Fachhochschule Köln
Cologne University of Applied Sciences

Web-basierte Anwendungen 2: Verteilte Systeme - Phase 2

Projektdokumentation über ein verteiltes System
zum Abonnieren von TV-Serien
um über deren Episodenaustrahlung informiert zu werden

Dozent: Prof. Dr. Kristian Fischer
Fachhochschule Köln

Betreuer: Renée Schulz
David Bellingroth
Christopher Messner
Fachhochschule Köln

ausgearbeitet von

Dennis Meyer, Matrikelnr. 11084479
Dominik Schilling, Matrikelnr. 11081691

Sommersemester 2013

Inhaltsverzeichnis

1 Einleitung

Im Rahmen der zweiten Projektphase des Moduls Web-basierte Anwendungen 2: Verteilte Systeme, geht es um die Entwicklung einer Applikation, die sich mit dem Datenaustausch in verteilten Systemen beschäftigt. Innerhalb dieses Systems soll eine synchrone Kommunikation mit Hilfe des REST Konzeptes, sowie ein asynchroner Datenaustausch auf Grundlage von XMPP realisiert werden. Zur Repräsentation der Funktionalität dient ein selbst entwickelter Client.

Folgende Dokumentation erfasst die einzelnen Entwicklungsschritte. Von der anfänglichen Konzeptidee inklusive Konzeption der benötigten XML Schemata, über die Implementierung des RESTful Webservices und XMPP Client, bis hin zur Umsetzung des Clients. Dabei werden, neben den finalen Ergebnissen, auch getroffene Abwägungen, Entwicklungen und Entscheidung hinsichtlich der Umsetzung dargestellt und erläutert.
Zuletzt folgt eine Projektreflektion, anhand derer Erfahrungen für weitere Projekte mitgenommen werden sollen.

2 Konzeption

2.1 Die Idee

Zu Beginn des Projektes ging es um das Finden eines geeigneten Anwendungsfalls aus der Realität. Für diesen sollte das geforderte System, eine sinnvolle Ergänzung für mögliche Anwender darstellen.
Zum einen muss die Möglichkeit bestehen, Informationen direkt auszutauschen und zum anderen sollte eine Art Benachrichtigungssystem existieren, bei denen die Interessenten nur beim Eintreten bestimmter Ereignisse informiert werden müssen.
Nach Überlegung fiel dabei die Wahl auf das Thema Film und Fernsehen und wurde im Spezielleren auf Serien eingegrenzt. Sinnvoll erscheint eine Umsetzung dieses Thema aufgrund der regelmäßigen Ausstrahlung von Folgen einer bestimmten Serie.
Während man für einen Film lediglich einmal die Ausstrahlungszeit in Erfahrung bringen muss, würde sich dieser Schritt bei Serieninteressierten Woche für Woche wiederholen. Dazu kommen kurzfristige Änderung im Fernsehkalender, die im unpassenden Fall dazu führen, dass eventuell Folgen verpasst werden. Schaut jemand nicht nur eine Serie, sondern hat eine Art festen Wochenplan, so kann hierbei auch der Überblick darüber verloren gehen, welche Folge man eigentlich zuletzt gesehen hat oder wie die Serie hieß, die einem neuliche empfohlen wurde.

Mit dieser grundlegenden Überlegung ging es an das Konzipieren des Funktionsumfangs des Systems unter dem Projekttitel **Serientracker**.
Die Idee ist, dass Serieninteressierte[1] die Möglichkeit besitzen bestimmte Serien zu favorisieren und Meldungen zu gewissen Themen abonnieren können. Er soll Zugriff auf einen Pool von Serien bekommen, die auf einem Server gespeichert und verwaltet werden. Der Benutzer erhält dann zu seinen Lieblingsserien Benachrichtigungen, sobald eine Episode dieser Serie im TV ausgestrahlt wird. Wenn vorher noch das Genre Comedy abonniert wurde, so erhält er zudem noch eine Benachrichtigung, welche Serien dieses Typs aktuell so laufen.
Außerdem soll die Möglichkeit bestehen eine Episode zu bewerten und als gesehen/ungesehen zu markieren. Diese Verwaltung findet innherhalb von privat angelegten Listen statt, die neben der klassischen Seen/Unseen Form auch als Watchlist oder ähnliches definiert werden kann.

