Dokumentation:
Webbasierte Anwendungen 2: Verteilte Systeme

Serien-Newsfeed

Betreut von
Prof. Dr. Kristian Fischer

Duc Duy Khuong und Stefan Börgeling

Inhalt

1. Einleitung

In dieser zweiten Phase der Veranstaltung, besteht die Aufgabe darin, die in der Vorlesung theoretisch vorgestellten Themen in die Praxis umzusetzen, d.h. es wird ein System entwickelt werden, welches den Anforderungen eines verteilten Systems entsprechen soll.
Dabei muss eine eigenständige Arbeit stattfinden, denn von der Konzeption bis hin zur Entwicklung soll alles nach und nach durchgeführt werden, bis am Ende das fertige Projekt entsteht.

2. Aufgabe

Die Aufgabe ist wie zuvor erwähnt die Entwicklung des verteilten Systems, anhand eines beispielhaften Szenarios. In mehreren Schritten soll dies stattfinden und über diverse Meilensteine können die Fortschritte festgehalten werden. Die Meilensteine sehen wie folgt aus:

(0. Konzeptioneller Meilenstein - Kommunikationsabläufe und Interaktionen)
1. Projektbezogenes XML-Schema erstellen
2. Ressourcen und die Semantik der HTTP-Operationen festlegen
3. RESTful Webservice implementieren
4. Konzeption asynchrone Kommunikation aufstellen
5. Arbeit mit dem XMPP-Client
6. Entwicklung des Clienten

3. Konzeptioneller Meilenstein

In dieser ersten Phase des Projekts ist zunächst die Frage nach dem Thema bzw. Szenario, welches umgesetzt werden soll das Hauptaugenmerk. Ein erstes Brainstorming wurde durchgeführt und die Idee ist entstanden, eine Art Video-Plattform zu entwickeln, welche von verschiedenen Quellen Videos abrufen kann und in einem Feed zusammengefasst darstellen soll. Die Idee hinter diesem System ist das Prinzip des Publish-Subscribe, d.h. es ist möglich verschiedene Themen zu abonnieren und Informationen dazu abzurufen, in diesem Fall die Videos.

Bei weiterer Überlegung ist dies jedoch schnell verworfen worden, da zeitgleich eine zweite Idee aufgekommen ist und zwar die Entwicklung eines Nachrichtenfeeds für Serien.
Mit Blick auf die zukünftigen Meilensteine ist klargeworden, dass an dem Videofeed nicht viel Spielraum für z.B. die HTTP-Operatoren sind, da die Videos an sich nicht bearbeitet werden können, da diese bereits feste Dateien sind und sonst kaum weitere Möglichkeiten bestehen würden Informationen auszutauschen oder zu ändern.

Daher ist die Entscheidung letztendlich auf den Serien-Newsfeed gefallen.
Hierbei ist es so gedacht, dass der Nutzer sich einen eigenen Feed erstellen kann, den er jederzeit abrufen kann um News zu seinen Lieblingsserien zu erhalten. Auch diese Idee soll auf dem Publish-Subscribe-System basieren. Dem Nutzer ist es also möglich, einzelne Serien zu abonnieren und dazu

allgemeine Informationen sowie aktuelle News abzurufen, während gleichzeitig auch Kommentare verfasst oder andere Elemente bearbeitet werden können.
Weiterhin, können Filter erstellt werden, die die Nachrichten nach bestimmten Eigenschaften filtern sollen, z.B. nach Genre oder nach bestimmten Serien etc.

Der Kommunikationsablauf sieht so aus, dass bei der synchronen Kommunikation die allgemeinen Informationen, wie der Cast, die Episodenliste oder sonstige Informationen direkt abgerufen werden können.
Die asynchrone Kommunikation soll so stattfinden, dass News ohne Interaktion des Nutzers abgerufen werden, sobald neue vorhanden sind.

Die Interaktionen, die dem Nutzer möglich sind:

- Login/(Registration)
- Abonnieren und Bearbeiten der Serien
- Erstellen und Bearbeiten der Filter
- Aufrufen der News
- Kommentare verfassen/anzeigen lassen

Der Login bzw. die Registration dient zur individuellen Personalisierung des Feeds, der Nutzer kann sich also einloggen und ein auf sich angepassten Feed erstellen und speichern.

Die Daten, die zwischen dem Client und Server übertragen werden, sind :

- die News an sich
- Informationen: Episodenliste, Cast
- Kommentare
- Anmelde-/Profilinformationen
- Filterinformationen

Vorläufiges Anwendungsfalldiagramm für das Konzept

4. Meilenstein 1: Projektbezogenes XML-Schema

Die ersten beiden Meilensteine dienen als Basis für die weitere Entwicklung der Anwendung.
Mit Hilfe des aufgestellten Konzepts wird das XML-Schema erstellt.
Am Anfang ist es nötig zu klären wie viele Schemata entwickelt werden sollen. All die Informationen die übertragen werden, müssen in die Schemata eingetragen werden.

Zum einen gibt es da alle Informationen, die die Serien selber betreffen und für diese von Bedeutung sind, dann sind da die Informationen, die aufgrund des Feeds selbst auftauchen. Also wurde überlegt, dies in mehrere XML-Schemata aufzuteilen. Dabei sind jeweils für den Feed, das Profil, die Serie, die Episoden, den Cast und die Kommentare ein Schema entstanden.
Das, welches die Informationen über den Feed enthält, also die News an sich, beinhaltet das Bild, einen Link zum Artikel, den Titel und eine kurze Beschreibung des Inhalts.

Das Schema zur Serie soll den Namen, das Genre sowie eine Beschreibung der Serie enthalten. Zusätzlich zu den beiden sind für die jeweiligen Schauspieler und Episoden sowie für verfasste Kommentare Schemata erstellt worden.

Nach Absprache mit den Tutoren hat sich herausgestellt, dass es sinnvoller ist mehrere Schemata zusammenzufassen. Die beiden für Schauspieler und Episoden werden in das Serien-Schema integriert, da diese untergeordnete Punkte sind, ebenso wie die Kommentare zu den Nachrichten gehören. Dadurch verringert sich die Anzahl benötigter XML-Dateien und es können mehr Informationen über eine Datei übertragen werden.

Was zusätzlich noch verbessert werden muss, sind zusätzliche Restriktionen, die die verschiedenen Werte begrenzt und bestimmte Bedingungen an die Elemente setzt.

Letztendlich ergeben sich 3 Schema-Dateien, aus denen die XML-Dateien generiert werden können, welche im nächsten Meilenstein verwendet werden.

5. Meilenstein 2 + 3: Ressourcen und die Semantik der HTTP-Operationen/RESTful Webservice

5.1 RESTful Webservice

Im zweiten Meilenstein geht es darum sich in die RESTful Webservices einzuarbeiten und daraufhin zu versuchen eine synchrone Kommunikation zwischen Client und Server zu entwickeln.
Hierbei wird versucht eine möglichst lose Kopplung zwischen den beiden Komponenten zu etablieren, d.h. die Verbindung zwischen zwei Systemen sollte sich auf eine effiziente Kommunikation beschränken und nicht komplett voneinander separat agieren, aber so, dass nur eine geringe Kopplung vorhanden ist. Synchrone Kommunikation bedeutet, dass der Client eine Anfrage losschickt und der Server direkt im Anschluss darauf antwortet, d.h. immer wenn der Client eine Anfrage stellt, muss der Server direkt antworten.
Der Architekturstil Representional State Transfer , abgekürzt REST, besitzt einige grundlegende Prinzipien, die folgendermaßen aussehen:
Da wäre zunächst das im Web einheitlich eingeführte Identifikationssystem, wodurch bestimmte Elemente eine bestimmte ID zugeteilt bekommen um eine eindeutige Identifikation zu ermöglichen, dies geschieht in Form von URIs (Uniform Resources Identifier).

Beim REST ist ein zentraler Aspekt die Ressource, bzw. dessen Repräsentation. Es können hierbei beliebig viele Ressourcen definiert werden, z.B. als Nachricht, Person etc.
Diese sind an sich nicht direkt sichtbar, sondern nur in Form von Repräsentationen.
Um auf Ressourcen, die auf einem Webserver liegen zugreifen zu können müssen diese eindeutig, wie zuvor bereits erwähnt, über eine URI identifizierbar sein.
Über die URI ist es dann möglich verschiedene Operationen über HTTP-Operatoren auf die Ressourcen anzuwenden. Die Standardoperationen dafür sind :

HTTP-Operator	Operation
GET	Anforderung der Repräsentation einer Ressource (Lesezugriff)
POST	Anlegen einer neuen Ressource. Rückgabe ist die URI zur neuen Ressource (Schreibzugriff)
PUT	Aktualisierung einer Ressource
DELETE	Löschen von Ressourcen
HEAD	Anforderung der Metadaten einer Ressource
OPTIONS	Anforderung der möglichen Operationen einer Ressource

Tabeller der HTTP-Operatoren

Die Kernaufgabe dieses Meilensteinabschnittes besteht darin, diese Operationen programmiertechnisch umzusetzen, in dem Fall für die wichtigsten GET, POST, PUT und DELETE.
Um damit beginnen zu können müssen im Vorfeld die Ressourcen festgelegt werden, auf die die späteren Operationen angewendet werden können.

