



Fachhochschule Köln  
Cologne University of Applied Sciences

Fachhochschule Köln  
Fakultät für Informatik und Ingenieurwissenschaften

**WBA 2**  
Phase 2  
Dokumentation

Campus Gummersbach  
im Studiengang  
Medieninformatik

ausgearbeitet von:  
**RAMONA LIND**  
(Matrikelnummer: 11083935)  
**LISA LANG**  
(Matrikelnummer: 11084995)

Gummersbach, 23 Juni

---

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>2</b>
1.1 Aufgabe . . . . .	2
<b>2 Prozess</b>	<b>2</b>
2.1 Idee: . . . . .	2
2.2 Synchrone Datenübertragung . . . . .	2
2.3 Asynchrone Datenübertragung . . . . .	2
2.4 Szenario . . . . .	3
<b>3 Projektbezogenes XML Schema / Schemata</b>	<b>4</b>
3.1 Kommunikationsablauf . . . . .	4
3.2 Umsetzung . . . . .	5
<b>4 Ressourcen und die Semantik der HTTP-Operationen</b>	<b>9</b>
4.1 HTTP-Operationen . . . . .	9
4.2 Tabelle . . . . .	9
4.3 Anwendung in System . . . . .	10
<b>5 RESTful Webservice</b>	<b>11</b>
5.1 Marshalling/Unmarshalling . . . . .	11
5.2 HTTP Methoden und Pfade . . . . .	11
5.3 Server . . . . .	12
<b>6 Konzeption + XMPP Server einrichten</b>	<b>13</b>
6.1 XMPP Server . . . . .	13
6.2 Publish/Subscribe . . . . .	13
6.3 Leafs(Topics) . . . . .	13
<b>7 XMPP - Client</b>	<b>14</b>
<b>8 Client - Entwicklung</b>	<b>19</b>
<b>9 Rückblick</b>	<b>20</b>
9.1 Erfolge . . . . .	20
9.2 Probleme . . . . .	20
9.3 Verbesserungen . . . . .	21
<b>10 Quellenverzeichnis</b>	<b>22</b>
10.1 Literatur . . . . .	22
10.2 Internetquellen . . . . .	22

# 1 Einleitung

## 1.1 Aufgabe

Aufgabe war die Konzeption der synchronen Interaktion von Systemkomponenten unter Anwendung des Architekturstils REST und der asynchronen Interaktion von Systemkomponenten unter Anwendung des Standards XMPP (Phase 2).

# 2 Prozess

## 2.1 Idee:

Es soll eine Anwendung entstehen, in der Songs mit anderen Usern geteilt werden können. Neue Songs können mit Angabe des Titel, Interpret und des Genre hochladen werden. Optional ist die Angabe von Kommentaren und Bewertungen.

Durch die Suchfunktion können andere User bestimmten Interpreten, Songs oder Genre auswählen, um darüber dann Informationen zu beziehen bzw. einen Song anzuhören oder den Interpreten, das Genre oder auch einen bestimmten User zu abonnieren. Des weiteren besteht die Möglichkeit des Bewerten und Kommentierens eines Songs.

## 2.2 Synchrone Datenübertragung

Ein synchroner Vorgang in dieser Anwendung ist ein Vorgang in dem etwas hochgeladen wird und die Veränderung für alle User zu sehen ist. Diese Information wird direkt gegeben und nicht erst durch Abonnements abgerufen. Beispielsweise passiert dies in dem Projekt bei dem Hochladen von Songs und dem Verfassen von Kommentaren.

- Musik hochladen, kommentieren und bewerten
- Zuteilung in Genres, Künstler, Titel
- nach Genre, Interpret oder Musiktitel filtern

Diese Vorgänge werden alle synchron laufen. Dies bedeutet der User hat direkten Einfluss auf die Daten und kann diese einsehen und verändern.

## 2.3 Asynchrone Datenübertragung

Asynchrone Vorgänge sind Daten die nicht direkt übertragen werden. Diese werden erst bei Abfrage gesendet.

- ausgewählte Infos über Künstler, Genre oder User abonnieren
- aktuelle Infos, neueste Uploads und Kommentare beziehen

Ein User kann Daten anfordern, also Informationen abonnieren. Diese Vorgänge laufen dann asynchron.

## 2.4 Szenario

User 'Peter' logt sich bei der Anwendung 'MusicBoX' ein und landet auf seiner persönlichen Startseite. Dort sieht er die neuesten Informationen zu seinen abonnierten Kanälen. Er gibt unter der Suchfunktion den Interpreten 'XY' ein und abonniert diesen. Daraufhin lädt er einen neuen Song von seinem Lieblingsinterpreten 'YZ' hoch und versieht ihn mit einem Genre und einem Kommentar.

User 'Dennis' hat den Kanal 'YZ' abonniert und bekommt somit nach dem LogIn die Info, dass ein neuer Song hochgeladen wurde. Er hat nun die Möglichkeit den neuen Song anzuhören, die Kommentare und Bewertungen einzusehen, sowie selbst zu kommentieren und bewerten. Daraufhin besucht er die Seite des Users 'Peter' und fügt diesen seinen abonnierten Kanälen hinzu. Von diesem Zeitpunkt an sieht User 'Dennis' immer, wenn User 'Peter' etwas kommentiert oder hochlädt. User 'Petra' möchte die Neuigkeiten des Genre 'MN' erfahren und wählt dies in der Suchfunktion aus. Daraufhin sieht auch sie den neuen Song von 'YZ' den User 'Peter' hochgeladen hat.