[1] Innerhalb des Kontext wird für den Benutzertyp des Anwenders/Nutzers auch diese Bezeichnung verwendet werden, da diese Personengruppe in ihrer Funktion die Benutzer des Systems darstellen werden.

Die **Server-Anwendung** soll die Nutzer über die TV-Austrahlung einer Episode zu einem Zeitraum informieren, die sich der Benutzer selbst definiert hat. Er kann demnach entscheiden, ob ihm eine Benachrichtigung für die ganze Woche genügt, ob er jeden Morgen über den aktuellen Tag informiert werden möchte oder eine Meldung 5 Minuten vor Sendestart ausreicht, da er planmäßig zu Hause sein wird.

Ein **Content-Admin** soll erweiterte Rechte bekommen, um die Content-Verwaltung zu übernehmen. Die Anwendung soll das Anlegen, Bearbeiten und Löschen von Serien bzw Episoden ermöglichen. Zudem ist somit das Korrigieren von Fehlern möglich, die von Usern eingeschickt werden oder die aufgrund von Planänderungen anfallen.

Als möglicher Zusatz ist eine Einbindung von Freunden geplant. Nutzer sollen sich gegenseitig hinzufügen/abonnieren können um sich gegenseitig zu benachrichtigen. Zum Beispiel in Form von *Freund X schaut gerade Y*, *Freund Z hat Serie/Episode mit 8,0 bewertet* oder *Freund Y empfiehlt dir Serie W*. Ob dieser Ansatz innerhalb des Projektes realisiert wird, hängt vom vorranschreiten der Umsetzung und der damit einhergehenden Abwägung für das eigentlich Ziel des Projektes ab.

2.2 Informationsaustausch

Zuvor genannte Funktionen würden, für einen Informationsaustausch zwischen Server und Anwendung, hinsichtlich folgender Einstufung der Datenübertragung umgesetzt werden.

2.2.1 Synchrone Datenübertragung

Zum einen hat der Anwender direkt die Möglichkeit auf Informationen in Form von Daten zuzugreifen und diese zu Manipulieren.

- Serien-Interessierte
  - Markieren von Episoden
    - Gesehen/Nicht gesehen
  - Bewertung einer Episode
    - Kommentar
    - Bewertung in Zahlen
  - Fehlermeldung
    - geänderte Sendezeit, fehlerhaftes Datum
  - Listen
    - Ausgabe (Un)Watched
    - Ausgabe vorhandene Serien

        * Ausgabe Follower/Following
    - Favorisierung
        * Anlegen
        * Löschen
        * Bearbeiten
            · Zeitpunkt der Benachrichtigung
- Content-Admin
    - Verwaltung der Episoden
        * Anlegen
        * Löschen
        * Bearbeiten

2.2.2 Asynchrone Datenübertragung

Ein weiterer Aspekt ist das Anfordern von Informationen, wobei die entsprechenden Informationen von Seiten des Servers von Bedingungen abhängig gesendet werden, was auch mehrfach geschehen kann.

- Serien-Interessierte
    - Benachrichtung bei TV-Austrahlung
    - Freunde mit gleicher Favorisierung bei Serienstart mit Check-in benachrichtigen
        * Freund X schaut auch W

      Empfehlung einer Serie von Freund(e) anzeigen
- Content-Admin
    - Benachrichtung bei Fehlermeldung durch User

2.2.3 Kommunikationsabläufe

Im Rahmen des Konzeptes wurden zwischen dem Server und dem User (hier repräsentativ für die Anforderung seitens der Anwendung) folgende Kommunikationsabläufe identifiziert. Die synchronen Interaktionen werden vom User iniziiert und greifen auf die beim Server abgelegten Datensätze zu. Dabei hat jeder User die Möglichkeit eine einfache Anzeige der Daten anzufordern oder eine Serie den Favoriten hinzuzufügen. Usern mit erweiterten Adminrechten wird hierbei auch die Manipulation der Daten gewährleistet, um eine Verwaltung zu ermöglichen. Die entsprechende Repräsentation der Information wird vom Server wieder an den User zurückgegeben.

Diese Ansätze werden über einen RESTful Webservice umgesetzt und mit entsprechenden GET, POST, PUT und DELETE Methoden realisiert.

