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

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. Es können neue Songs mit Angabe des Titel, Interpret und des Genre hochladen werden. Durch den Filter nach Interpret, Musiktitel und Genre können andere User nach bestimmten Songs suchen, um diese anzuhören oder den Künstler, 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 ist ein Vorgang in dem etwas Hochgeladen wird und die Veränderung für alle auf der Startseite zu sehen ist. Diese Information wird direkt gegeben und nicht erst durch Abbos abgerufen. Beispielsweise passiert dies in dem Projekt bei dem Hochladen von Videos 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. Das 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 Daten Anfordern, also Informationen Abonnieren. Diese Vorgänge laufen dann Asynchron.

2.4 Szenario

Ein User "Peter"lädt ein neues Lied von seinem Lieblingskünstler "XY"hoch und versieht es mit einem Kommentar. User "Dennis"den Kanal des Künstlers "XYäbonniert und bekommt so die Infos zu dem hoch geladenen Song und den zugehörigen Kommentaren und Bewertungen. Peter hat das Lied einem passendem Genre zugeordnet. User "Petra"findet durch den Genre Filter den neuen Song.

3 Projektbezogenes XML Schema / Schemata

Folgende Unterpunkte müssen für die Musikseite gegeben sein:

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.Und er kann die Songs nach Genre und Interpreten oder Titel filtern.
Auch kann der User Informationen Abonnieren und bekommt Neuigkeiten Asynchron mitgeteilt.

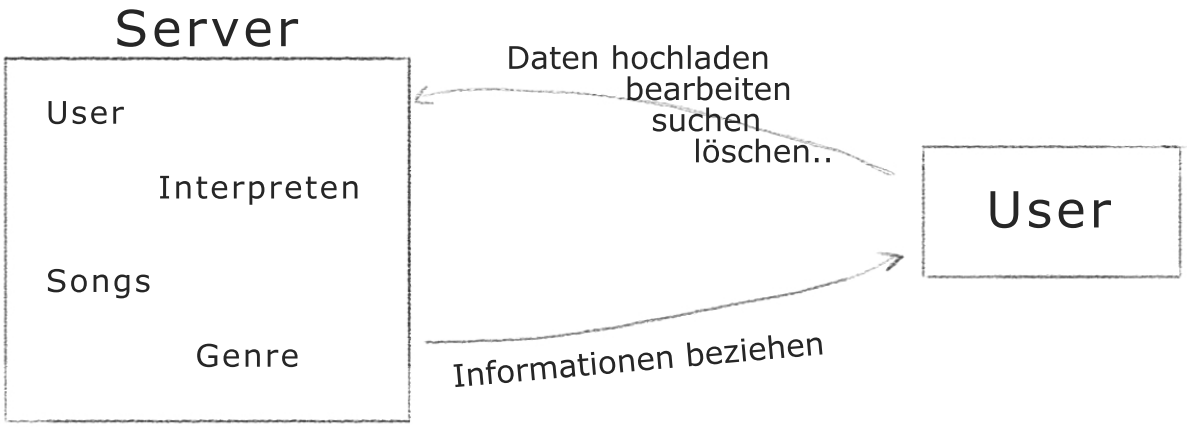

Abbildung 1: Kommunikationsablauf

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 dem 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. Man könnte auch das Datum der Anmeldung speichern oder Wohnort und vieles mehr. Aus Zeitgründen wurde sich für die oben genannten Daten entschieden.

```
<xs:element name="users">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="user" maxOccurs="unbounded" minOccurs="0">
  <xs:complexType>
   <xs:sequence>
    <xs:element type="xs:string" name="nick" />
    <xs:element type="xs:string" name="name" />
    <xs:element type="xs:string" name="date" />
    <xs:element type="xs:string" name="pic" />
    </xs:sequence>
   <xs:attribute type="xs:int" name="id" use="required" />
  </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</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, Geburtsdatum , Genre zu dem der Interpret zu geordnet wird. Es wird noch eine Beschreibung gespeichert. Auch jeder Interpret hat eine ID.

```
<xs:element name="interpreten">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="interpret" maxOccurs="unbounded" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element type="xs:string" name="name"/>
            <xs:element type="xs:string" name="date"/>
            <xs:element type="xs:string" name="pic"/>
            <xs:element type="xs:string" name="genre"/>
            <xs:element type="xs:string" name="description"/>
          </xs:sequence>
          <xs:attribute type="xs:int" name="id" use="required"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Code 2: Interpreten

Songs: In der Songsdatei 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.