5.2 Ressourcen

Es wird anhand der XML-Schemata überlegt, welche Daten als Ressourcen definiert werden sollen.
Da es drei verschiedene Schemata gibt, ist der Schritt sinnvoll, diese als Primärressourcen zu definieren, demnach gibt es also Nachrichten, Serien und Profile. Diese sind aber nicht die einzigen sondern beinhalten Subressourcen, die ebenfalls über die HTTP-Operationen angesprochen werden sollen.
Folglich ergeben sich für die in dem Projekt relevanten Ressourcen:

5.2.1 NachrichtenService

- Nachrichten (/news)

GET	Gibt eine Liste aller Nachrichten zurück
POST	Erstellt eine neue News und fügt sie hinzu

- News(/news/{news_id})

GET	Gibt eine bestimmte News zurück
PUT	Verändert eine bereits vorhandene News oder fügt sie hinzu, wenn diese nicht vorhanden ist
DELETE	Löscht eine bestimmte News

- Kommentare(/news/{news_id}/kommentare)

GET	Gibt eine Liste aller Kommentare zu einer bestimmten News zurück
POST	Erstellt einen neuen Kommentar und fügt ihn hinzu

- Kommentar(/news/{news_id}/kommentare/{kommentar_id})

GET	Gibt einen bestimmten Kommentar einer bestimmten News zurück
DELETE	Löscht einen bestimmten Kommentar, der zu einer News verfasst wurde
PUT	Bearbeitung eines Kommentares oder hinzufügen wenn dieser noch niciht vorhanden ist

5.2.2 SerienService

- Serien(/serie)

GET	Gibt eine Liste aller Serien zurück
POST	Erstellt eine neue Serie und fügt sie hinzu

- Serie(/serie/{serienname})

GET	Gibt die Serie mit dem bestimmten Namen zurück
DELETE	Löscht eine bestimmte Serie

- Serie(/serie/{serien_id})

PUT	Verändert eine bereits vorhandene Serie oder fügt eine hinzu falls sie noch nicht vorhanden ist

- SerienID(/serie/{serienname}/id)

GET	Gibt die ID einer bestimmten Serie zurück

- Staffeln(/serie/{serienname}/staffel)

GET	Gibt eine Liste aller Staffeln zu einer Serie aus
POST	Erstellt eine neue Staffel und fügt sie hinzu

- Staffel(/serie/{serienname}/staffel/{staffel_id})

GET	Gibt eine Staffel zu einer bestimmten Serie aus
DELETE	Löscht eine Staffel einer Serie
PUT	Verändert eine vorhandene Staffel oder fügt eine neue hinzu falls nicht vorhanden

- Episode(/serie/{serienname}/staffel/{staffel_id}/episode/{episoden_id})

GET	Gibt die Episode einer bestimmten Staffel einer bestimmten Serie aus
DELETE	Löscht diese Episode

- Episoden(/serie/{serienname}/staffel/{staffel_id}/episode/)

POST	Erstellt eine neue Episode

- Cast(/serie/{serienname}/cast)

GET	Gibt die Liste des Casts einer Serie aus
PUT	Verändert den Cast, d.h. fügt Schauspieler hinzu oder löscht sie
POST	Fügt einen Darsteller zum Cast hinzu

-
- Cast(/serie/{serienname}/cast/{darstellername})

DELETE	Löschen eines bestimmten Darstellers

5.2.3 ProfileService

- Profile(/profile)

GET	Gibt die Liste aller Profile aus
POST	Erstellen eines neuen Profiles

- Profil(/profile/{profil_id}/)

GET	Gibt ein bestimmtes Profil aus
PUT	Verändert ein bestimmtes Profil
DELETE	Löscht ein bestimmtes Profil

- Filter(/profile/{profil_id}/filter)

GET	Gibt die angelegten Filter eines gewählten Profil aus
POST	Erstellt einen neuen Filter

- bestimmter Filter(/profile/{profil_id}/filter/{filtername})

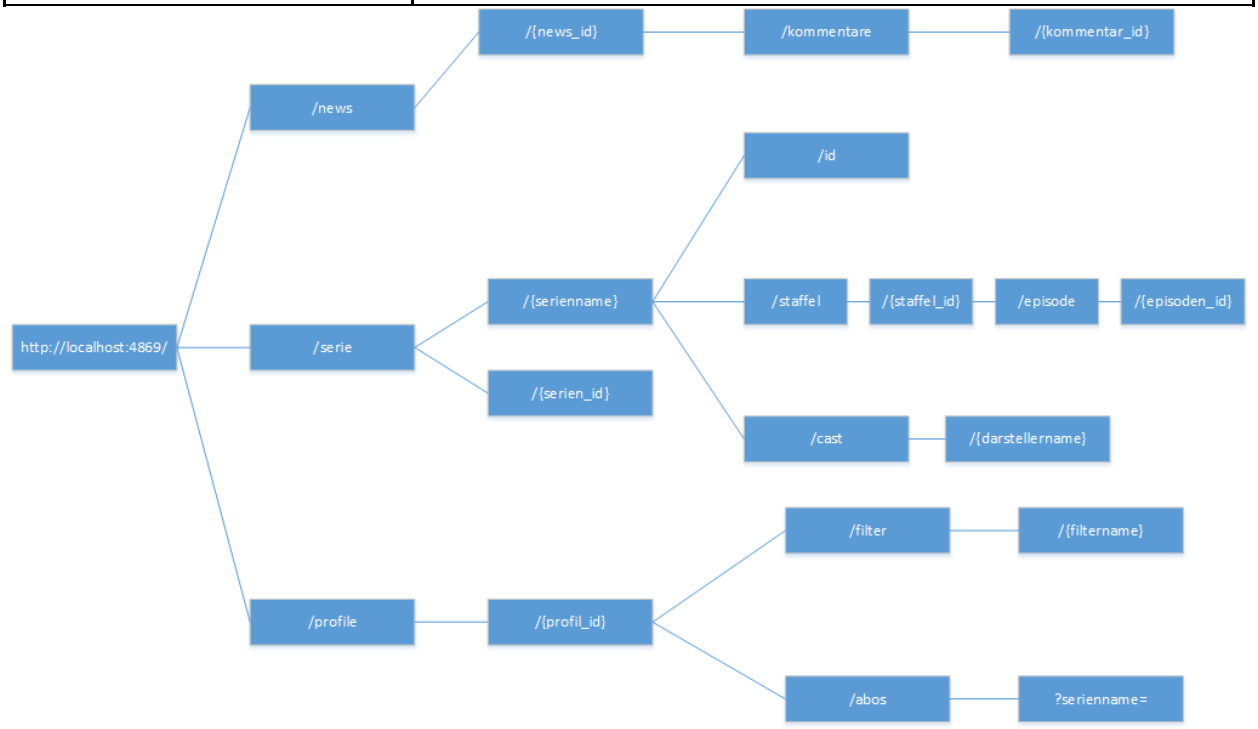
GET	Gibt einen bestimmten Filter eines Profils aus
DELETE	Löscht einen bestimmten Fehler

- Abonnement(/profile/{profil_id}/abos)

GET	Gibt eine Liste alle abonnierter Serien aus

- Serie(/profile/{profil_id}/abos?serienname=)

POST	Fügt eine neue Serie zu den Abos hinzu
DELETE	Löscht die Serie aus den Abos



Zusammenfassung aller Ressoucen im Baumdiagramm

5.3 URI-Bildung

Wie zu sehen ist, kann fast jede Ressource bearbeitet bzw. abgerufen werden. Einzige Ausnahmen sind der Cast, der nur abgerufen und verändert werden kann, weil jeder Serie immer einen festen Cast hat und sich nur geringfügig ändert. Eine Episode kann hingegen nur abgerufen, gelöscht und erstellt werden.
Eine Problematik, die aufgedeckt wird, ist die Länge einiger URIs. Die Idee hinter ist, die Hierarchie der Primär- und Subressourcen so gut widerzuspiegeln, wie sie bereits in den XML-Schemata vorhanden ist. Außerdem ist es für den User später leichter sich durch die URI finden, da sie selbsterklärend ist.

```
http://localhost:4869/serie/{serienname}/staffel/{staffel_id}/episode/{episoden_id}
```

URI-Beispiel ohne eingesetzte Parameter

Anhand des oberen Beispiels lässt sich das besser verdeutlichen. Wenn man nun auf eine bestimmte Episode einer Serie zugreifen muss, um dazu z.B. Informationen erhalten zu können, muss man drei Parameter übergeben, erstens: den Seriennamen, zweitens: die Staffel, die die Episode beinhaltet und die Episoden-ID, welche hier gleichzusetzen ist mit der Episodennummer.

Hier erkennt man gut wie sich die Hierarchie in der URI abbildet. Man hat die Serie, wozu es diverse Staffeln geben kann, die ihr untergeordnet sind und die Episoden sind logischerweise der Staffel untergeordnet, da die Staffel eine Liste der Episoden beinhaltet.
Ausgefüllt sieht die URI wie folgt aus:

```
http://localhost:4869/serie/Dexter/staffel/1/episode/2
```

URI-Beispiel mit eingesetzten Parametern

Hier greift man auf die zweite Episode der ersten Staffel von der Serie Dexter zu und kann darauf nun die HTTP-Operationen anwenden, die mit Hilfe des Jersey Frameworks realisiert werden können. Das Framework ermöglicht es, die verschiedenen Annotationen im REST-Service zu nutzen und zudem den REST-Service in Java zu implementieren.

5.4 Implementierung und Codebeispiele

Nachdem die Ressourcen festgelegt sind beginnt die Implementierung der einzelnen Services. Als Grundlage dienen dazu die erstellten XML-Dateien und Schemata. Die XML-Dateien werden benötigt für das Testing der jeweiligen Operationen, die Schemata werden dazu verwendet die JAXB-Klassen erstellen zu können.
Diese ermöglichen es auf die einzelnen Elemente der XML-Dateien zuzugreifen und diese nach belieben zu manipulieren.

Bevor mit der Programmierung der Service begonnen werden kann, muss ein REST-Server konfiguriert werden, der die Simulation der Client/Server-Kommunikation ermöglicht.
Dafür wird eine Klasse “RestServer” angelegt, in der die Konfiguration des Servers gespeichert wird:

```
public class RestServer {


	public static void main(String[] args) throws Exception {


		String url = "http://localhost.com:4869";

		SelectorThread srv = GrizzlyServerFactory.create(url);

		System.out.println("Server wurde gestartet" + "\n" + "URL: " + url);
		Thread.sleep(1000 * 60 * 10);
		srv.stopEndpoint();
		System.out.println("Server wurde beendet");
	}

}
```

Klasse für den Restserver

Man verwendet hierbei das Grizzly Framework, welches erlaubt den Server zu simulieren.
Zuerst wird die gewünschte URL erzeugt, die ”http://localhost.com:4869” lautet. Mit dieser URL wird dann der Server gestartet. Danach kann man noch wahlweise angeben nach wie viel Zeit der Server wieder beendet werden soll, in dem Fall sind es 10 Minuten.
Wenn man die Datei nun ausführt startet der Server und man kann über einen beliebigen Browser mit der URL auf den Server zugreifen.
Nun geht es daran mit der Implementierung der HTTP-Operationen zu beginnen.
Nachfolgend wird beispielhaft ein Service, der SerienService näher beleuchtet, um die REST-Methoden einmal repräsentativ zu erklären und den Code etwas zu beleuchten.