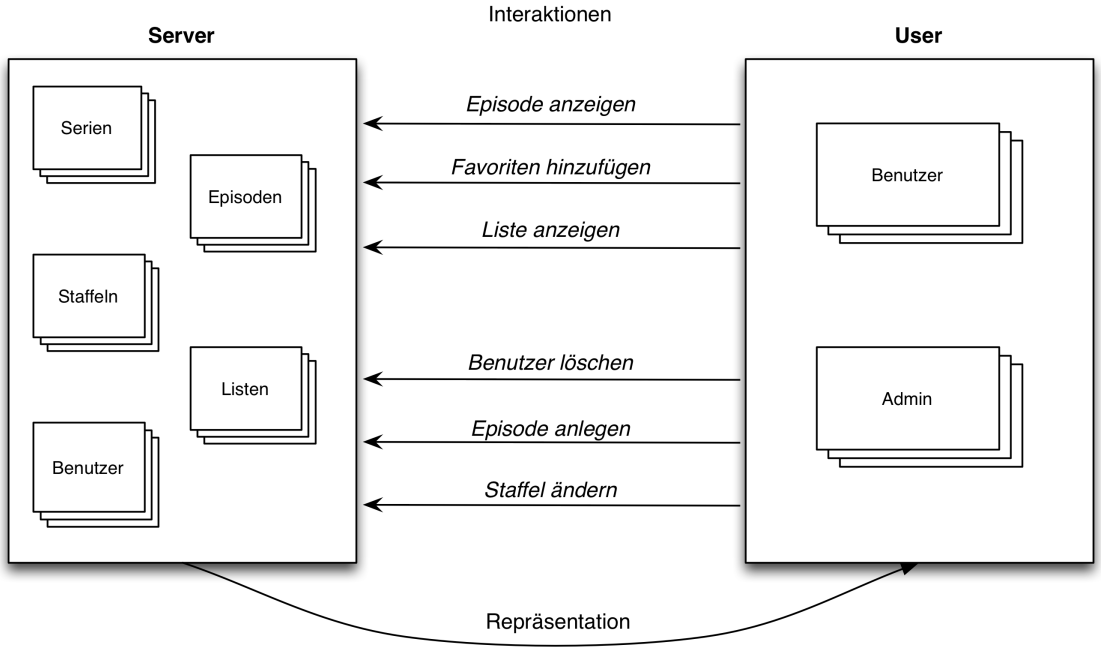


Abbildung 2.1: Synchrone Kommunikationsabläufe

Die Asynchronen Kommunikationsabläufe liefern Informationen bei bestimmten Events vom Server an die entsprechenden User als Empfänger. Dieser Ansatz wird mit Hilfe von XMPP realisiert werden und ist im Kapitel 3.4 genauer dargelegt.

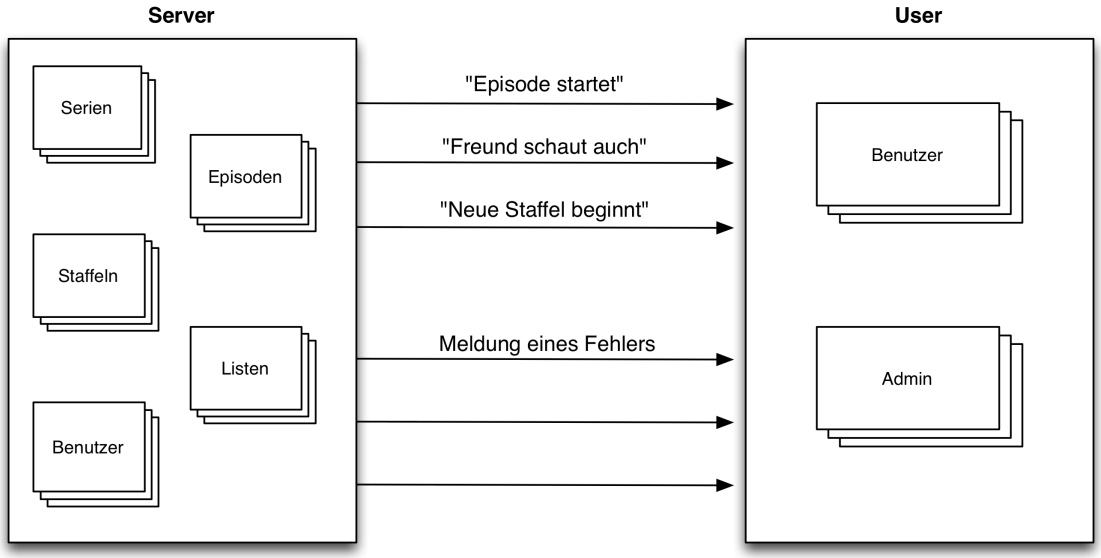


Abbildung 2.2: Asynchrone Kommunikationsabläufe

3 Entwicklung des Projektes

3.1 Projektbezogenes XML Schemata

3.1.1 Vertiefung

Der erste Meilenstein befasst sich mit der Repräsentation von Daten in XML.