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="
    qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="songs">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="song" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element type="xs:string" name="interpret"/>
              <xs:element type="xs:string" name="title"/>
              <xs:element type="xs:string" name="genre"/>
              <xs:element type="xs:string" name="releasedate"/>
              <xs:element type="xs:string" name="description"/>
```

```
                <xs:element type="xs:string" name="rate"/>
              </xs:sequence>
              <xs:attribute type="xs:byte" name="id" use="required"/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="genre">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="Rock"></xs:enumeration>
          <xs:enumeration value="Pop"></xs:enumeration>
          <xs:enumeration value="Hardrock"></xs:enumeration>
          <xs:enumeration value="Clasik"></xs:enumeration>
          <xs:enumeration value="Jazz"></xs:enumeration>
          <xs:enumeration value="RnB"></xs:enumeration>
          <xs:enumeration value="Reggae"></xs:enumeration>
          <xs:enumeration value="Elecktro"></xs:enumeration>
          <xs:enumeration value="Soul"></xs:enumeration>
          <xs:enumeration value="Schlager"></xs:enumeration>
          <xs:enumeration value="Punk"></xs:enumeration>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
    <xs:element name="releasedate">
      <xs:simpleType>
        <xs:restriction base="xs:positiveInterger">
          <xs:maxInclusive value="2014"/>
          <xs:minInclusive value="1800"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
    <xs:element name="rate">
      <xs:simpleType>
        <xs:restriction base="xs:positiveInterger">
          <xs:maxInclusive value="0"/>
          <xs:minInclusive value="5"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
</xs:schema>
```

Code 3: Songs

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

Kommentar eine ID.

```
<xs:element name="comments">
   <xs:complexType>
    <xs:sequence>
      <xs:element name="comment" maxOccurs="unbounded" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element type="xs:string" name="user"/>
            <xs:element type="xs:string" name="date"/>
            <xs:element type="xs:string" name="text"/>
          </xs:sequence>
          <xs:attribute type="xs:byte" name="id" use="optional"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Code 4: Comments

Zu diesen XML Daten wurden noch XSD Daten mit je zwei Beispielsetzen generiert. Es wurde setzt auf eine Valide, Wohlgeformte Form geachtet.
Zu erst 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”.
Daraufhin wurden unseren 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.

4.1 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

4.2 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 Informaionen 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.

5 RESTful Webservice

Bei dem dritten Meilenstein wurden zu den Daten von User und Interpreten die 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 Anwendung Marshalling / Unmarschalling Methoden findet man in XmlHelper.java. Dort wird zu Comments, Interpreten, Users, Songs eine Marshal/ Unmarshall Methode geschrieben. Auf diese wird dann zugegriffen.

5.2 HTTP Methoden und Pfade

Dannach wurden die Operatoren Get, Delete und Post erstellt und bearbeitet. Dies wurde für Interpreten und User gemacht.
USER
Als erstes wird über 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 MINE-Typ. Es kann grundsätzlich auch weggelassen werden, ist aber besser es zu setzten. Es setzt den Typ XML, es gibt aber auch noch weitere MINE-Typen. @Get und @Producesapplication/xml um als Methode zu Kennzeichnen, die Ressource per HTTP-Get verfügbar macht Daten im Mime-Typ application/xml, am ende wir die Methode zurückgegeben. @GET
@Produces(äpplication/xml")
public Users getUsers() throws JAXBException, SAXException
return this.xmlHelp.unmarshalUsers();

ID
Die Annotation @Path(id) sorgt für Aufruf an die Uri /user/id die an die Methode getUser weitergeleitet wird.Id ist ein ausdruck der Expression Language (EL). Mit der Parameter-Annotaion @QueryParam um die Werte über die QueryParameter des Aufrust der Form /users?name
Die Queryparams werden durch eine for each schleife geleitet um die einzelnen Daten abzufragen. Ohne explizite Angabe weiterer Einschränkungskriterien werden alle User erfragt, wenn die URL länger wird, kommen weiterer Einschränkungen hinzu.
@PUT

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

In der Klasse AppServer wird der Grizzly Server erstellt. Es wird eine Standart 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 Put Methode zugreift und Testet das Holen der Daten.

6 Konzeption + XMPP Server einrichten

Es gibt zwei Arten von Nodes einmal Leaf Nodes, welche Nachrichten beihalten und Colection Nodes welche Nodes beihalten. Die Aufgabe war es sich mit Leaf Nodes zu beschäftigen. In dem Projekt wurden Songs und Genres als Leafs definiert. Publishen kann jeder User, wie auch Subsciben.
Ein Publischer sendet seine Daten. Subscriber die den User abonniert haben empfangen die Daten Asynchron.
Daten die Abonniert wurden, also Daten der Songs, Kommentare, Bewertungen und die Inhalte des Genres, werden übertragen.