Eine Voraussetzung muss allerdings vorher noch erfüllt sein, damit die XML-Dateien verwendet und verändert werden können. Dazu wird das Verfahren des Unmarshalling und Marshalling verwendet. Das Unmarshalling wandelt die XML-Datei in Java-Objekte um, wodurch es möglich ist, diese zu modifizieren. Um die Änderung wieder in die XML zu schreiben verwendet man das Marshalling, das den umgekehrten Vorgang durchführt. Was beachtet werden muss ist, dass bei den Methoden der richtige Pfad zu den verwendeten XML verwendet wird.

Um anzufangen erstellt man sich zunächst die Klasse für seinen gewählten Service, in dem Fall “SerienService”. Um die URI nachher wie gewünscht zu erhalten müssen Pfade mit Hilfe der @Path Annotation angegeben werden.

```
@Path("/serie")
public class SerienService {

}
```

Beispiel für Path-Angabe am Klassenkopf

Dieses Beispiel bedeutet, dass alle URIs auf denen die noch folgenden HTTP-Operationen angewendet werden mit “/serie” beginnen müssen.

Die Implementierung der GET-Methode ist vom Codeumfang her am kleinsten, da bei der Serien XML-Datei bereits nach dem Unmarshallen eine Liste der Serien ausgegeben wird.

```
@GET
@Produces(MediaType.APPLICATION_XML)
public Serien getSerien() throws JAXBException, FileNotFoundException {

	Serien serien = unmarshalSerien();

	return serien;

}
```

Beispiel für die Implementierung der GET-Methode

Vor jedem Methoden-Kopf muss angegeben werden, welche HTTP-Operation mit der Methode ausgeführt werden soll. Außerdem kann man mit `@Produces` und `@Consumes` spezifiziert werden welche Form der Repräsentation übertragen wird. “Produces” gibt die Repräsentation an, die an den Client gesendet wird und “Consumes” was die Ressource für eine Repräsentation vom Client erwartet wird.

Im obigen Beispiel wird die GET-Operation auf die Ressource mit der URI `”http://localhost.com:4869/serie”` angewendet, also eine Anforderung der gesamten Serien-Liste an den Server. Das zurückgelieferte Ergebnis ist folglich die Liste in Form von XML:

```
<serien>
	<serie id="1">
		...
	</serie>
	<serie id="2">
		...
	</serie>
	<serie id="3">
		...
	</serie>
</serien>
```

Ausgabe des Servers nach dem GET-Request

Dieser Request gibt nun die gesamte Liste aus, doch wenn man nun eine bestimmte Serie aus der XML-Datei extrahieren möchte, kann man eine weitere GET-Methode implementieren, die dies ermöglicht.

Wie schon zuvor geschehen, setzt man die GET-Annotation und den Path. Hier kann man nun die URI erweitern und zwar um den Parameter “Serienname”, man setzt diesen in geschweifte Klammern um zu verdeutlichen, dass es sich um Paramater handelt.
Die Path-Annotation sieht dann wie folgt aus, @Path:(“/{serienname}”) und die sich daraus ergebende URI: ”http://localhost.com:4869/serie/{serienname}”.
Wenn man nun auf diese Ressource die GET-Operation ausführt, erhält man die gewünschte Serie.

```
@GET
@Path("/{serienname}")
@Produces(MediaType.APPLICATION_XML)
public Serie getSerie(@PathParam("serienname") String serienname)
            throws JAXBException, FileNotFoundException {

      Serien s_daten = unmarshalSerien();
      Serie s = null;

      for (int i = 0; i < s_daten.getSerie().size(); i++) {
            if (serienname.equals(s_daten.getSerie().get(i).getName())) {
                  s = s_daten.getSerie().get(i);
            }
      }
      return s;
}
```

GET-Methode zum Abrufen von einer Serie

Hier wird zunächst wieder die Liste mit Hilfe des Unmarshalling abgerufen und in die Variable “s_daten” der Klasse Serien gespeichert. Dann wird eine Variable “s” der Klasse Serie mit null initialisiert, in welcher später die Serie gespeichert wird.
Mit der For-Schleife wird die komplette Liste durchgegangen und die if-Anweisung prüft derweil, ob der übergebene Parameter “serienname” mit einer der Namen in der XML-Datei übereinstimmt. Wenn dies der Fall ist wird die Serie aus der Liste geholt und in die Variable “s” gespeichert und schließlich zurückgegeben.

Die Ausgabe sieht dann folgendermaßen aus:

```
<serie id="1">
      ...
</serie>
```

Ausgabe der getSerie()-Methode

Nachdem das Abrufen und Ausgeben der Serien nun funktioniert, geht es weiter mit den Methoden zur Modifizierung der XML-Dateien, also der POST-, PUT- und DELETE-Operation.

Die POST-Operation ist dazu da, neue Elemente zur bestehenden XML-Datei hinzuzufügen. Man könnte auch sagen, dass PUT dafür zuständig sei und dies wäre grundsätzlich auch nicht falsch, je nach Implementierung können beide Methoden dafür stehen Sachen hinzuzufügen oder zu verändern, das ist Auslegungssache. Hier ist entschieden worden die POST-Methode auf das hinzufügen zu beschränken und demnach die PUT-Methode dazu zu verwenden, Veränderungen durchzuführen.

```
@POST
@Consumes(MediaType.APPLICATION_XML)
@Produces(MediaType.APPLICATION_XML)
public Response postSerie(Serie s) throws JAXBException,
            FileNotFoundException {

      Serien s_daten = unmarshalSerien();

      s.setId(nextIdSerie());

      s_daten.getSerie().add(s);
      marshalSerien(s_daten);

      return Response.status(201).build();
}
```

POST-Operation für eine Serie

Diese POST-Methode wird, wie die GET-Methode für die Liste, auf die URI ”http://localhost.com:4869/serie” ausgeführt, da das Element nachher an die Liste angehängt wird. Die Repräsentation für die wechselseitige Kommunikation ist hier ebenfalls die XML-Form.

Wie man an dem Methodenkopf erkennen kann hat diese Methode einen Rückgabewert “Response”, das heißt, es wird am Ende ein HTTP-Statuscode zurückgegeben, der das Ergebnis der HTTP-Operation ausgibt. Es gibt Codes im Bereich 100-500, wobei die im Bereich 100 zu Informationszwecken dienen, die im Bereich 200 stehen für eine erfolgreiche Verarbeitung, z.B. 200 für Ok, 201 für Created oder 204 für No Content. Der 300er-Bereich steht für eine Umleitung, der Server informiert den Client, dass Umleitung erfolgt. Stauscodes, die mit 400 beginnen stehen für Fehler auf Client-Seite, z.B. 300 für Bad Request oder 404 für Not Found, die 500er-Codes stehen parallel dazu für Fehler auf Serverseite.

In dieser Methode wird der Statuscode 201, also Created, zurückgegeben, wenn die Serie erfolgreich erstellt wurde.

Um eine Serie hinzufügen zu können muss diese an die Methode übergeben werden, wie man im Kopf erkennen kann. Es muss eine Serie in Form des eines JAXB-Objekts Serie übergeben werden. Folglich muss die Übergabe in Form von XML-Code stattfinden und die Semantik der ursprünglichen XML-Datei eingehalten werden:

```
<serie>
        ...
</serie>
```

Übergabe an die POST-Methode

In der Methode wird zunächst wieder die Serien unmarshallt und in die Variable s_daten abgespeichert. Im nächsten Schritt wird die ID der neuen Serie gesetzt und wie man sieht wird dafür die Methode nextIdSerie() aufgerufen, die so aussieht:

```
public String nextIdSerie() throws JAXBException, FileNotFoundException {
	List<Serie> list = unmarshalSerien().getSerie();
	int id = Integer.parseInt(list.get(list.size() - 1).getId());

	if (list.size() > 0) {
		id++;
	} else {
		id = 0;
	}
	return String.valueOf(id);
}
```

Methode nextIdSerie() zur Berechnung der ID

Diese Methode ist dazu da, die neue ID für das Element zu berechnen. Dazu werden die Serien in eine Liste gespeichert sowie die ID des letzten Objekts in der Liste. Dann wird eine Abfrage durchgeführt für den Fall, dass die Liste leer ist. Der Rückgabewert ist dann die ID des letzten Objekts, die um eins erhöht wurde.

Nachdem die ID erfolgreich gesetzt wurde, wird die Serie der Liste hinzugefügt und alles wieder in die XML-Datei gemarshallt und wenn alles erfolgreich abgelaufen ist, erhält man den Statuscode 201.
Als nächste Operation ist PUT an der Reihe:

```
@PUT
@Path("/{serien_id}")
@Consumes(MediaType.APPLICATION_XML)
@Produces(MediaType.APPLICATION_XML)
public Response putSerie(@PathParam("serien_id") String serien_id, Serie s)
            throws JAXBException, FileNotFoundException {

      Serien s_daten = unmarshalSerien();
      int id_temp = Integer.parseInt(serien_id);

      //Wenn Serien bereits vorhanden ist, dann verändern
      for (int i = 0; i < s_daten.getSerie().size(); i++) {
            if (serien_id.equals(s_daten.getSerie().get(i).getId())) {

                  String id = s_daten.getSerie().get(i).getId();
                  s.setId(id);
                  s_daten.getSerie().set((Integer.parseInt(id)-1), s);//Vermeidung
von IndexOutOfBoundsException
                  break;
                  }
                  //Wenn Serie nicht vorhanden ist, dann hinzufügen
            if (id_temp > s_daten.getSerie().size()) {
                  s.setId(nextIdSerie());
                  s_daten.getSerie().add(s);
                  break;
            }
      }
      marshalSerien(s_daten);

      return Response.ok().build();
}
```

PUT-Operation für eine Serie

Hier muss im Gegensatz zur POST-Methode wieder ein Path angegeben werden, da ein bestimmtes Element bearbeitet werden soll und dazu eine Identifikation notwendig ist. Anstatt des Seriennamen wird hier die ID der Serie verwendet, da es Probleme mit dem ID setzen und der Überprüfung, ob die Serie vorhanden ist, gab.