Damit bei der Verwendung der XML Dateien bei der späteren Verarbeitung mit JAXB keine Probleme auftreten, ist es notwendig eine Validierung der Dateien durch Definition zugehöriger XML Schemas durchzuführen. Vorteil bei der Verwendung eines Schemas ist, neben der Kontrolle auf Wohlgeformtheit und der Verwendung definierter Datentypen und Strukturen, auch das festlegen von Restriktionen.

Hinsichtlich des zugrunde liegenden Konzeptes und den benötigten Informationen, gibt es viele Elemente innerhalb der Dateien, die nur mit Strings realisiert werden können. Das Problem bei freier Definition besteht darin, dass die Datensätze sehr fehleranfällig sind, wenn es um die Benutzung durch Menschen geht. Rechtschreibfehler beim Namen des Landes, des Fernsehsenders oder des Genres, würden für den Benutzer der Anwendung noch kein Problem darstellen, da er vermutlich deuten könnte was gemeint ist. Für die Umsetzung ist es aber von Vorteil, die Datensätze möglichst konsistent zu halten.

Das System des Serientrackers beruht darauf, die Verwaltung von Serien anhand von Listen zu ermöglichen. Wie bereits in der Besprechung der Umsetzung erwähnt, wird anhand dieser Informationen auch die asynchrone Datenübertragung realisiert.
Das Abonnieren von Informationen zu laufenden Serien des Genre Action, greift bei Benachrichtigung auf Datensätze zu, die diesem Elementwert unter dieser eindeutigen Zeichenfolge zugeordnet sind. Formulierungsfehler wie *Aktion*, die von einer einheitlichen Schreibweise abweichen, führen dementsprechend zu Komplikationen, weil sie die Konsistenz der Informationssätze beschädigen.

Hinsichtlich des Themas Serien, führten die Vorüberlegungen zu dem Entschluss, dass folgende Objekttypen innerhalb des Serientrackers von interesse sind und wiederkehrende Elemente darstellen:

Serie
Eines der wichtigsten Elemente, dass alle Informationen beinhaltet, die für den Benutzer von bedeutung sind, ist die Serie. Eine Serie besitzt allgemeine Informationen wie Name, eine Beschreibung, der Sender auf dem sie ausgestrahlt wird oder das Produktionsland. Im realen Kontext wird eine Serie zudem in Seasons (Staffeln) ausgestrahlt, die jeweils eine bestimmte Anzahl von Episoden enthalten.

Season
Bestandteil einer Serie, von der es im Laufe der Jahre immer neue Objekte gibt und die nach einer gewissen Anzahl an Episoden als abgeschlossen gelten.

Episode
Ein Kernelement des Systems, dass ein wichtigen Typ für die asynchrone Kommunikation darstellt. Durch das Abonnieren von Genres oder Serien, wird die Benachrichtigung in Bezug auf eine einzelne Episode ausgelöst, die sich durch ihr jeweiliges Austrahlungsdatum und -zeitpunkt kennzeichnet. Zudem kann der Inhalt der Episode von Interesse sein.

User
Neben den Serieninformation, gibt es die Benutzer des Serientrackers, die sich anmelden und durch Listen die Dienste des Serientrackers abonnieren. Allgemein werden hierbei personenbezogene Daten wie Username und echter Name erwartet, so wie Zusatzinformationen, die für andere Benutzer von Interesse sein könnten und die Person hinter dem Profil genauer beschreiben. Beispiele wären das Alter, ein Profilbild, Wohnort oder eine kurze Beschreibung.

Liste
Die Liste kann als Sammlung mehrerer Serien zu einem bestimmten Thema aufgefasst werden. Hierbei gibt zum einen Listentypen die definitiv vorhanden sein müssen, wie beispielsweise Genrelisten oder die Favoritenliste eines Benutzers. Diese erfasst die Serien, zu denen der Eigentümer Benachrichtigungen erhalten will. Zum anderen besteht aber auch die Möglichkeit, dass der Benutzer sich Listen anlegt, die seinen individuellen Wünschen entsprechen.