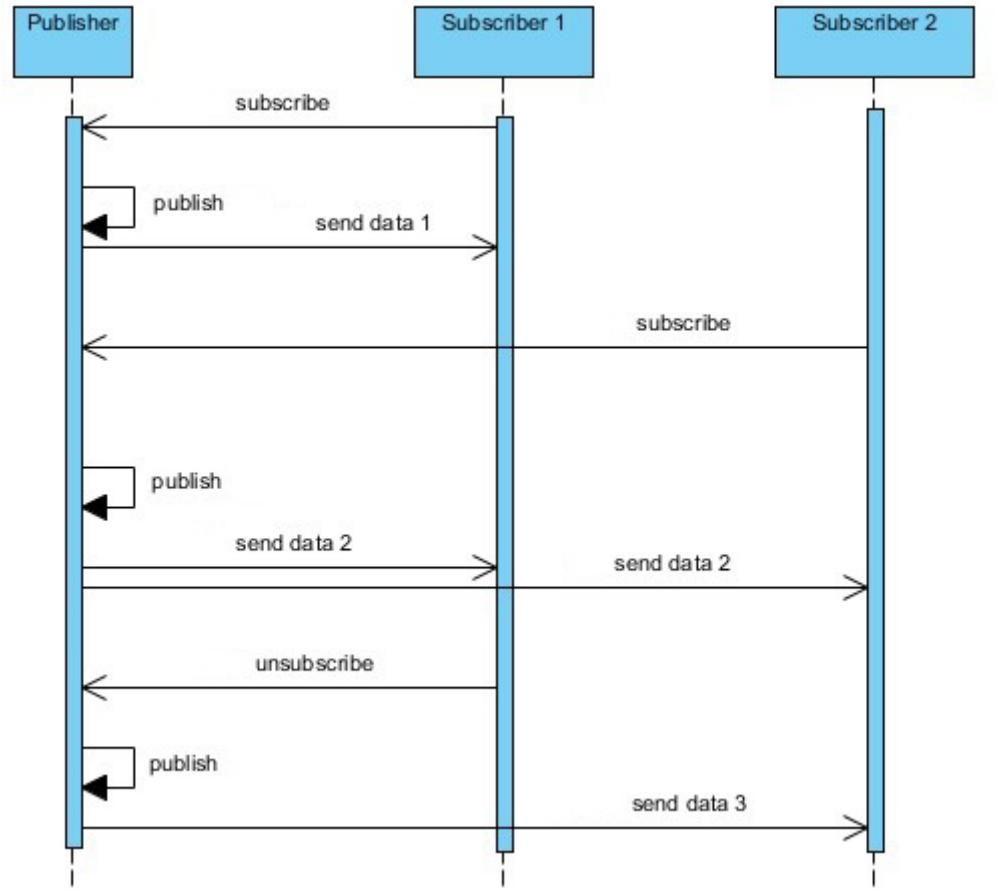

Abbildung 2: Publisch und Subscibe Ablauf

7 XMPP - Client

In Xmpp.java wird ein Xmpp Client aufgebaut.
Zuerst wird eine Verbundung zu dem Server erstellt.
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 die Funktionen in die Gui einarbeiten zu können.

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

```
public void discoverServices() throws XMPPException {
    System.out.println("Discovering Services...");
    sdm = ServiceDiscoveryManager.getInstanceFor(con);
    DiscoverItems items = sdm.discoverItems(XmppData.host);
    Iterator<DiscoverItems.Item> iter = items.getItems();
    while (iter.hasNext()) {
      DiscoverItems.Item i = iter.next();
      System.out.println(i.toXML());
    }
```

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

```
public void discoverNodes() {
    System.out.println("Discovering Nodes...");
    sdm = ServiceDiscoveryManager.getInstanceFor(con);
    DiscoverItems items = null;
    try {
      items = sdm.discoverItems("pubsub." + XmppData.host);
    } catch (XMPPException e) {
      e.printStackTrace();
      System.out.println("Discover failed");
    }
    Iterator<DiscoverItems.Item> iter = items.getItems();
    while (iter.hasNext()) {
      DiscoverItems.Item i = iter.next();
      System.out.println(i.getEntityID());
      System.out.println(i.getNode());
      System.out.println(i.getName());
    }
  }
```

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

```
public List<String> getNodes() {
    // Create a manager using an existing Connection
    ServiceDiscoveryManager mgr = ServiceDiscoveryManager
        .getInstanceFor(con);
    DiscoverItems items = null;
    try {
      items = mgr.discoverItems("pubsub." + XmppData.host);
    } catch (XMPPException e) {
      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 der Note 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:"+
            nodeID, payload);
4       PayloadItem<SimplePayload> item = new PayloadItem<SimplePayload>("
            expiration" + System.currentTimeMillis(),
5       simplePayload);
6       node.publish(item);
7       try {
8         System.out.println();
9       } catch (XMPPException e) {
10        e.printStackTrace();
```

```
11          System.out.println("Fehler beim publishen");
12        }
13      }
```