### 3 Projektbezogenes XML Schema / Schemata

In diesem Schritt musste man sich überlegen, welche Unterpunkte wichtig für die Anwendung sind. Folgende Punkte wurden analysiert:

Kommentare

- Nutzer
- Datum
- Text

Musik

- Interpret
- Title
- Genre
- Erscheinungsdatum
- Timestamp
- Bewertung + Text

Nutzer Account

- Name
- Anmeldedatum
- Kommentare
- Timestamp

Zu diesen Daten wurden dann mehrere XML Dokumente erstellt. In jedem stehen spezifische Daten zu den verschiedenen Kategorien.

#### 3.1 Kommunikationsablauf

Kommunikationsabläufe zwischen dem User und dem Server sind folgende:

Der User kann Songs die auf dem Server in einer Datenbank abgelegt sind abrufen, kommentieren und bewerten. Des weiteren kann er die Songs nach Genre und Interpreten oder Titel filtern.

Auch kann der User Informationen abonnieren und bekommt Neuigkeiten asynchron mitgeteilt.

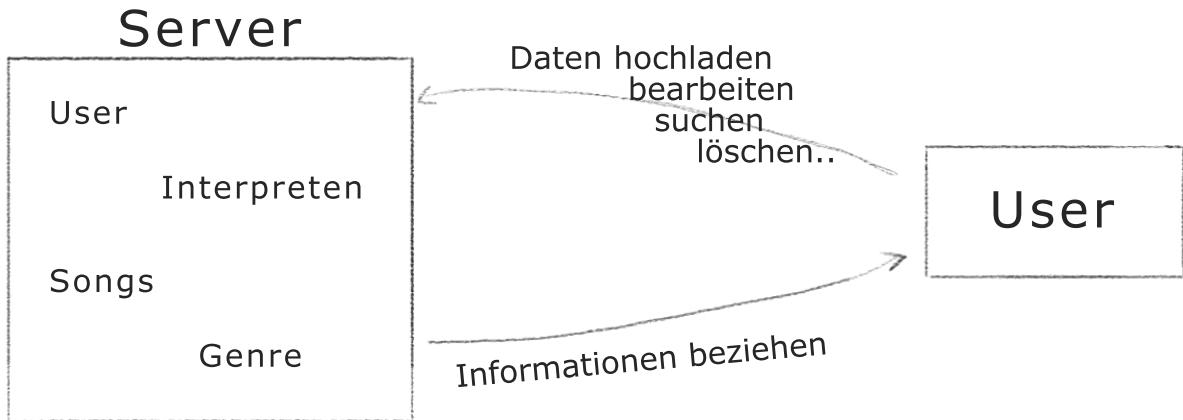


Abbildung 1: Kommunikationsablauf

## 3.2 Umsetzung

Es wurden mehrere XML Schemata zur Umsetzung Realisiert. Dazu wurde zu jedem Unterpunkt ein Schema erstellt.

In den XML Datensätzen werden die Daten gespeichert, die man zu den einzelnen Überthemen braucht. Es wurden folgende Überthemen analysiert.

User: In der Datei 'User' werden alle Daten über den angemeldeten User gespeichert. Dies sind unter anderem Name, Nickname, ID und ein Bild. Diese Daten könnten noch um einiges erweitert werden, wie beispielsweise durch das Datum der Anmeldung oder den derzeitigen Wohnort und vieles mehr. Aus Zeitgründen wurde sich allerdings für die oben genannten Daten entschieden.

```

1 <xs:element name="users">
2   <xs:complexType>
3     <xs:sequence>
4       <xs:element name="user" maxOccurs="unbounded" minOccurs="0">
5         <xs:complexType>
6           <xs:sequence>
7             <xs:element type="xs:string" name="nick" />
8             <xs:element type="xs:string" name="name" />
9             <xs:element type="xs:string" name="date" />
10            <xs:element type="xs:string" name="pic" />
11          </xs:sequence>
12          <xs:attribute type="xs:int" name="id" use="required" />
13        </xs:complexType>
14      </xs:element>
15    </xs:sequence>
16  </xs:complexType>
17</xs:element>
```

Code 1: Users

Interpreten: In dem File 'Interpreten' werden die Daten zu dem Interpreten gespeichert. Auch hier wurde sich für ein paar wenige Daten entschieden. Dies sind Name des Interpreten, Datum der ersten Veröffentlichung und das Genre, zu dem der Interpret zugeordnet wird. Zusätzlich wird noch eine Beschreibung gespeichert. Ebenfalls hat jeder Interpret eine eigene ID.

```

1 <xs:element name="interpreten">
2   <xs:complexType>
3     <xs:sequence>
4       <xs:element name="interpret" maxOccurs="unbounded" minOccurs="0">
5         <xs:complexType>
6           <xs:sequence>
7             <xs:element type="xs:string" name="name"/>
8             <xs:element type="xs:string" name="date"/>
9             <xs:element type="xs:string" name="pic"/>
10            <xs:element type="xs:string" name="genre"/>
11            <xs:element type="xs:string" name="description"/>
12          </xs:sequence>
13          <xs:attribute type="xs:int" name="id" use="required"/>
14        </xs:complexType>
15      </xs:element>
16    </xs:sequence>
17  </xs:complexType>
18</xs:element>
```