Message
Als zusätzlicher Typ, der jedoch erst in der asynchronen Kommunikation verwendung finden wird, wurde die Message identifiziert. Hierbei handelt es sich um eine bestimmte Nachricht, die den entsprechenden Usern bei Eintreten eines Events zugeschickt wird und ein neues Ereignis meldet.

3.1.2 Realisierung der Schemata

Nach der theoretischen Planung fand die Realisierung der XML Schemata statt. Dabei wurde für jeden der zuvor identifizierten Obertypen ein eigenes Schema definiert und die einzelnen Elemente/Attribute mit Datentypen und Restriktionen belegt. Die Aufteilung der einzelnen Typen auf ein seperates Schema, fand mit den Gedanken statt, die Struktur der Daten möglichst einfach und lesbar zu halten. Eine Serie die mehrere Staffeln enthält, die wiederrum jeweils eine Menge von Episoden auffassen, würde ein sehr komplexes Element definieren, dass während der Entwicklung schnell zu unübersichtlich werden kann.

Da die Daten einer Episode, inhaltlich jedoch weiterhin davon abhängen, von welcher Serie diese ist und in welcher Staffel sie vorkam, wird eine Referenzierung mit Hilfe von global eindeutigen IDs eingeführt. Jede Serie, User, Season, Episode, Message und Liste wird mit einer einzigartigen Folge von Zeichen beschrieben, über diese es möglich ist auf gewünschte Informationen zuzugreifen und entsprechende Elemente auszulesen.

Dieses Prinzip lässt sich am Beispiel des XML Schemas einer Episode veranschaulichen. Hierbei handelt es sich um einen Codeauszug, der die gesamte Struktur einer Episode vorgibt:

```
1  <xs:element name="episode">
2    <xs:complexType>
3      <xs:choice minOccurs="0">
4        <xs:sequence>
5          <xs:element ref="episodeNumber"/>
6          <xs:element ref="title"/>
7          <xs:element ref="overview"/>
8          <xs:element ref="airdate"/>
9          <xs:element ref="images"/>
10       </xs:sequence>
11     </xs:choice>
12
13     <xs:attribute ref="serieID" use="required"/>
14     <xs:attribute ref="seasonID" use="required"/>
15     <xs:attribute ref="episodeID" use="required"/>
16   </xs:complexType>
17 </xs:element>
```

Code 3.1: Definition des complexElement Episode mit Elementen und Attributen

Eine Episode bekommt damit eine eindeutige **episodeID** zugeordnet und ist, gesehen unter allen Episoden, eindeutig identifiziert. Zudem erhält jede Episode auch die Referenz der serienID und seasonID als Attribute, um Verweise und Zuordnungen zu realisieren. Da sich jedes Objekt demnach durch eine Kennung repräsentiert, reicht bei anlegen von Listen und Containern ein einfacher Verweis auf entsprechendes Element, wodurch Datenredundanz bei mehreren Schemata verhindert wird, die inhaltlich voneinander abhängen. Innerhalb der XML Datei einer Serie, reicht somit der Verweis auf die einzelnen Seasons über ihre ID, ohne dass deren Inhalte mehrfach abgespeichert werden müssen. Dieser Redundanzverlust wäre auch realisierbar gewesen, wäre eine Season und Episode direkt innerhalb des Serieschemas gespeichert worden. Damit hätte eine Referenz auf die Serie genügt und auf eine Episode könnte mit einer Abfrage der Seasonnummer und Episodennummer getätigt werden. Der Entschluss zur Aufteilung der Schemata führte jedoch zwangsläufig zu dieser Entscheidung, wobei die Folge daraus auch komfortable Vorzüge in der Flexibilität der einzelnen Elemente mit sich bringt.
Um die Struktur des gesamten Serien Elements zu ermöglichen und vorhandene IDs global nutzen zu können, wird jedes Schema über eine Masterdatei in die einzelnen XML Schemas inkludiert, um bereits definierte Elemente und Attribute wiederverwendbar zu machen und die Datenmenge zu reduzieren.

```
1  <!-- Dieses Schema inkludiert alle zu verwendenen Schemata-->
2
3  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
4
5    <xs:include schemaLocation="Series.xsd"/>
6    <xs:include schemaLocation="Serie.xsd"/>
7    <xs:include schemaLocation="Seasons.xsd"/>
8    <xs:include schemaLocation="Season.xsd"/>
9    <xs:include schemaLocation="Episodes.xsd"/>
10   <xs:include schemaLocation="Episode.xsd"/>
11
12   <xs:include schemaLocation="Lists.xsd"/>
13   <xs:include schemaLocation="List.xsd"/>
14   <xs:include schemaLocation="Users.xsd"/>
15   <xs:include schemaLocation="User.xsd"/>
16   <xs:include schemaLocation="Message.xsd"/>
17
18   <!-- Globale Elemente -->
```