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 (XMPPException e) {
5         e.printStackTrace();
6         System.out.println("Knoten: "+nodeID+" konnte nicht geladen werden"
              );
7         return null;
8       }
9     }
```

Und am Ende wird die Verbindung wieder beendet.

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

8 Client - Entwicklung

In der Abbildung 3

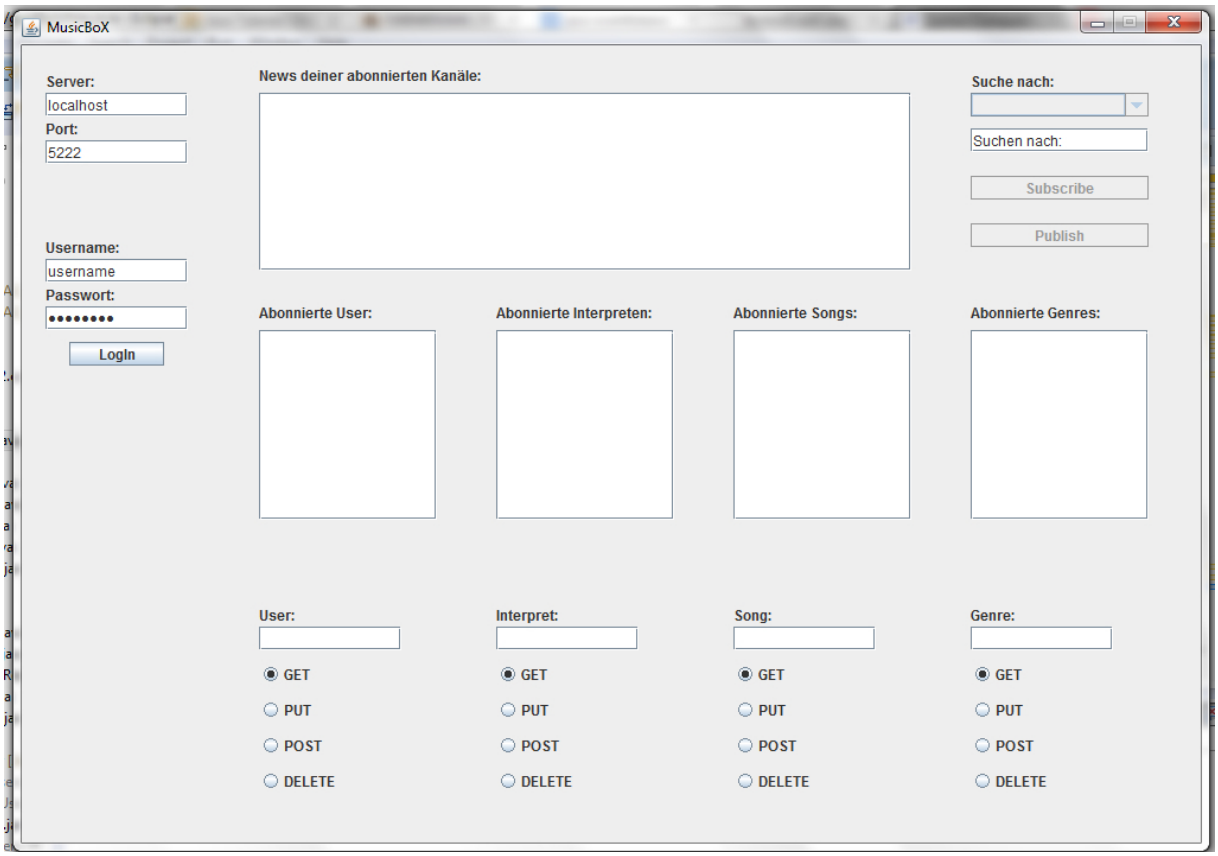

Abbildung 3: Sceenschot der GUI

9 Rückblick

9.1 Erfolge

9.2 Probleme

9.3 Verbesserungen

10 Quellenverzeichnis

10.1 Literatur

[FBBR07] Text..

10.2 Internetquellen

[REST] Einarbeitung in REST
http://openbook.galileocomputing.de/java7/1507_13_002.html

[Parameter] Erklärung der Parameter.
http://openbook.galileocomputing.de/java7/1507_13_002.html
http://www.mkyong.com/webservices/jax-rs/jax-rs-pathparam-example
http://www.mkyong.com/webservices/jax-rs/jax-rs-queryparam-example

[Publish Subscribe] Publish Subscribe Erklärung.
http://www.roboticswikibook.org/conf/display/patt/Publish-Subscribe

[Gui Tutorial] Tutorial wie man eine GUI erstellt.
http://www.java-tutorial.org/layout-manager.html

[Medieninformatik Wiki] Alle Informationen und Links die geben waren.
http://www.medieninformatik.fh-koeln.de/w/index.php/WBA2_SoSe13:Phase2