Code 2: Interpreten

Songs: In der 'Songs'-Datei werden die nötigen Daten zu Songs gespeichert. Es wird der Interpret des Songs gespeichert, der Titel, das zugehörige Genre, das Releasedatum, eine Beschreibung zu dem Song, und die Bewertung. Außerdem werden hier Restriktionen gespeichert, um unnötige und sinnlose Eingaben zu vermeiden. Es gibt Restriktionen zu dem Genre, dem Releasedate und der Bewertung.

```

1 <xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
2   xmlns:xs="http://www.w3.org/2001/XMLSchema">
3   <xs:element name="songs">
4     <xs:complexType>
5       <xs:sequence>
6         <xs:element name="song" maxOccurs="unbounded" minOccurs="0">
7           <xs:complexType>
8             <xs:sequence>
9               <xs:element type="xs:string" name="interpret"/>
10              <xs:element type="xs:string" name="title"/>
11              <xs:element type="xs:string" name="genre"/>
12              <xs:element type="xs:string" name="releasedate"/>
```

```

13         <xs:element type="xs:string" name="description" />
14         <xs:element type="xs:string" name="rate" />
15     </xs:sequence>
16     <xs:attribute type="xs:byte" name="id" use="required" />
17   </xs:complexType>
18 </xs:element>
19 </xs:sequence>
20 </xs:complexType>
21 </xs:element>
22 <xs:element name="genre">
23   <xs:simpleType>
24     <xs:restriction base="xs:string">
25       <xs:enumeration value="Rock" />
26       <xs:enumeration value="Pop" />
27       <xs:enumeration value="Hardrock" />
28       <xs:enumeration value="Clasik" />
29       <xs:enumeration value="Jazz" />
30       <xs:enumeration value="RnB" />
31       <xs:enumeration value="Reggae" />
32       <xs:enumeration value="Elecktro" />
33       <xs:enumeration value="Soul" />
34       <xs:enumeration value="Schlager" />
35       <xs:enumeration value="Punk" />
36     </xs:restriction>
37   </xs:simpleType>
38 </xs:element>
39 <xs:element name="releasedate">
40   <xs:simpleType>
41     <xs:restriction base="xs:positiveInteger">
42       <xs:maxInclusive value="2014" />
43       <xs:minInclusive value="1800" />
44     </xs:restriction>
45   </xs:simpleType>
46 </xs:element>
47 <xs:element name="rate">
48   <xs:simpleType>
49     <xs:restriction base="xs:positiveInteger">
50       <xs:maxInclusive value="0" />
51       <xs:minInclusive value="5" />
52     </xs:restriction>
53   </xs:simpleType>
54 </xs:element>
55 </xs:schema>

```

Code 3: Songs

Comments: In den Kommentaren wird der verfassende User gespeichert. Das Datum an dem der Kommentar verfasst wurde. Und der Kommentar an sich. Zudem bekommt jeder Kommentar eine ID.

```

1 <xs:element name="comments">
2   <xs:complexType>
3     <xs:sequence>
4       <xs:element name="comment" maxOccurs="unbounded" minOccurs="0">
5         <xs:complexType>
6           <xs:sequence>
7             <xs:element type="xs:string" name="user"/>
8             <xs:element type="xs:string" name="date"/>
9             <xs:element type="xs:string" name="text"/>
10            </xs:sequence>
11            <xs:attribute type="xs:byte" name="id" use="optional"/>
12          </xs:complexType>
13        </xs:element>
14      </xs:sequence>
15    </xs:complexType>
16  </xs:element>
```

Code 4: Comments

Zu diesen XML Daten wurden nun XSD-Daten mit je zwei Beispielsätzen generiert. Es wurde stets auf eine Valide, wohlgeformte Form geachtet.

Zuallererst wurden alle Daten in einer XML Datei gespeichert. Doch das Speichern in separate Dateien gestaltet die weitere Verarbeitung einfacher und so wurde diese Datei noch aufgeteilt.

User, Songs und Filter sind ComlexTypes und fassen die anderen Daten mit ein. Es muss immer der Username und der Titel des Songs gegeben sein. Unter anderem wird die ID als Attribut übergeben.

## 4 Ressourcen und die Semantik der HTTP-Operationen

Der nächste Schritt war nun die Umsetzung des synchronen Kommunikationsablaufs. Dies geschieht mithilfe von REST.

Bei RESTful Services geht es darum, eine Ressource über einen Web-Server verfügbar zu machen und eindeutig über eine URI zu identifizieren.

Zunächst mussten die einzelnen Ressourcen festgelegt werden. Dies wurde schon nahezu durch die Definition der Oberklassen und die Verarbeitung in den XML Datensätzen erledigt und sind somit 'User', 'Interpreten', 'Songs' und 'Comments'.

### 4.1 HTTP-Operationen

Daraufhin wurden dem zwei ausgewählten Ressourcen 'User' und 'Interpreten' die benötigten Methoden zugewiesen. Dafür stehen die HTTP-Operationen POST, zum Anlegen von neuen Informationen, sowie GET zum Auslesen, PUT zum Ändern und DELETE zum Löschen dieser Informationen bereit. Eine Ressource muss allerdings nicht jede der Operationen beinhalten.

Eine Ressource wird jeweils mit '/ressourcename' dargestellt und beinhaltet die Subressource, welche immer eine ID ist und mit '/ressourcename/id' angesprochen wird.

### 4.2 Tabelle

Ressource	URI	Methode
Liste aller User	/user	GET, POST
Liste einzelner User	/user{id}	GET, PUT, POST, DELETE
Liste aller Songs	/song	GET, POST
Liste einzelner Songs	/song{id}	GET, PUT, POST, DELETE
Liste aller Interpreten	/interpret	GET, POST
Liste einzelner Interpreten	/interpret{id}	GET, PUT, POST, DELETE
Liste aller Kommentare	/comments	GET, POST
Liste einzelner Kommentare	/comments{id}	GET, PUT, POST, DELETE
Liste aller Songs in einem Genre	/genre{id}/ song	GET
Liste aller Songs eines Interpreten	/interpret{id}/ song	GET
Liste aller Songs eines Users	/user{id}/ song	GET

## 4.3 Anwendung in System

Am Beispiel der Ressource 'User' wäre somit GET '/user' dafür zuständig eine Liste aller User auszugeben und mit POST könnte ein neuer User hergestellt werden. Mit einem GET bei '/user/123' werden alle Details des Users mit der ID '123' aufgelistet. Dies bedeutet man kann alle gespeicherten Daten, wie beispielsweise Name, Eintrittsdatum und Nickname, des Users '123' einsehen.