Wie schon zuvor muss auch hier eine Serie in Form von XML-Code übergeben werden um das gewünschte Element zu ändern.
Der erste Schritt ist das unmarshallen, speichern in eine Liste und das Anlegen einer Variable, die die Serien-ID speichert, die beim Ausführen der Operation in der URI übergeben wird. Diese wird später noch von Bedeutung sein.

Jetzt beginnt die Überprüfung, ob die Serie schon vorhanden ist. Es wird zunächst die gesamte Liste durchgegangen und überprüft ob die übergegebene ID mit einer ID übereinstimmt, die bereits vorhanden ist. Ist dies der Fall, wird die ID, dieses Objekts an das das neue Element Serie übergeben und die alte Serie wird überschrieben. Hier ist es wichtig dass, bei der set()-Methode die richtige Stelle an der Liste eingegeben wird, damit diese auch verändert wird und nicht fälschlicherweise eine andere. Dazu nimmt man die ID des zu überschreibenden Objekts und zählt diese um eins runter, um die Serie mit der gewünschten ID zu erhalten, da die Liste üblicherweise mit 0 als erster Stelle beginnt.
Das vermindern um eins verhindert zudem eine IndexOutOfBoundsException, die auftreten kann, wenn die angegebene ID größer ist als die Liste an sich.

Dies ist im Nachhinein überflüssig geworden, da anstatt der ID nun der Serienname übergeben wird anstatt einer Zahl als ID.

Doch für diesen Fall gibt es eine if-Abfrage, und zwar braucht man hier die zuvor angelegte Variable id_temp wieder und prüft ob diese größer als die Größe der Liste an Serien ist. Trifft das zu, wird die Serie einfach der Liste angehängt, da sie noch nicht vorhanden ist.

Ein Problem welches hierbei zunächst auftritt ist das mehrfache erstellen einer Serie, da die for-Schleife komplett durch die Liste geht und die Abfragen jedes Mal ausführt. Wenn aber die ID z.B. 10 Stellen größer ist als die Liste, werden soviele erstellt, bis die Zahl gleich ist. Gelöst wird dieses Problem durch das break, wenn die Serie einmal hinzugefügt wurde, wird aus der Schleife gesprungen und nicht sie wird nicht weiter durchlaufen. Ebenso passiert dies beim Fall, wenn die Serien schon vorhanden ist, sobald sie gefunden und geändert wurde, ist es nicht mehr von Nöten die Schleife weiter zu durchlaufen und die Methode springt wieder raus.

Am Ende wird alles wieder in die XML gemarshallt und man erhält einen Response-Code “Ok”.

Die letzte HTTP-Operation, die nun noch fehlt ist die DELETE-Operation, welche das Löschen bestimmter Elemente “Serien” aus der Liste ermöglicht.
Es wird erneut eine Serie aus der Liste über den Parameter “seriennamen” ausgesucht, ist diese jedoch nicht vorhanden, gibt es einen Response-Code 404 “Not Found” vom Server zurück. Davor überprüft die Methode ob es die Serie mit dem Namen schon gibt, falls ja wird diese gelöscht und einen Ok Response-Code zurückgegeben.

```
@DELETE
@Path("/{serienname}")
@Produces(MediaType.APPLICATION_XML)
public Response deleteSerie(@PathParam("serienname") String serienname)
		throws JAXBException, FileNotFoundException {

	Serien s_daten = unmarshalSerien();

	for (int i = 0; i < s_daten.getSerie().size(); i++) {
		if (serienname.equals(s_daten.getSerie().get(i).getName())) {
			s_daten.getSerie().remove(i);
			marshalSerien(s_daten);
			System.out.println("Serie wurde gelöscht");
			return Response.ok().build();
		}
	}
	System.out.println("News wurde nicht gefunden. ");
	return Response.status(404).build();
}
```

DELETE-Operation für eine Serie

Während des Prozesses der Implementierung der einzelnen REST-Services sind einige Probleme aufgetreten, die während des Testens aufgetreten sind.
Ein schwerwiegendes Problem ist, dass ein Fehler aufgetreten ist, der nichts mit dem Code an sich zu tun hat, sondern mit der Schema-Datei. Beim Ausführen einer HTTP-Operation wurde immer eine traceException geworfen, aber es ist nicht klar geworden, was die Ursache dafür ist. Nach Absprache mit Kommilitonen, bei denen das gleiche Problem auftritt ist festgestellt worden, dass die Schema-Datei, die beim jeweiligen Service verwendet wird, die Ursache des Problems ist.
Das Schema muss so umgeschrieben werden, dass die hierarchische Verschachtelung an sich erstmal aufgelöst und über einen anderen Weg gelöst wird.
Alle Elemente mit “complex Type”-Elementen müssen “ausgelagert” werden und mit einer Referenz an in der ursprünglichen Position referenziert werden (siehe XML-Schema im GitHub). Das Problem ist zunächst nur bei einem Service aufgetreten, während bei den anderen noch alles normal abgelaufen ist. Später ist dieser Fehler beim nächsten Service aufgetreten und die Schema-Datei musste ebenfalls verändert werden. Als Präventionsmaßnahme sind alle XML-Schemata auf die gleiche Weise verändert werden um dem Problem aus dem Weg zu gehen.

Weiterhin wird beim 3. Meilenstein verlangt, dass PathParams und QueryParams jeweils mindestens einmal verwendet werden muss. Das ist eingehalten, aber trotzdem hat es ein Problem bei den QueryParams gegeben. An manchen Stellen ist es nicht möglich diese z.B. anstatt von PathParams einzusetzen, denn dann kommt eine Fehlermeldung die besagt : ...Service has ambiguous resource for HTTP method GET and output mime-type: application/xml. Dieses Problem tritt aber nicht bei PathParams auf, weshalb an einigen Stellen statt geplanten QueryParams nun PathParams verwendet werden.

Des Weiteren ist die Entscheidung gefallen Genres bei den Filtern und Abonnements außen vor zu lassen, da die Umsetzung zeitlich und programmiertechnisch problematisch geworden ist.
Das Problem ist die Menge der Ressourcen, die bereits von Anfang vorhanden ist, diese alle umzusetzen ist nicht innerhalb der vorgegebenen Zeit komplett gelungen. Darum ist Genre als Ressource entfernt worden.

Ein weiteres Problem das entdeckt wurde, nachdem die Services implementiert waren, sind die Filter-IDs und Serien-IDs, die bei den POST und DELETE Methoden für die Ressourcen “bestimmte Filter” und “Serie” in den Abonnements verwendet wurden. Das ist geändert worden, da diese bestimmte Namen und in der späteren GUI ebenfalls benannt werden können und bisher ist das Abrufen und die Identifikation über IDs gelaufen, die nur bestimmte Zahlen sind, die aber als String gespeichert worden. Die Methoden diese Parameter benötigen sind nun angepasst und verändert und man kann die Filter und Serien in den Abos sowie in den Filtern über ihre Namen ansprechen.

Nachdem die RESTful-Webservices soweit implementiert sind, geht es nun darum im nächsten Meilenstein die asynchrone Kommunikation zwischen Client und Server zu realisieren, was über XMPP geschehen wird.

6. Meilenstein 4+5: Konzeption asychrone Kommunikation + XMPP-Client

6.1 Publish-Subscribe und XMPP

In diesem Abschnitt des Projekts dreht sich alles darum, nachdem die synchrone Kommunikation zwischen dem Client und dem Server bereits funktioniert, die asynchrone Kommunikation zu ermöglichen.
Schon zu Anfang ist gesagt worden, dass das System auf einem Publish-Subscribe Prinzip beruhen soll, das heißt es gibt zwei Entitäten, den Abonnierer (Subscriber) und den Veröffentlicher (Publisher). Der Subscriber kann Themen (Topics) abonnieren und kann daraufhin Nachrichten vom Publisher erhalten, z.B. wenn ein neues Ereignis eintritt o.Ä. Die Umsetzung wird mit Hilfe des XMPP erreicht.
XMPP (Extensible Messaging and Presence Protocol) ist ein Internetstandard, der es ermöglicht Echtzeit-Kommunikation zwischen zwei Entitäten zu ermöglichen. Dazu wird der XML-Standard verwendet und bei der Kommunikation werden jeweils XML-Dateien übertragen.

Das XMPP ist eine Weiterentwicklung von Jabber und beinhaltet daher dessen Funktionen und Anwendungsgebiete, welches primär das Instant Messaging ist. Jedoch ist das nicht die einzige Funktion von XMPP, dazu gehört unter Anderem auch Peer-to-Peer Sessions zu erstellen oder One-to-One-Messaging, das Multi-Party-Messaging oder das für das Projekt relevante Publish-Subscribe.

Wie schon erwähnt wird bei der Kommunikation auf den XML-Standard gesetzt und in Folge dessen ergibt sich eine Semantik, die dabei eingehalten werden muss, zusammengesetzt aus sogenannten XML-Stanzas.

Es gibt drei Typen von Stanzas:

- **iq**
  Abkürzung für Info/Query, hiermit kann kann eine Request-Response-Kommunikation realisiert werden, also kann eine Entität eine Anfrage stellen und daraufhin eine Antwort bekommen. Besonders beim Publish-Subscribe wird dieses Stanza verwendet.
- **message**
  Wird für das Versenden einer Nachricht von einer Entität zur Anderen verwendet
- **presence**
  Dieses Stanza ist dazu da um von einer Entität presence-Anfragen an andere Entitäten zu verschicken um zu überprüfen, ob diese verfügbar sind. Ein Beispiel ist die Kontaktliste beim Instant-Messaging-Service, wo angezeigt wird ob eine Person online ist oder nicht.