Code 3.2: Auszug aus der Masterinklude Serientracker.xsd

Neben dem Include der einzelnen Schemata, werden auch vereinzelte Elemente darin definiert, die innerhalb der einzelnen Schemata wiederholt Verwendung finden. Hierbei vor allem die einzelnen ID Attribute.

Während der Entwicklung dieser Variante, gab es anfängliche Schwierigkeiten innerhalb der Umsetzung. Der Verweis innerhalb eines XML Schemas auf ein Anderes und die entsprechende Verwendung der Elemente und Attribute, führte zu Komplikationen innerhalb der Deklarationen. Auch eine Definition eines einheitlichen Namespaces führte zu keiner Lösung. Im zweiten Ansatz folgte dann die direkte Auseinandersetzung zwischen den Optionen Include und Import, wobei die letztendliche Wahl auf die Inkludierung fiel. Ein Import erlaubt den Verweis auf unterschiedliche Namespaces, wobei ein Include auf einen einheitlichen Namespace verweist und bei nichtbestehen, den des Oberschemas übernimmt.
Da Namespaces in der Regel dazu dienen, Elementen und Attributen einen gewissen Namensraum zu gewährleisten, der bei gängigen Bezeichnungen wie *Name* zu Konflikten führen kann, ist die Definition bei größeren Projekten durchaus zu empfehlen. Ob dabei ein Einzelner oder Mehrere angelegt werden sollten, ist eventuell auch abhängig von der gesamten Struktur des Schemas. Da im Rahmen dieses Projektes jedoch mit relativ geringem Datenumfang gearbeitet wird und mit keiner Komplikation der Bezeichnungen zu rechnen ist, wurde auf eine genaue targetNamespace Definition verzichtet. Auch wenn bestimmte Elementbezeichnungen innerhalb eines *http://Serientracker.de* Namensraum denkbar gewesen wäre. Letztendlich spielt dieser Aspekt für die Umsetzung innerhalb des Projektes jedoch keine tragenden Rolle. Daher wurde auf die Definition verzichtet und die entsprechende Include Variante verwendet.

Ein weiterer Punkt der in der Entwicklungsphase von Bedeutung war, war die Frage danach, wie die Daten innerhalb des lokalen Speichers abgelegt werden. Durch die Definition einzelner Schemas erhält jedes Objekt dieses Typs auch eine eigene XML Datei. Im späteren Verlauf könnten sich daraus aber Probleme beim gezielten Zugriff entwickeln, denen frühzeitig Abhilfe geschaffen werden sollte.
Aus diesem Grund wurde eine weitere Gruppe von Elementen innerhalb XML angelegt, die Containerklassen.
Für jeden Elementtyp, der als eigene Entität angelegt wird, wurde eine Containerklasse definiert, innerhalb derer beliebig viele Objekte eines bestimmten Typs aufgenommen werden können. Dieses Objekt dient in der letztendlichen Verwaltung als Sammlung aller Elemente.