Mit POST und PUT können dann neue Informationen hinzugefügt bzw. Daten geändert werden und mit DELETE kann sich dieser User selbst löschen.

Zu den weiteren Ressourcen passiert dies analog.

In der späteren Umsetzung wurde die Kommentar- Funktion allerdings vernachlässigt und die Genre-Einteilung wurde nicht so stark ausgearbeitet, da man sich mehr auf die Ressourcen 'User' und 'Interpreten' spezialisierte.

Siehe Abbildung 2

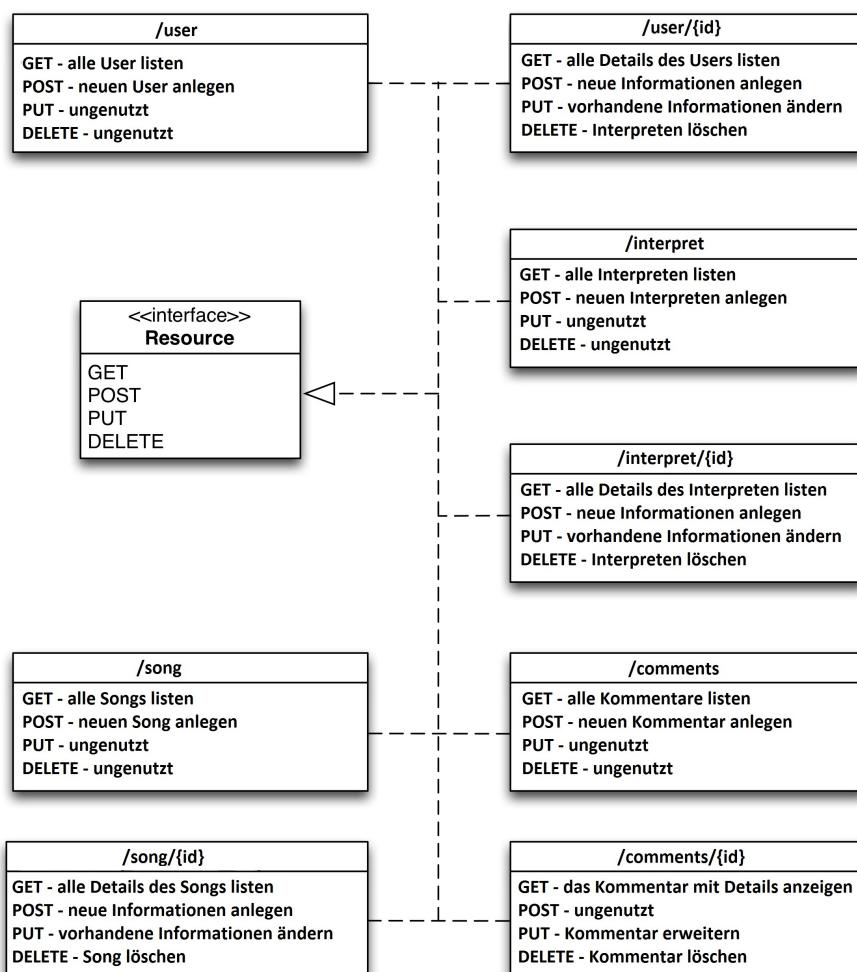


Abbildung 2: HTTP-Methoden

## 5 RESTful Webservice

Bei dem dritten Meilenstein wurden die Daten von User und Interpreten als Ressourcen verfasst.

### 5.1 Marshalling/Unmarshalling

Die Methoden des Marshalling / Unmarshalling wurde schon in Phase 1 gebraucht und dient dazu, Daten in die XML Files zu speichern oder wieder zu verwenden.

Die Marshalling / Unmarshalling-Methoden findet man in XmlHelper.java. Dort wird zu Comments, Interpreten, Users und Songs eine Marshall/ Unmarshall-Methode geschrieben. Auf diese wird dann zugegriffen.

### 5.2 HTTP Methoden und Pfade

Danach wurden die Operatoren GET, DELETE, PUT und POST erstellt und bearbeitet. Dies wurde für Interpreten und User erledigt.

USER

Zuerst wird der Basispfad festgesetzt. Dieser ist 'users'.

```
@Path("/users")
```

Dann kommt die erste HTTP Methode, ein @GET. Das GET ist wie oben schon erwähnt zum Abrufen der Daten.

@Produces Setzt deutlich einen MIME-Typ. Es kann grundsätzlich auch weggelassen, wird aber bevorzugt gesetzt. Es setzt den Typ XML, aber es gibt aber auch noch weitere MIME-Typen.

@Get und @Producesapplication/xml werden gesetzt, um es als Methode zu kennzeichnen. Die Ressource wird per HTTP-GET verfügbar gemacht und Daten im MIME-Typ 'application/xml' gespeichert und am Ende wird die Methode schließlich zurückgegeben.

```
@GET
```

```
@Produces("application/xml")
```

```
public Users getUsers() throws JAXBException, SAXException  
return this.xmlHelp.unmarshalUsers();
```

ID

Die Annotation @Path(id) sorgt für einen Aufruf an die URI '/user/id', die an die Methode 'getUser' weitergeleitet wird. Die Id ist ein Ausdruck der Expression Language (EL). Mit der Parameter-Annotaiton @QueryParam werden die Werte über die QueryParameter des Aufrufs der Form /users?name angezeigt.

Die Queryparams werden durch eine for-each-Schleife geleitet, um die einzelnen Daten abzufragen. Ohne explizite Angabe weiterer Einschränzungskriterien werden alle User er-

fragt. Falls die URL länger wird, kommen weiterer Einschränkungen hinzu.

@PUT

@Consumes gibt den MIME-Typ der gesendeten Daten an. @PUT und @Consumes sind nötig. Da die Funktion void ist, muss kein @Produces gesetzt sein.