Beim XMPP erhält jede Entität eine eindeutige Identifikation durch seine Jabber ID (JID), diese wird für die Kommunikation im Zusammenhang mit den Stanzas verwendet. Außerdem besitzen die drei Stanza-Typen Attribute die bei der Kommunikation verwendet werden und bei allen gleich sind.

- from Absender des Stanzas (JID)
- to Empfänger des Stanzas (JID)
- id ID für die Verarbeitung des Stanzas
- type Spezifizierung für das Stanza
- xml:lang Sprache des Inhalts der XML

Um weitere Informationen über eine bestimmte Entität zu erlangen gibt es das “Service Discovery”. Es ermöglicht zum einen Funktionen, Identifizierung und Protokolle in Erfahrung bringen zu können und zum anderen alle Items abzufragen, die mit einer Entität assoziiert werden. Dies ist wichtig für die Entwicklung des XMPP-Clients im Hinblick auf die Leafs.

6.2 Leafs (Topics)

Im Publish-Subscribe-Prinzip des XMPP kann eine Entität von sogenannte Leafs (Topics) von einer anderen Entität abonnieren. Diese Leafs sind gleichbedeutend zu bestimmten Themen die abonniert werden können. Durch diesen Schritt ist es möglich eine asynchrone Benachrichtigung zu erhalten .

Die Leafs für den Serienfeed sind die Serien, die abonniert werden können um Benachrichtigungen zu erhalten, sobald neue Nachrichten für eine bestimmte Serie vorhanden sind. Diese sind in Smack die LeafNodes, in denen die zu überbringende Nachricht enthalten ist. Um mehrere zusammenzufassen gibt es die CollectionNodes, die mehrere LeafNodes enthalten können.
Im Serienfeed war vorgesehen, die LeafNodes "Serien" in die CollectionNodes "Genres" zu verpacken. Zusätzlich zum Filter soll dies gewährleisten, dass der Benutzer Benachrichtigungen zu allen Serien eines Genres erhalten kann, wenn er diese abonniert hat. Da CollectionNodes die Möglichkeit bieten, wenn sie abonniert wurden, Benachrichtigungen für alle beinhalteten LeafNodes zu generieren.
Aus Zeitgründen fällt dies leider weg und es sind nur die einzelnen Leafs vorhanden.

6.3 Implementierung und Codebeispiele

Für die Implementierung des XMPP in diesem Abschnitt ist zusätzlich die Software Openfire notwendig, sowie die Smack API, welche die Erstellung eines XMPP Clienten sowie diverse XMPP-Funktionen zur Verfügung stellt.

Mit Openfire ist es möglich den eigenen Computer zum Server zu machen um die XMPP-Dienste zu testen. Man konfiguriert den Server und kann z.B. einen User erstellen, welcher später auf die GUI zugreifen kann. Zu beachten ist, dass der Openfire Server läuft sobald das Programm gestartet wird, da dieses sonst nicht ausgeführt werden kann.

```
private static ConnectionConfiguration config =
new ConnectionConfiguration("localhost", 5222);
    private static XMPPConnection connection = new XMPPConnection(config);
    private static PubSubManager mgr = new PubSubManager(connection);
    private String username;
    private String password;
       private ArrayList<PayloadItem<SimplePayload>> notifications;
    private ArrayList<LeafNode> serien = new ArrayList<>();
       private Listener listener = new Listener();
```

Zu erstellende Instanzen und Variablen

Zunächst wird eine Instanz der Klasse ConnectionConfiguration erstellt, welche die Eigenschaften der Server-Verbindung darstellt. Dazu werden zwei Parameter übergeben, die zusammen den Pfad zum Server ergeben, in dem Fall "localhost" und den Port "5222".
Danach wird eine Instanz von XMPPConnection "connection" erstellt und die vorher festgelegte Konfiguration übergeben.

Ein PubSubManager “mgr” wird erstellt, der für die Handhabung mit den Nodes verwendet wird und es werden diverse Variablen initialisiert wie eine Liste für die Nodes “serien” und für die “notifications” und Strings “username” und “password” für die Benutzerdaten.

```
public void connect(JTextField usernameTextfield, JPasswordField passwordTextfield) {
        this.usernameTextfield = usernameTextfield;
        this.passwordTextfield = passwordTextfield;
        try {
            username = usernameTextfield.getText();
            password = passwordTextfield.getText();
            connection.connect();
            connection.login(username, password);
            Roster roster = connection.getRoster();  // weiÃŸ nicht
            roster.setSubscriptionMode(SubscriptionMode.accept_all); // weiÃŸ nicht
            System.out.println("Hallo " + username + " Sie haben sich erfolgreich
verbunden");
            anAlleAltenAnhaengen(); // listener hinzufÃ¼gen an alle
        } catch (XMPPException ex) {
            System.out.println("Fehler beim Einloggen");
        }
    }
```

connect()-Methode

Die Methode connect() ist zum Aufbau der Verbindung zum Openfire Server, dabei werden die Parameter “usernameTextfield” und “passwordTextfield” übergeben, dessen Werte von der GUI übergeben werden. Der Benutzer gibt in das Fenster, das beim Start des Programms aufgerufen wird seine Daten (Username und Passwort) ein welche an die Funktion weitergeleitet werden. Mit den Daten wird die Verbindung und das Einloggen ausgeführt und man erhält eine Meldung für die erfolgreiche Anmeldung oder beim Scheitern eine Fehlermeldung. Zudem wird festgelegt, dass alle Subscription-Anfragen akzeptiert werden. Die Methode anAlleAltenAnhaengen() sorgt dafür, dass bei jedem Verbinden mit dem Server ein Listener an alle bereits verfügbaren Nodes hinzugefügt wird, damit die Benachrichtigungen auch im Nachhinein funktionieren.

Zum Trennen der Verbindung wird einfach die disconnect()-Methode verwendet

```
public void disconnect() {
        connection.disconnect();
    }
```

disconnect()-Methode

Eine der wichtigsten Funktionen für diesen Meilenstein ist das Erstellen der neuen LeafNodes, falls neue Serien hinzugefügt werden. Zunächst folgt eine Überprüfung, ob die Serie, die neu dazukommen soll bereits existiert, wenn nicht, wird eine Node angelegt.
Zusätzlich wird die Node konfiguriert und ihr werden Eigenschaften zugefügt, wie z.B. ob eine Abonnierung (form.setSubscribe(true);) möglich ist oder ob Benachrichtigungen verschickt werden können oder wie die Zugriffsrechte auf die Nodes sind.

```
public void createSerie(JTextField serienNameTextfield, JList serienList) {
        this.serienListe = serienList;
        this.serienNameTextfield = serienNameTextfield;
        serienName = serienNameTextfield.getText();
        try {
            mgr.getNode(serienName);
            System.err.println("Serie existiert bereits.");
        } catch (XMPPException e) {
        }try {
            if ("".equals(serienName)) {
                System.out.println("Bitte Seriennamen eingeben");
            } else {
                ConfigureForm form = new ConfigureForm(FormType.submit);
                form.setAccessModel(AccessModel.open);
                form.setPublishModel(PublishModel.open);
                form.setDeliverPayloads(true);
                form.setPersistentItems(true);
                form.setPresenceBasedDelivery(false);
                form.setSubscribe(true);
                LeafNode node = (LeafNode) mgr.createNode(serienName, form);
                serien.add(node);
                serienListe.setModel(refreshSerienList());
            }
        } catch (XMPPException e) {
            System.err.println("Fehler beim erstellen der Serie");
        }
    }
```

Erstellen einer Node für eine Serie

Nachdem die Node erstellt ist, kann der Benutzer diese abonnieren, dafür wird zunächst eine leere Node angelegt, worin die Serie gespeichert werden soll, falls es die schon geben sollte. Es wird der Serienname übergeben und die Node geholt mit dem dazugehörigen Namen. Im nächsten Schritt wird geprüft, ob bereits ein Abonnement dieses Leafs vorliegt, was über die Methode “isSerieSubscribed(serienname)” stattfindet.

Ist das nicht der Fall, so wird ein Listener hinzugefügt, damit der Benutzer später eine Benachrichtigung erhalten kann, wenn etwas neues gepublished wird.
Für das Abonnement ist die JID des Benutzers erforderlich, die geholt wird, bevor der Vorgang durchgeführt wird. Diese setzt sich zusammen aus dem Usernamen, dem ein “@”-Zeichen sowie die lokale Serveradresse angehängt wird. Ist nun kein Fehler aufgetreten wird die Serie abonniert .

```
public void serieSubscribe(String serienName) {
        LeafNode node = null;
        try {
            node = (LeafNode) mgr.getNode(serienName);
        } catch (XMPPException ex) {
            System.out.println("Fehler beim Holen der Serie");
        }

        if (isSerieSubscribed(serienName)) {
            System.out.println("Serie bereits abonniert");
        } else {
            try {
                node.addItemEventListener(listener);   // eventlistener hinzufÃ¼gen damti
man notifications erhÃ¤lt
                System.out.println("Listerner hinzugefÃ¼gt");
                node.subscribe(getJID());       // benutzer mit der jid abonniert die node
                System.out.println(serienName + " abonniert");
            } catch (Exception e) {
                System.out.println("Serie konnte nicht abonniert werden");
            }
        }
    }
```

