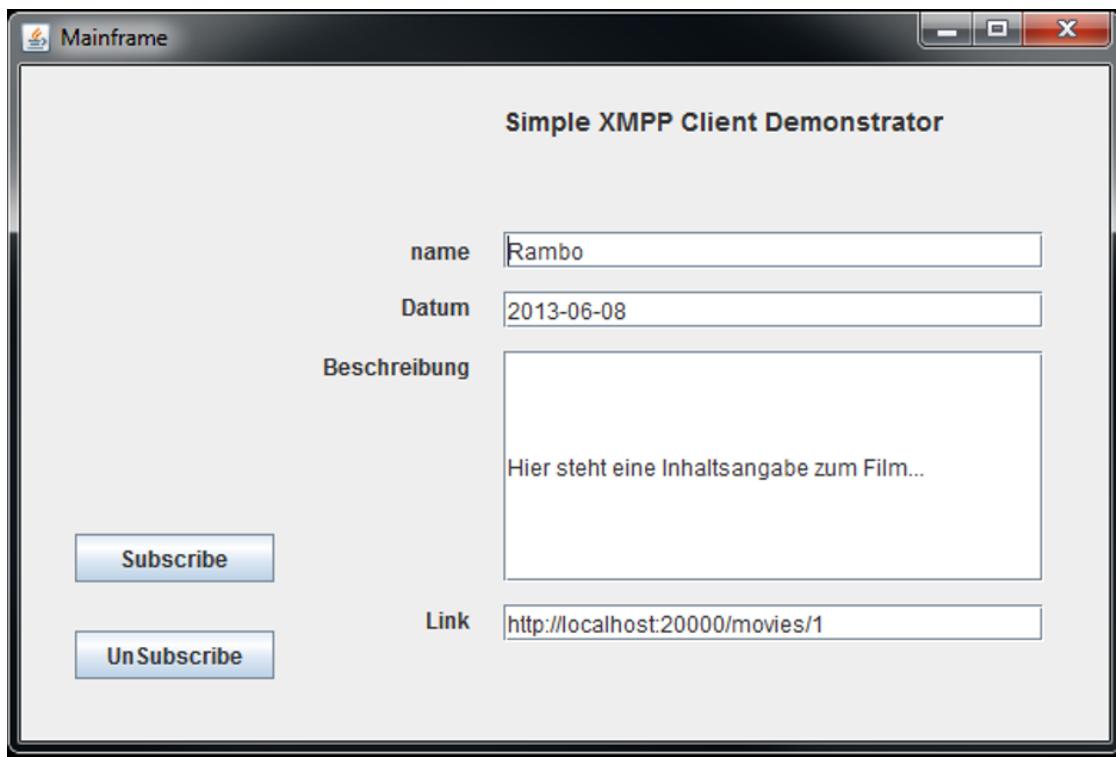


Abb: 5.1 XMPP Demo Client

6 Schluss

6.1 Allgemein

Generell hat sich uns bis zum heutigen Tage nicht erschlossen, warum wir für eine solche Anwendung zwingend Java verwenden müssen. Gerade im Bereich von Webanwendungen würden wir eher zu Ruby on Rails, node.js, Python oder auch PHP tendieren. Wir müssen auch zugeben, dass uns dieser Umstand etwas enttäuscht hat und die Motivation, sich in Java Klassen einzuarbeiten zu müssen, eine ganz andere war, als das z.B. mit node.js gewesen wäre.

6.2 Restserver

Es war sehr interessant zu sehen, wie eine REST Funktionalität mittels Java realisiert werden kann bzw. wie sich eine HTTP Funktionalität in Java einbauen lässt. Was wir an dieser Stelle aus zeitlichen Gründen nicht erreicht haben, ist die volle REST Funktionalität des Servers. Der Server lieferte zurzeit nur XML aus. Für eine volle REST Funktionalität sollte er zumindest noch die Repräsentationen HTML und JSON ausliefern können. Auch hatten wir angedacht, anstatt des Grizzly eher einen Tomcat Server für einige Funktionen zu verwenden. Aufgrund der Zeit fiel das jedoch komplett aus.

6.3 XML

XML war uns als Datenformat bereits hinreichend aus unserer ehemaligen Tätigkeit bekannt und ist darüber hinaus ein weitverbreitetes Datenaustauschformat in zahlreichen Anwendungen. Es stellte uns vor keine größeren Probleme, valide XML Dokumente und Schemata zu erstellen. Der einzige negative Punkt bei dem Datenaustauschformat XML ist der teils immense Overhead, der Formatbedingt produziert wird. Gerade im Bereich von Webanwendungen respektive Datenkommunikation tendieren wir daher generell dazu, ein kompakteres Datenformat wie JSON zu verwenden.

6.4 JAXB

Ist eine gute Erfindung und hat uns den Arbeitsprozess wesentlich erleichtert. Ein Verbesserungsvorschlag für die Entwickler des Softwarepakets wäre die Umsetzung von generierbaren Teilkomponenten eines XML-Schemas. Es wäre von großem Vorteil wenn untere Knotenelemente der Struktur als XML exportierbar wären und nicht jedes mal eine neue „Factory“ für jeden benötigten Teilexport erzeugt werden müsste.

6.5 XMPP

Eine grundlegende Kenntnis des XMPP Protokoll erachten wir in jeglicher Hinsicht als sinnvoll. Jedoch haben wir bedingt durch die fehlerhafte und äußerst schlecht dokumentierte API sehr viel Zeit mit dem Erarbeiten von Workarounds verschwendet. Diese Zeit hätten wir lieber dafür genutzt, eine breitere Kenntnis des XMPP Protokolls zu erlangen.

6.6 Java-Swing

Die Einarbeitung in das Entwicklungsverfahren von Java-Swing-Oberflächen beanspruchte mindestens zwei Wochen Arbeitszeit. Eine mangelhafte Anpassbarkeit auf individuell gestaltete Oberflächenentwürfe sowie die schwierige Fehleranalyse und -behebung kommen noch erschwerend hinzu. Einfacher wäre eine Umsetzung der XML Dokumente mittels XSLT zu XHTML gewesen.

7 Quellen

Gestaltung wissenschaftlicher Arbeiten
ISBN 978-3-8252-2774-6

Dissertation von Roy Fielding
<http://www.ics.uci.edu/%7Efielding/pubs/dissertation/top.htm>

XML Handbuch
ISBN 3-8272-9575-0

O'Reilly - XMPP: The definitive Guide
ISBN 978-0-596-52126-4

O'Reilly - REST in Practice
ISBN 978-0-596-80582-1

Galileo - Java ist auch eine Insel
ISBN 978-3-8362-1802-3

SMACK Api Doc
<http://www.igniterealtime.org/builds/smack/docs/latest/javadoc/>