@POST

Die Post Funktion ist dafür da, neue Daten anzulegen.

@DELETE

Die Delete Funktion ist dafür da, Daten wieder zu löschen.

Dieser Vorgang wurde dann auch für Interpreten angepasst.

Grundsätzlich gibt es keine feste Vorgabe, wann man QueryParam oder PathParams benutzt. PathParams werden verwendet, wenn Abrufe auf IDs basieren. QueryParam werden dagegen für Filter verwendet oder wenn keine feste Liste von Optionen bereitsteht. Da es die Aufgabe war beide zu benutzen, wurde sich entschieden in User eine QueryParam-Abfrage zu machen, um dort nicht nur einen ID Abruf zu haben.

## 5.3 Server

In der Klasse AppServer wird der Grizzly Server erstellt. Es wird eine Standard-Hostadresse angegeben. Der Server läuft 10 Minuten bevor er wieder gestoppt wird.

```

1 public class AppServer
2 {
3     public static void main( String [] args ) throws Exception
4     {
5         String url = "http://localhost:4534";
6
7
8         SelectorThread srv = GrizzlyServerFactory.create( url );
9
10        System.out.println( "URL: " + url );
11        Thread.sleep( 1000 * 60 * 10 );
12        srv.stopEndpoint();
13    }
14}
```

Code 5: AppServer

In der Klasse AppClient wird ein Client nachgestellt, der auf die Methoden zugreift und testet das Bearbeiten und Benutzen der Daten.

## 6 Konzeption + XMPP Server einrichten

Nun soll die asynchrone Verarbeitung mithilfe des XMPP Servers ermöglicht werden.

### 6.1 XMPP Server

Um nun die asynchrone Übertragung zu ermöglichen, ist der XMPP Server notwendig. Dieser ist ein Internetstandard, der es ermöglicht XML Daten in Echtzeit zu übertragen. Darüber hinaus wird die Smack(x)-API benötigt, um den Zugriff auf den XMPP Server zu erhalten.

### 6.2 Publish/Subscribe

Es soll möglich sein Publish/Subscribe zu verwenden. Ein Publisher hat die Möglichkeit Nachrichten zu einem bestimmten Thema/Topic zu veröffentlichen. Der Subscriber kann diese Daten beziehen, indem er das Thema/Topic abonniert hat. Zwischen diesen beiden Komponenten liegt ein Vermittler(Message Broker/Event Dispatcher), welcher die entsprechenden Topics an den Publisher und Subscriber weiterleitet.

Siehe Abbildung 3

### 6.3 Leafs(Topics)

Die Themen/Topics, die man abonnieren bzw. dort etwas veröffentlichen kann, nennt man Leafs. Diese werden als LeafNodes repräsentiert, die eindeutig einem Topic zugeordnet sind und nur Nachrichten beinhalten. Des Weiteren gibt es CollectionNodes, welche mehrere Nodes beinhalten und somit erlauben, dass man eine Kollektion von LeafNodes mit einem übergeordneten Node abonnieren kann.

In der Anwendung 'MusicBoX' kann jeder angemeldete User Publisher und Subscriber sein. Dies geschieht beispielsweise durch das Abonnieren eines Interpreten-Kanals, was den User zu einem Subscriber macht, oder das Ergänzen von Daten zu diesem Interpreten, was ihn zu einem Publisher macht.

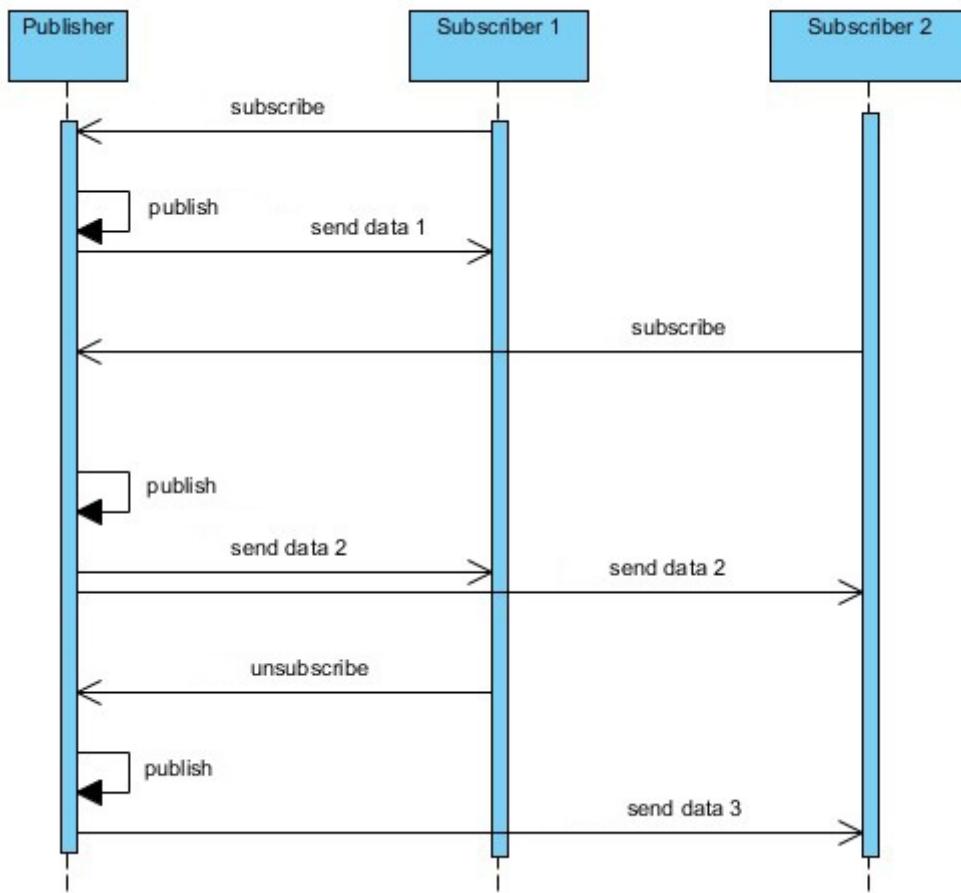


Abbildung 3: Publish und Subscribe Ablauf

## 7 XMPP - Client

In Xmpp.java wird ein Xmpp Client aufgebaut.