Abonnieren einer Serie

Nun da der Benutzer die Serie abonniert hat, erwartet er, bei neuen News die zu einer Serie veröffentlicht wurden eine Benachrichtigung, etwas neues vorhanden ist. Dafür muss die Node gepublished werden. Der Inhalt der in dieser Benachrichtigung stehen soll wird über den Payload gelöst. Der Payload ist die neue Nachricht die dem Benutzer übertragen wird. Um etwas zu publishen wird eine Serie aus der vorhandenen Serienliste ausgewählt und dessen Node abgrufen. Der Payload, also die News kann dann über die GUI eingetragen werden und wird in den SimplePayload “p” gespeichert, welches daraufhin in das PayloadItem eingetragen wird. Dann wird die Funktion node.publish(item) aufgerufen, also wird die ausgewählte Node mit dem Item gepublished und der Benutzer erhält im direkten Anschluss eine Nachricht, dass etwas neues gepublished wurde.

```
public void publishSerie(JList serienList, JTextArea seriePayloadTextarea) {
        this.serienListe = serienList;
        this.seriePayloadTextarea = seriePayloadTextarea;
        LeafNode node;
        String payloadText;

        payloadText = seriePayloadTextarea.getText();
        if (serienListe.isSelectionEmpty() || "".equals(payloadText)) {
        System.out.println("Bitte eine Auswahl Treffen");
        } else {
            try {
                serienName = serienListe.getSelectedValue().toString();
                node = mgr.getNode(serienName);
                payloadText = "<publish>" + payloadText +"!!"+serienName+ "</publish>";
                SimplePayload p = new SimplePayload("root", "", payloadText);
                PayloadItem<SimplePayload> item = new PayloadItem<>(null, p);
                node.send(item);
                System.out.println("Es wurde gepublished ");
            } catch (XMPPException ex) {
                System.out.println("Fehler beim Publishen");
            }
        }
    }
```

Publishen einer Serie

Damit die Benachrichtigung auch beim Benutzer ankommt, sobald ein Item gepublished wird, muss ein EventListener eingesetzt werden, der eine Aktion durchführt sobald es zum publishen kommt.
Diese Methode ist sehr provisorisch und nimmt sich das Item und liest den Inhalt aus, in dem der gesamte String ausgelesen wird und die gewünschte Stelle, der Inhalt, ausgeschnitten wird. Genauso wird auch der Name der Node ausgelesen, da rechtzeitig keine andere Methode implementiert werden konnte.
Diese Nachricht wird mit der POST-Methode aus dem REST-Service in die FeedXML eingespeichert und erscheint dann als News dort.

```
        @Override
        public void handlePublishedItems(ItemPublishEvent<Item> event) {
            System.out.println("Neues Item ist da!");
            String xml =event.getItems().get(0).toString();
            int x = xml.indexOf("\">");
            int y = xml.indexOf("</");
            int z = xml.indexOf("!!");
            String nachricht = xml.substring(x+2,z);
```

```
            String serien_Name = xml.substring(z+2,y);
            News n = new News();

            n.setTitel(nachricht);
            n.setSerienname(serien_Name);
            try {
                nachrichtenService.postNachricht(n);
                nachrichtenListe.setModel(refreshNachrichtenList());
            } catch (    JAXBException | FileNotFoundException ex) {
                System.out.println("Gehler beim posten der Nachricht");
            }
        }
    }
```

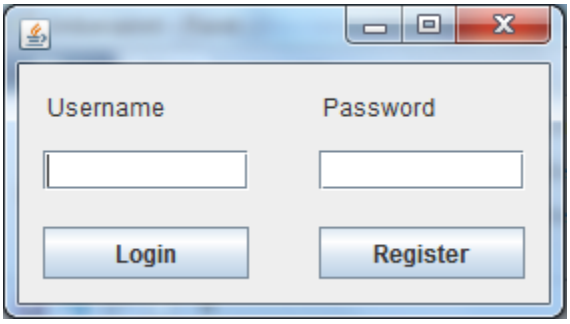
Methode um Inhalt eines gepublishten Items zu erhalten

Nun da der XMPP-Dienst eingerichtet wurde, kommt der letzte Meilenstein und zwar das Erstellen der grafischen Benutzeroberfläche und die Implementierung der Funktionalitäten.

7. Der Client

In der finalen Phase des Projekts gilt es die beiden einzelnen Komponenten RESTful Webservices und die XMPP-Dienste zu vereinen und in einer GUI für den Benutzer ansteuerbar zu machen.
Für die Erstellung wird Java Swing benutzt und die Oberfläche mit Hilfe vom Swing GUI von Netbeans umgesetzt.

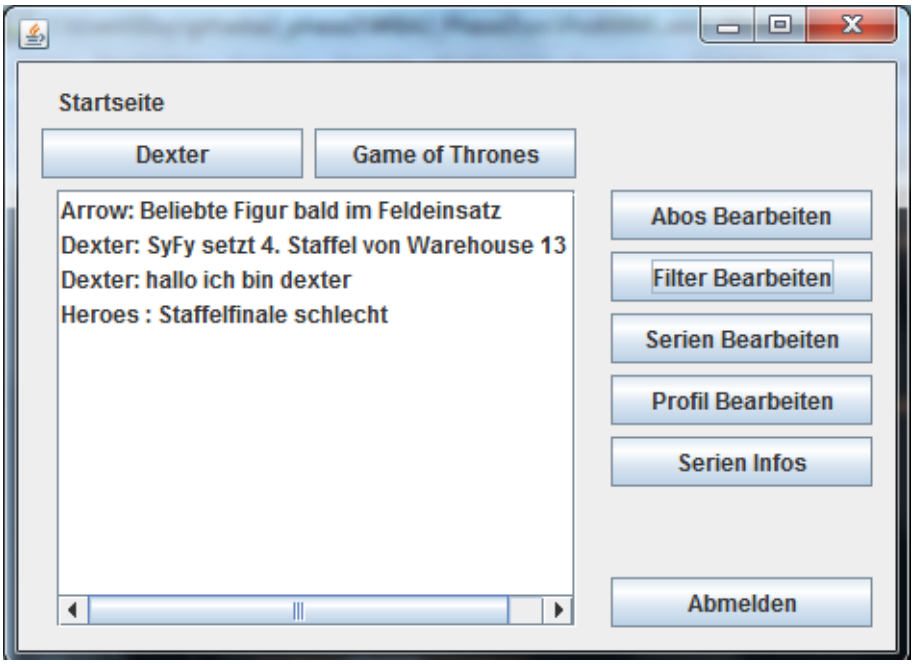
Um das gesamte Programm zu starten muss im Vorfeld wieder der Openfire Server gestartet werden um das Anmelden und den gesamten XMPP-Dienst zu ermöglichen.



Login-Fenster

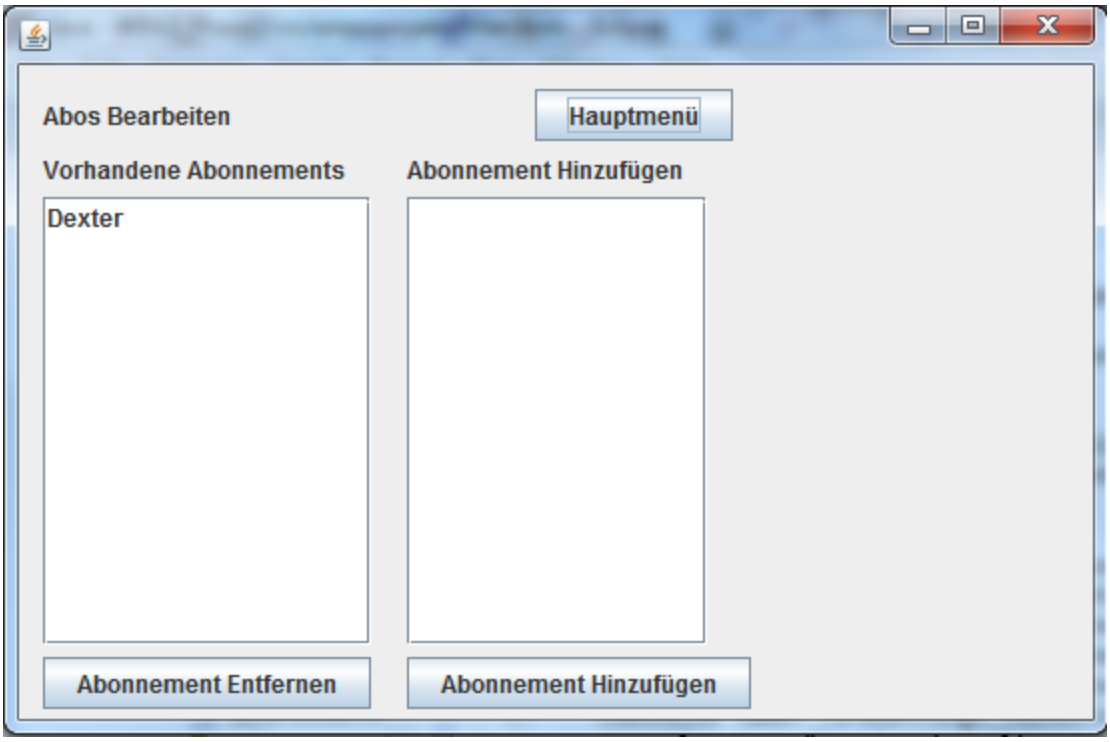
Ist dies geschehen und das Programm gestartet, erscheint das erste Fenster welches das Login-Fenster ist und man wird aufgefordert seine Benutzerdaten, bestehend aus dem Usernamen und dem Passwort, einzugeben. Nach erfolgreicher Eingabe folgt eine Weiterleitung auf das Startfenster, wo die Filter aufgelistet sind sowie alle aktuellen News dazu. Bevor jedoch ein Filter ausgewählt ist, werden alle

aktuellen News zu allen Serien angezeigt. Will der Benutzer nun Nachrichten zu einem bestimmten Filter erhalten, wählt er diesen aus und die News werden im Anschluss gefiltert.