Zuerst wird eine Verbindung zu dem Server hergestellt.

Danach wird der User angemeldet. Es müssen der Username und das Passwort angegeben werden. Dann kann der Login erfolgreich abgeschlossen werden. Danach wird ein PubSub-Manager mit einer existierenden Verbindung erstellt.

Es wurden die Funktionen nachträglich mehrere Male geändert, um diese in die GUI einarbeiten zu können.

```

1  public void makeConn() {
2
3      System.out.println(server);
4      ConnectionConfiguration config = new ConnectionConfiguration(server,
5          port);
6      // pass some connection options
7      config.setSASLAuthenticationEnabled(true);
8      SASLAuthentication.supportSASLMechanism("PLAIN", 0);
  
```

```

9     con = new XMPPConnection(config);
10    }
11
12    public boolean doConnect() {
13        // let's connect
14        try {
15            con.connect();
16            con.login(user, passw);
17            // Create a pubsub manager using an existing Connection
18            psm = new PubSubManager(con, "pubsub." + con.getHost());
19            System.out.println("LoggedIn");
20            return true;
21        } catch (XMPPException e) {
22            System.err.println("Login failed!");
23            return false;
24        }
25    }

```

Um die richtigen Informationen zu bekommen ist es notwendig, die Informationen zu den verbunden Items / Knoten innerhalb des Smack Client zu konfigurieren. Dazu muss ein NodeInformationProvider erstellt und registriert werden. Das geschieht mit dem ServiceDiscoveryManager. Um diesen Manager zu erstellen muss getInstanceFor(con) zu der Klasse ServiceDiscoveryManager gesendet werden. Wenn der Manager erstellt wurde ist es möglich Informationen in den Nodes zu speichern.

```

1  public void discoverServices() throws XMPPException {
2      System.out.println("Discovering Services... ");
3      sdm = ServiceDiscoveryManager.getInstanceFor(con);
4      DiscoverItems items = sdm.discoverItems(XmppData.host);
5      Iterator<DiscoverItems.Item> iter = items.getItems();
6      while (iter.hasNext()) {
7          DiscoverItems.Item i = iter.next();
8          System.out.println(i.toXML());
9      }

```

In der Nächsten Funktion wird erst mal wie zuvor schon eine Verbindung zu dem ServiceDiscoveryManager aufgebaut und die Items auf Null gesetzt. Dann werden die gegebenen Items abgerufen. Die entdeckten Items werden geholt und dann angezeigt.

```

1  public void discoverNodes() {
2      System.out.println("Discovering Nodes... ");
3      sdm = ServiceDiscoveryManager.getInstanceFor(con);
4      DiscoverItems items = null;
5      try {

```

```

6     items = sdm.discoverItems("pubsub." + XmppData.host);
7 } catch (XMPPException e) {
8     e.printStackTrace();
9     System.out.println("Discover failed");
10 }
11 Iterator<DiscoverItems.Item> iter = items.getItems();
12 while (iter.hasNext()) {
13     DiscoverItems.Item i = iter.next();
14     System.out.println(i.getEntityID());
15     System.out.println(i.getNode());
16     System.out.println(i.getName());
17 }
18 }
```

Diese Funktion gibt am ende eine Array Liste der Nodes aus.

```

1 public List<String> getNodes() {
2     // Create a manager using an existing Connection
3     ServiceDiscoveryManager mgr = ServiceDiscoveryManager
4         .getInstanceFor(con);
5     DiscoverItems items = null;
6     try {
7         items = mgr.discoverItems("pubsub." + XmppData.host);
8     } catch (XMPPException e) {
9         e.printStackTrace();
10        System.out.println("Discover failed");
11    }
12    List<String> nodeList = new ArrayList<String>();
13    Iterator<DiscoverItems.Item> iter = items.getItems();
14    while (iter.hasNext()) {
15        DiscoverItems.Item i = iter.next();
16        nodeList.add(i.getNode());
17    }
18    return nodeList;
19 }
```

Hier wird ein Node erstellt.

```

1 public void createNode(String nodeID) {
2     // Knoten erstellen und konfigurieren
3     LeafNode leaf = getNode(nodeID);
4     ConfigureForm form = new ConfigureForm(FormType.submit);
5     form.setAccessModel(AccessModel.open);
6     form.setDeliverPayloads(true);
7     form.setNotifyRetract(true);
```

```

8     form.setPersistentItems(true);
9     form.setPublishModel(PublishModel.open);
10    try {
11        leaf.sendConfigurationForm(form);
12    } catch (XMPPException e) {
13        e.printStackTrace();
14        System.out.println("sendConfigurationForm failed");
15    }
16 }
```

Ein Item in einem Node wird “gepublisht”. Dazu muss die Node geholt und ein Payload erstellt werden. In den Payload wird dann das neue Node geladen.

```

1  public void pubItemInNode(String nodeID, String payload) {
2      LeafNode node = getNode(nodeID);
3      SimplePayload simplePayload = new SimplePayload(nodeID, "pubsub:" +
4          nodeID, payload);
5      PayloadItem<SimplePayload> item = new PayloadItem<SimplePayload>(" +
6          "expiration" + System.currentTimeMillis(),
7          simplePayload);
8      node.publish(item);
9      try {
10         System.out.println();
11     } catch (XMPPException e) {
12         e.printStackTrace();
13         System.out.println("Fehler beim publishen");
14     }
15 }
```