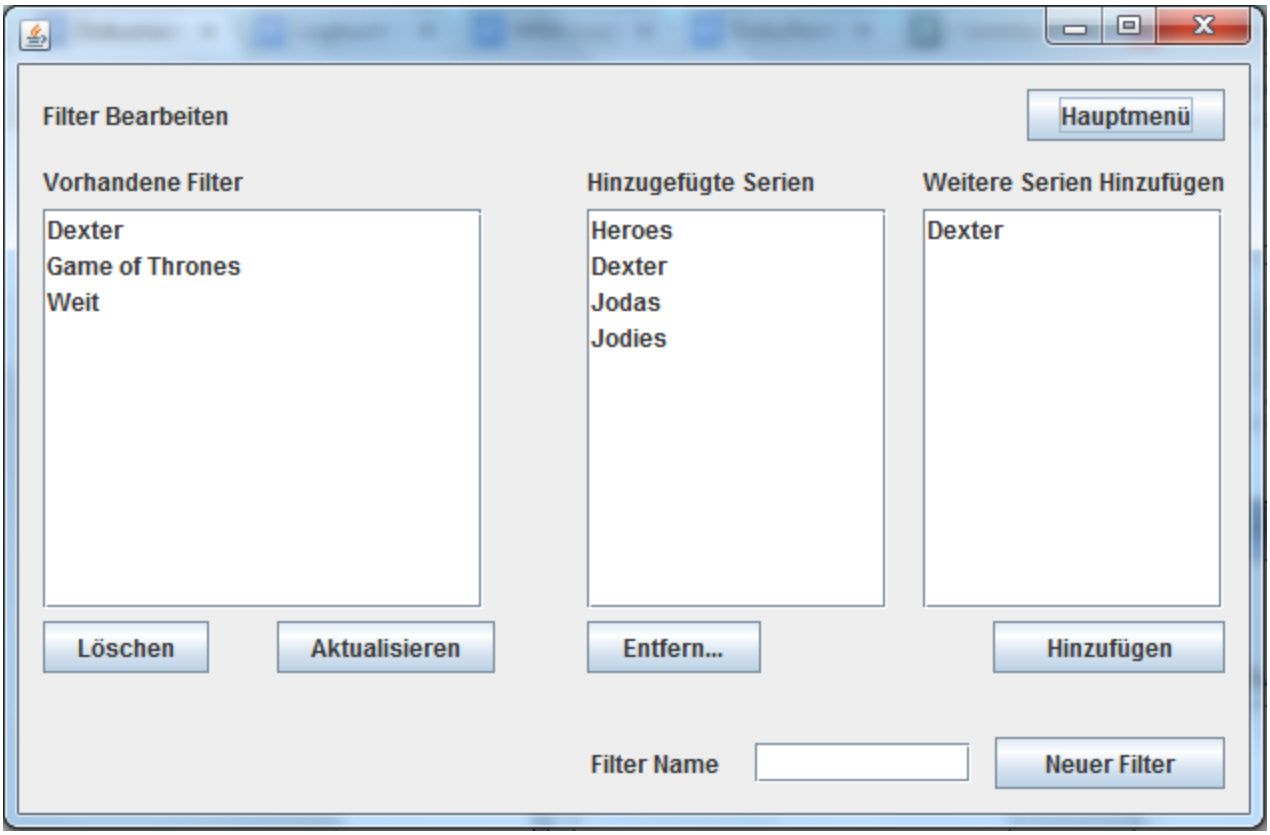
Startseite

Auf der rechten Seite des Startfensters sind alle relevanten Buttons vorhanden, die einen zu den weiteren Fenstern leiten sollen, welche wären: Abos/Serien/Filter/Profil bearbeiten, Serien Infos sowie der Abmelde-Button.
Klickt man auf den “Abos bearbeiten”-Button, so erscheint ein Fenster mit zwei Listen, einmal die bereits vorhanden Abonnements und einmal verfügbare Serien die noch nicht abonniert wurden. Hier kann man nach Belieben neue Serien abonnieren oder deabonnieren. Über den Hauptmenü-Button kehrt man wieder zurück zum Startfenster.



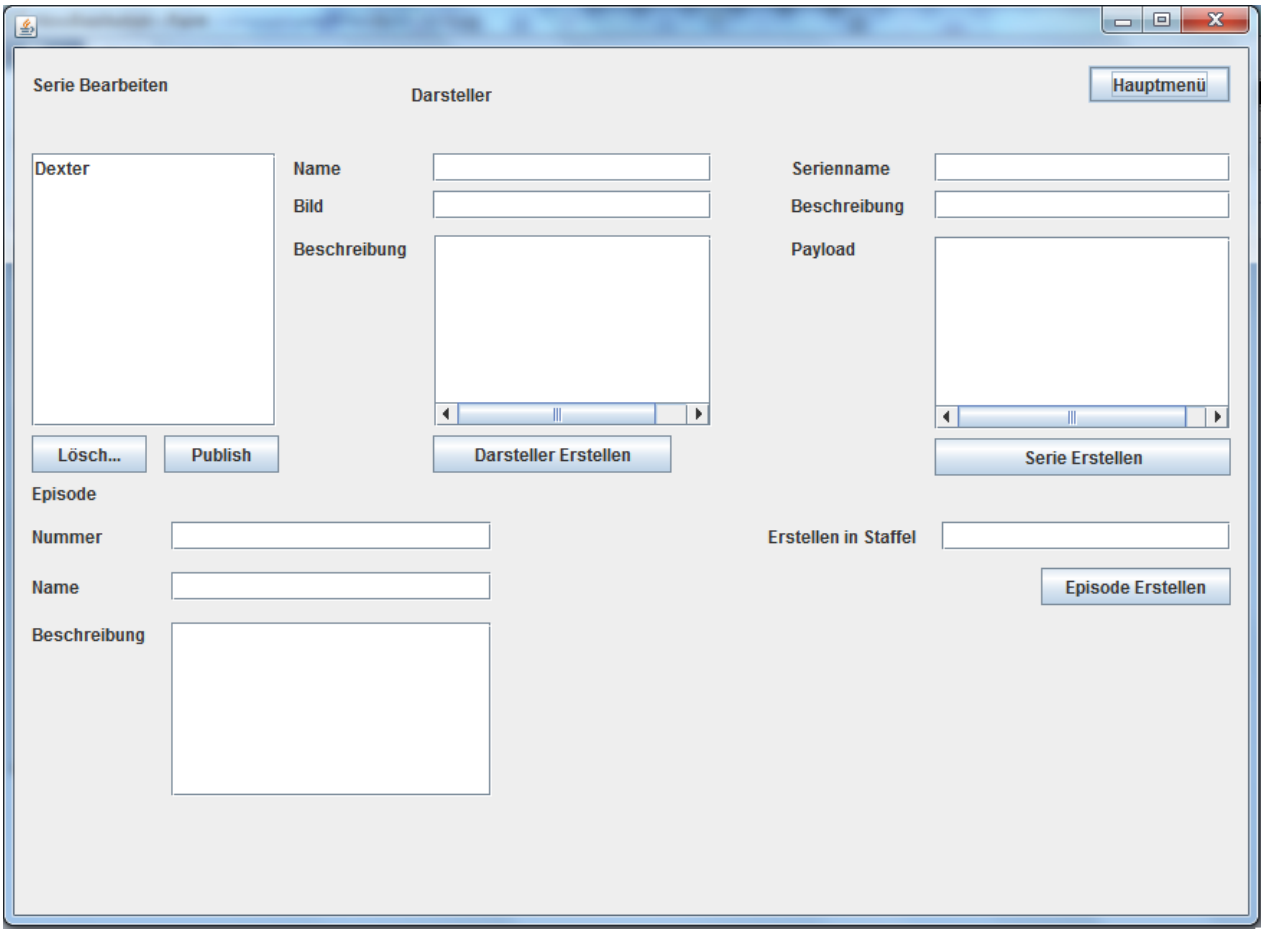
"Abos bearbeiten" - Fenster

Das "Filter bearbeiten" Menü sieht ähnlich aus wie das "Abos bearbeiten" Menü. Man hat die Liste aller erstellten Filter, den Inhalt der einzelnen Filter in der Mitte, die nach Auswahl eines bestimmten angezeigt werden und ganz rechts Serien, die noch zu einem hinzugefügt werden können. Will der Benutzer einen neuen Filter anlegen, so gibt er in das Textfeld "Filter Name" einen Namen an und klickt dann auf "Neuer Filter" und dieser wird dann ebenfalls links in der Liste angezeigt. Um einen Filter auszuwählen und dessen Inhalt angezeigt zu bekommen, muss einer ausgewählt werden und auf "Aktualisieren" geklickt werden um die Ansicht zu aktualisieren.



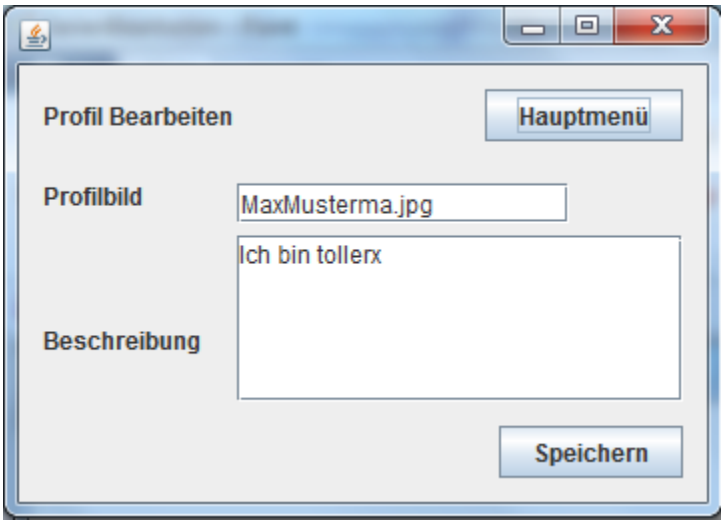
"Filter bearbeiten"-Fenster

Will man nun eine Serie bearbeiten, wählt man diesen Menüpunkt und gelangt in größeres Fenster wo der Inhalt, der zu einer Serie gehört (Darsteller, Episoden und Staffeln), modifiziert werden kann.
Hier ist auch die wichtigste Funktionalität im Hinblick auf XMPP und zwar das publishen einer neuen News. Dafür wird eine Serie ausgewählt und im Feld "Payload" rechts, die Titel der Nachricht eingegeben, die neu erstellt werden soll. Wenn man dann auf "Publish" klickt, kommt eine Ausgabe in der Konsole, dass dies geschehen ist und man erhält auch eine Benachrichtigung. Geht man danach wieder in das Hauptmenü, erscheint die zuvor gepublishte News im Startfenster.