Es wird in der nächsten Funktion ein Node abonniert.

```

1  public void subToNode(String nodeID, ItemEventListener<Item> listener) {
2      // Knoten "besorgen" und abonnieren
3      LeafNode node = getNode(nodeID);
4      node.addItemEventListener(listener);
5      try {
6          node.subscribe(user+ "@" +XmppData.host);
7      } catch (XMPPException e) {
8          e.printStackTrace();
9          System.out.println("subToNode failed");
10     }
11 }
```

Dann wird der Node aufgerufen.

```
1 public LeafNode getNode( String nodeID ) {
2     try {
3         return psm.getNode( nodeID );
4     } catch ( XMPPEException e ) {
5         e.printStackTrace();
6         System.out.println( "Knoten: " + nodeID + " konnte nicht geladen werden"
7             );
8         return null;
9     }
}
```

Und am Ende wird die Verbindung wieder beendet.

In der XmppData.java sind die Konstanten zu dem Server, um diese schneller und einfacher aufzurufen.

```
1 public void disconnect() {
2     con.disconnect();
3 }
```

## 8 Client - Entwicklung

Die Entwicklung des grafischen User Interfaces wurde in Java Swing durchgeführt. Sie soll eine grobe Idee davon geben, wie die Anwendung 'MusicBoX' aussehen könnte.

Zunächst muss sich der registrierte User mit seinem Username und seinem Passwort einloggen. Des weiteren muss der Server und der Port eingegeben sein, um eine Verbindung herstellen zu können. Die verschiedenen Elemente auf der Startseite kann der User bereits schon vor dem LogIn sehen, doch sind diese erst nach erfolgreicher Anmeldung aktiv.

Dies wurde mit einem ActionListener realisiert. Dieser fängt bestimmte Fehler ab und sorgt dann für eine Verbindung, die er aus der vorher geschriebenen XMPP Datei holt. Diese musste mehrere Male auf das Abrufen angepasst werden.

Nach dem LogIn werden die verschiedenen Elemente eingeschaltet.

Die neuesten Informationen seiner abonnierten Kanäle werden ihm in der Box 'News der abonnierten Kanäle' nun durch asynchrone Verarbeitung angezeigt. Des weiteren sieht er auf den ersten Blick vier Boxen mit, die gefüllt werden mit seinen abonnierten Usern, Interpreten, Songs und Genres.

Ebenfalls hat er die Möglichkeit nach diesen Themen zu suchen und neue Kanäle zu abonnieren. Dies funktioniert über die Suchfunktion, bei der zuallererst die Ressource ausgewählt wird und dann durch das Eingeben der gewünschte User, Interpret, Song oder Genre gefunden werden kann. Ist der gewünschte Kanal gefunden, kann dieser durch einen Klick auf 'Subscribe' abonniert werden. Falls der gesuchte Kanal bereits abonniert ist, kann wird der 'Subscribe'-Button durch einen 'Unsubscribe'-Button ersetzt, um jederzeit das Abonnement zu beenden.

Möchte der User beispielsweise einen neuen Song hochladen, erfolgt dies durch die HTTP-Methode POST, die ausgewählt wurde, nachdem der User die richtige Ressource gewählt und den Namen des neuen Songs eingegeben hat.

(Siehe Abbildung) 4

Aufgrund von Zeitknappheit wurde die GUI nur grafisch erstellt. Außer der Anmeldefunktion sind keine weiteren Buttons funktional. Damit ein erfolgreicher LogIn überhaupt zustande kommt, ist zuerst die Verbindung mit dem Openfire-Server herzustellen.

REST und XMPP müssen also noch in die GUI implementiert werden, um die synchrone und asynchrone Kommunikation zu ermöglichen. Dieser Schritt konnte bei uns leider nicht mehr ausgeführt werden,

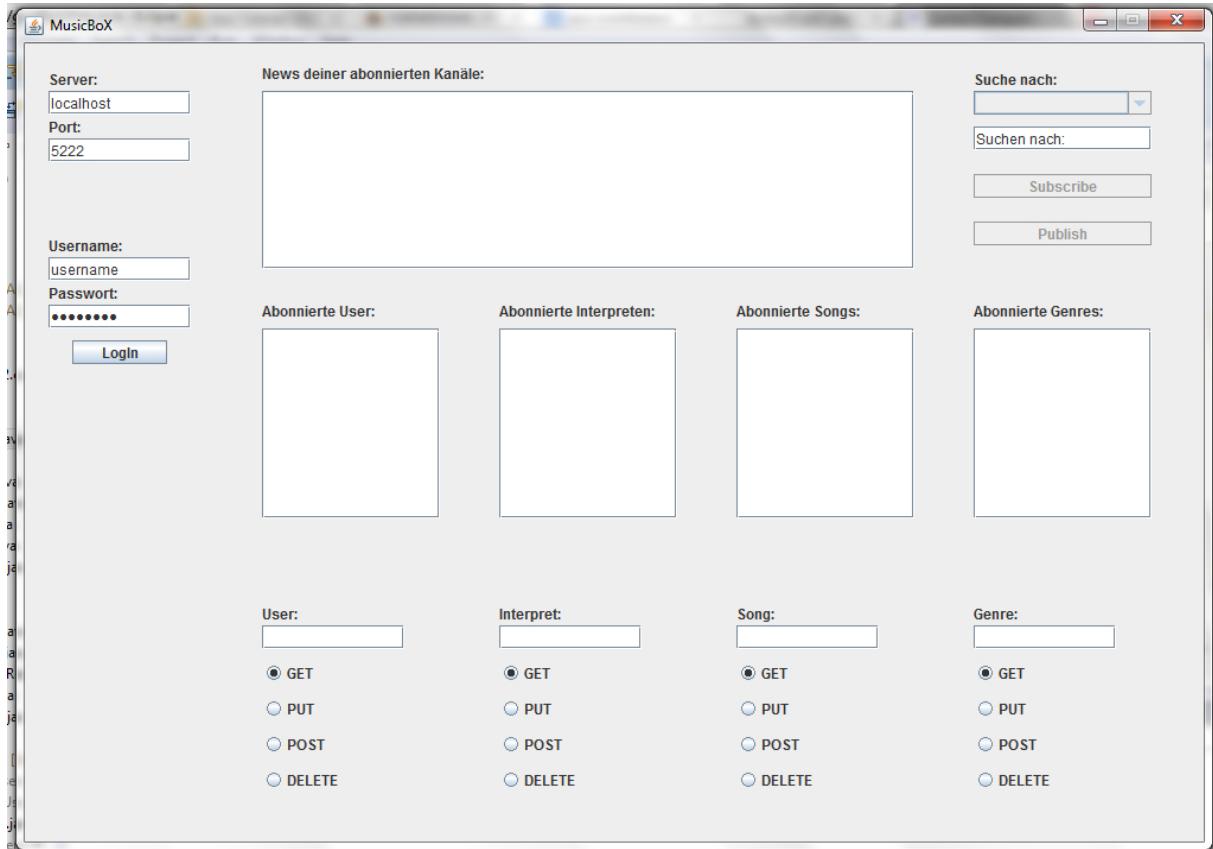


Abbildung 4: Screenshot der GUI

## 9 Rückblick

### 9.1 Erfolge

Es war eine sehr intensive und umfangreiche Arbeit, die aber insgesamt das Wissen über REST und Serveranwendungen erweitern konnte. Man brauchte ein gutes Zeitmanagement und musste sich viel selbst erarbeiten. Was das Wissen auch um einiges erweitert hat.

### 9.2 Probleme

Da das Vorwissen zu diesen Themen nicht besonders ausgeprägt war, musste man sich zunächst einen Überblick verschaffen und viele Dinge recherchieren, was reichlich Zeit in Anspruch nahm. Einige Dinge sind erst bei späteren Problemen oder weiterer Bearbeitung klar geworden, woraufhin man sich immer wieder mit Fehlerkorrektur aufhielt. Man bekam viel Hilfe von Kommilitonen oder auch aus dem Internet, doch widersprachen sich diese Dinge auch teilweise und es brauchte viel Zeit um einen Durchblick zu gewinnen und dieses Wissen in der Anwendung zu gebrauchen. Das eigentliche Programmieren wurde dann stark verlangsamt, durch das Fehlerfinden in dem sehr schnell komplex und verschachtelt werdenden Code.

Am Anfang konnte man die Arbeit durch fehlende Erfahrung schlecht einschätzen und hat sich ein einigen Stellen zu viel vorgenommen. Als langsam der Durchblick größer wurde und sich mit allem lange genug beschäftigt wurde, wurde allerdings die Zeit knapp, so dass nicht mehr alles, was verlangt war, erledigt werden konnte wurde. Das User Interface wurde erstellt, doch leider fehlt die vollständige Implementierung von REST und XMPP. Soweit es möglich war wurde alles noch getestet.

### 9.3 Verbesserungen

Schon im Laufe der Arbeit kamen immer neue Ideen zur Verbesserung auf. Zum einen hätte man von Anfang die Ressourcen stärker ausarbeiten können. Zum anderen weitere Ressourcen einbinden. Für die Anwendung wäre ein Admin mit speziellen Rechten nützlich gewesen, der als einziger beispielsweise die Möglichkeit hat User zu löschen, denn so wie die Anwendung im Moment aufgebaut ist, hätte jeder User die Möglichkeit einen anderen User zu löschen. Desweiteren müsste der XMPP Server und das User Interface noch verbessert werden. Und am wichtigsten wäre wohl die endgültige Einbindung von REST und XMPP in die GUI.

---

## 10 Quellenverzeichnis

### 10.1 Literatur

[REST und HTTP] Stefan Tilkov, REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien: Dpunkt Verlag; Auflage: 2. akt. und erw. Auflage (30. Juni 2011)

### 10.2 Internetquellen

- |                         |   |
|-------------------------|---|
| [REST]                  | Einarbeitung in REST<br><a href="http://openbook.galileocomputing.de/java7/1507_13_002.html">http://openbook.galileocomputing.de/java7/1507_13_002.html</a>   |
| [Parameter]             | Erklärung der Parameter.<br><a href="http://openbook.galileocomputing.de/java7/1507_13_002.html">http://openbook.galileocomputing.de/java7/1507_13_002.html</a><br><a href="http://www.mkyong.com/webservices/jax-rs/jax-rs-pathparam-example">http://www.mkyong.com/webservices/jax-rs/jax-rs-pathparam-example</a><br><a href="http://www.mkyong.com/webservices/jax-rs/jax-rs-queryparam-example">http://www.mkyong.com/webservices/jax-rs/jax-rs-queryparam-example</a> |
| [Publish Subscribe]     | Publish Subscribe Erklärung.<br><a href="http://www.roboticswikibook.org/conf/display/patt/Publish-Subscribe">http://www.roboticswikibook.org/conf/display/patt/Publish-Subscribe</a>   |
| [Gui Tutorial]          | Tutorial wie man eine GUI erstellt.<br><a href="http://www.java-tutorial.org/layout-manager.html">http://www.java-tutorial.org/layout-manager.html</a>  |
| [Medieninformatik Wiki] | Alle Informationen und Links die geben waren.<br><a href="http://www.medieninformatik.fh-koeln.de/w/index.php/WBA2_SoSe13:Phase2">http://www.medieninformatik.fh-koeln.de/w/index.php/WBA2_SoSe13:Phase2</a>  |