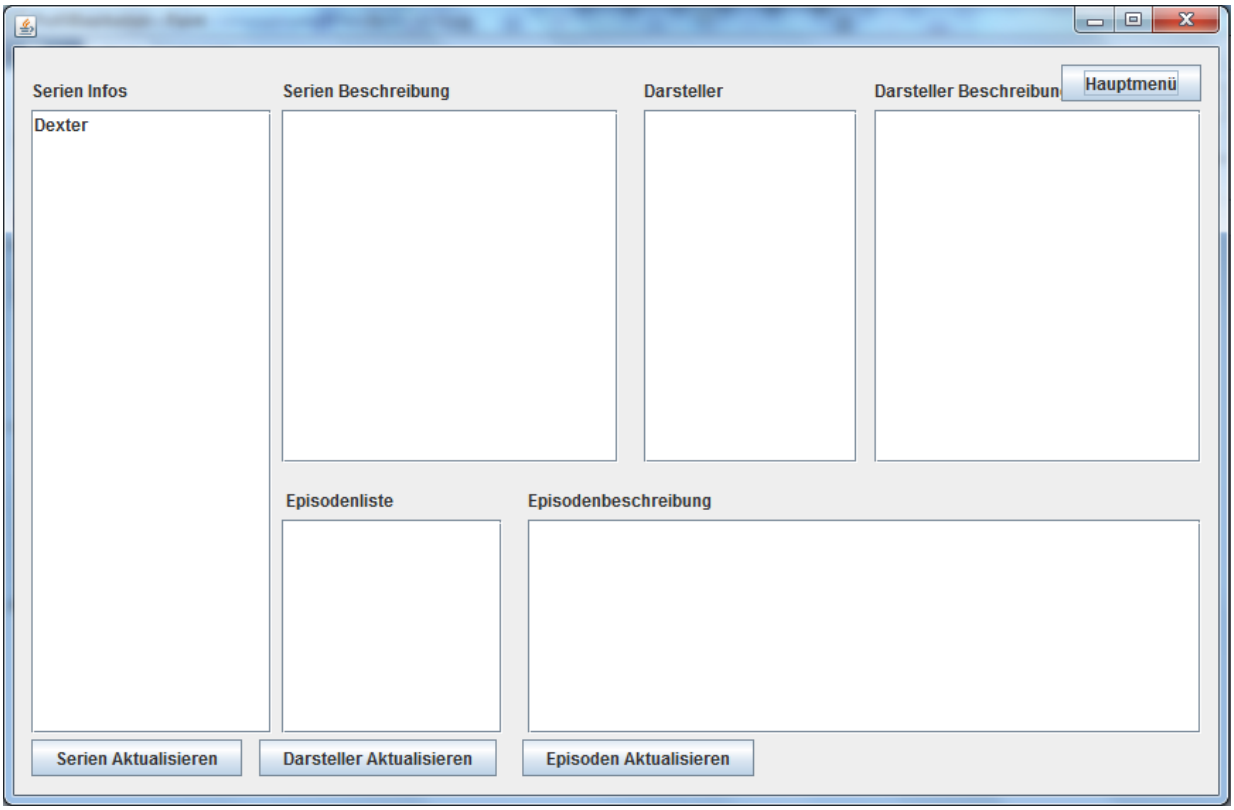
"Serie bearbeiten" - Fenster

Das Fenster "Profil bearbeiten" beschränkt sich darauf, dass der Benutzer einen Link zu einem Profilbild eingeben kann, sowie eine kurze Beschreibung.



"Profil bearbeiten" - Fenster

Um Informationen zu einer Serie abzurufen, geht man in das “Serien Infos” Menü. Ganz links stehen alle verfügbaren Serien zur Auswahl und wenn eine gewählt und auf “Serien Aktualisieren” geklickt ist, erscheint die Serienbeschreibung, die Episodenliste und Darsteller. Der Benutzer kann nach dem gleichen Verfahren dann auch die Informationen zu einzelnen Episoden und Darstellern abrufen.



“Serien Infos” - Fenster

Was bewusst vernachlässigt wurde ist die Funktion News bearbeiten und Kommentare zu einzelnen Artikeln zu schreiben. Aufgrund von Zeitmangel, ist diese Implementierung weggefallen, jedoch ist das nur ein inhaltlicher Aspekt und die Hauptfunktion des Serienfeeds ist dadurch auch nicht beeinträchtigt worden. Außerdem wurde die Funktion einen neuen Benutzer zu registrieren entfernt, da es Probleme mit der Implementierung gegeben hat und daraufhin entschieden wurde das Szenario nur mit einem User abzuspielen.

8. Fazit

8.1 Ungelöste Probleme

Nachdem das Projekt soweit beendet wurde, dass es zur Abgabe bereit ist, sind noch einige Probleme geblieben, die nicht gelöst werden konnten. Die Filteranzeige auf der Startseite z.B. sollte über Tabs ansteuerbar sein, doch bei der Implementierung haben sich Probleme ergeben, so dass die Tabs am Ende Buttons geworden sind. Beispielhaft sind deswegen immer nur zwei Filter abgebildet, um zu verdeutlichen wie das Prinzip ist.
Eine Kleinigkeit ist zudem, dass wenn Filter gelöscht werden, die Serien nicht mit gelöscht werden, aber in der Realität werden Serien eigentlich nicht gelöscht, daher ist das ein kleineres Problem.
Des Weiteren können Serien doppelt zu Filtern hinzugefügt, was aber sinnlos ist, da eine Serie nur einmal in einem Filter zu sein braucht.
Die GUI ist außerdem aus gestalterischer Sicht nicht sehr ansprechend und ist sehr überladen. Die Priorität lag eher auf der Funktionalität als auf dem Design, dieses wurde auf das notwendigste reduziert. Darunter leidet auch die intuitive Bedienung der Oberfläche.
Außerdem konnten einige Fehler, z.B. fehlerhafte Eingaben nicht komplett abgefangen werden und die Daten, die auf den Servern können Probleme beim Aufruf auf einem anderen System verursachen, wenn nicht der gleiche Datensatz vorhanden ist.

8.2 Rückblick und Ausblick

Rückblickend auf die Workshop-Phase des Faches Webbasierte Anwendungen 2: Verteilte Systeme bleibt zu sagen, dass es bei der Bearbeitung einige Schwierigkeiten gab. Der Umfang war an sich angemessen und zu Anfang noch gut zu bearbeiten, doch ab dem Meilenstein mit dem REST-Service wurde es zeitlich ziemlich knapp. Die Betreuung der Tutoren war gut doch an manchen Stellen gibt es noch Verbesserungsbedarf. Bei REST hatten wir z.B. das Problem, dass wir zum Meilenstein zwar die Methoden implementiert hatten, doch waren diese fehlerhaft und wir mussten selber später feststellen, dass vieles überarbeitet werden muss. An dieser Stelle hätten wir gerne vorher mehr Unterstützung durch die Tutoren gehabt, da wir den bereits vorhandenen straffen Zeitplan kaum eingehalten konnten und zum Ende hin der Arbeitsaufwand aufgrund der Verbesserungen die durchgeführt werden mussten sehr hoch war.
Außerdem war das fehlende Vorwissen manchmal sehr problematisch, da wir uns erst komplett in alle Themen einlesen mussten bevor wir mit der Umsetzung beginnen konnten. Das ist zwar kein Problem, doch da nebenbei andere Fächer ebenfalls Zeit erfordern, war dies ein erschwerendes Hindernis.
Dadurch ist der Code an einigen Stellen etwas unaufgeräumt und es fehlen inhaltlich einige Funktionalitäten, doch die hauptsächliche Funktion, die die Anwendung erfüllen soll ist soweit erreicht worden.

Zusammenfassend kann also gesagt werden, dass für zukünftige Jahrgänge die Vorbereitung in der Vorlesung oder im Praktikum etwas besser gemacht werden kann und dass in der Zeitplanung eventuell was verändert werden kann, um die Belastung der Studenten etwas geringer zu halten.

9. Quellenverzeichnis

1. XMPP - The Definitive Guide, Peter Saint-Andre, Kevin Smith & Remko Troncon
2. REST und HTTP, 2. aktualisierte und erweiterte Auflage, Stefan Tilkov
3. http://docs.oracle.com/cd/E19226-01/820-7627/gipxf/index.html Produces Annotation
4. http://docs.oracle.com/cd/E19226-01/820-7627/gipyt/index.html Consumes Annotation
5. http://www.w3.org/International/questions/qa-when-xmllang.de.php xml:lang in XML-Schemas
6. http://xmpp.org/rfcs/rfc3920.html XMPP
7. http://www.igniterealtime.org/builds/smack/docs/latest/javadoc/ Smack Dokumentation
8. http://www.torsten-horn.de/techdocs/jee-rest.htm#JaxRsHelloWorld-Grizzly REST-Beispiele, Torsten Horn
9. http://www.se.uni-hannover.de/pages/de:tutorials_restful_guestbook#xjc Beispiel für RESTful Webservices anhand eines Gästebuchs
10. http://www.w3.org/TR/webarch/#id-resources, Architecture of the World Wide Web, Volume One, 21.06.2013
11. http://openbook.galileocomputing.de/java7/1507_13_002.html#dodtp231820a7-1c06-4e4c-9a34-dd977e4975c8, Java 7 - Mehr als eine Insel, Christian Ullenboom
12. http://www.igniterealtime.org/builds/smack/docs/latest/documentation/extensions/pubsub.html, Pubsub Documentation
13. http://www.igniterealtime.org/builds/smack/docs/latest/documentation/extensions/disco.html , Service Discovery
14. http://xmpp.org/extensions/xep-0060.html, XEP-0060: Publish-Subscribe
15. http://xmpp.org/about-xmpp/history/, XMPP History