Folgendes Beispiel veranschaulicht den Ansatz und wurde in dieser Form für jeden Elementtyp angelegt:

```
<!-- Element das alle Serien aufnimmt -->
  <xs:element name="series">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="serie" minOccurs="0" maxOccurs="
           unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

Code 3.3: Auszug aus der Series.xsd Definition

Um die vorhandenen Daten semantisch sinnvoll und reichhaltig anzulegen, wurde bei der Entwicklung der XML Schemata darauf geachtet, diesen Anspruch durch die verschiedenen Datentypen, Restriktionen und Benutzungstypen zu gewährleisten.

3.1.3 Datentypen

Grundsätzlich werden innerhalb des Serientrackers Daten unterschiedlichen Typs benutzt, die entsprechend des realen Kontextes am sinnvollsten erscheinen. Aufgrund der Komplexität, soll hierbei nicht auf jedes einzelne Element eingegangen werden, sondern nur ein Überblick darüber vermittelt werden, mit welchen Grundideen vorgegangen wurde.

Neben den einfachen Standartdatentypen wie String, Boolean, Integer, Time (...), bietet XML die Möglichkeit weitere spezifische Typen wie anyURI oder ID zu definieren.
Da innerhalb des Serientrackers viele Information lediglich in Textform sinnvoll sind, wurde für diese der Typ **String** verwendet. Beispielhaft seien hier die Titel und Beschreibungen, sowie Benutzernamen und Länder erwähnt.
Jahreszahlen, Episodenlänge und Season- und Episodennummern werden mit Zahlen als **Integer** ausgedrückt. Bestimmte Zeitangaben wie Ausstrahlungszeit, bei denen es nur um den Zeitpunkt geht, in der einfach Form **Time** und genauere Angaben, wie bei Ausstrahlungstag in Hinsicht auf die Benachrichtigung, über den **dateTime** Typ. Linkverweise auf Bildquellen wie bei Avatar erhielten den Typ **anyURI**.

Einzelne Elemente wie User und List, weisen hierbei noch eine Besonderheit auf. Wie bereits bei den Kommunikationsabläufen innerhalb des Konzepts angesprochen, wird die Anwendung von Benutzern verwendet, die verschiedenen Rechtetypen angehören.

Aus diesem Grund wird jeder User mit dem Attribute Admin gekennzeichnet, der innerhalb einer einfachen **Boolvariablen** die entsprechenden Rechte festgelegt. Ähnliches Prinzip wird bei den Listen verwenden zur Unterscheidung, ob die Sichtbarkeit für jeden Benutzer gestattet ist oder nur für den Besitzer der Liste.

Zur angesprochenen Referenzierung und Identifizierung einzelner Entitäten, zeichnen sie sich durch eine eindeutige ID als Attribut aus. Generell wurden die Informationen wie ID oder Rechte, die ein Objekt nicht inhaltlich sondern eher aus verwaltender Sicht genauer beschreiben, als Attribute definiert. Inhaltliche Informationen die Bestandeteil des Objektes sind, werden hingegen als Elemente angelegt.

Für den Datentyp einer solchen ID, wurde ein eigenes Element folgendermaßen definiert:

```
1  <xs:simpleType name="idType">
2    <xs:restriction base="xs:string">
3      <xs:pattern value="|(ss|sn|ep|us|ls|me)_[0-9a-z]{8}"/>
4    </xs:restriction>
5  </xs:simpleType>
```

Code 3.4: Definition der globalen IDs

Am Anfang der Zeichenketten steht ein Kürzel, dass den jeweiligen Typ angibt. SN steht dabei für Season, EP für Episode und ähnliches. Danach folgt ein optisches Trennzeichen, gefolgt von einer beliebigen Characterfolge aus 8 gemischten Zeichen mit Zahlen und Buchstaben. Eine Serie erhält damit eine Zuordnung der Form *ss_0a1b2c3d*.

3.1.4 Restriktionen

Damit die Informationen innerhalb der XML Dateien inhaltlich sinnvoll sind und während der Verarbeitung keine weiteren Probleme entstehen (zum Beispiel durch unterschiedliche Schreibweisen), wurden für einige Elemente Restriktionen definiert.
Dabei ist das Ziel, die Fehleranfälligkeit zu reduzieren und im Kontext logische Informationen zu gewährleisten. Vorhandene Restriktionen, also Einschränkungen/Bedingungen, die für einzelne Elemente getroffen wurden, sind im folgenden mit einer kurzen Begründung aufgeführt. Häufig treten diese in Form von Längenbegrenzungen bei Texten auf oder als Grenzbereichen bei Zahlen.
Weitere Elemente kennzeichnen sich entsprechend dadurch, dass der Inhalt auf eine bestimmte Auswahl an Möglichkeiten eingeschränkt wurde. Beim Konzipieren dieser Grenzen und Auswahlmöglichkeiten, ist es nicht möglich die perfekte Variante zu erzielen, sondern eine Auswahl von Optionen zu treffen, die im Kontext als zuverlässig und praktikabel erscheinen.

Manche Begrenzugnen wie Anzahl an Episoden einer Staffel oder Episodenlaufzeit, wurden in Bezug auf die Realität getroffen und angelehnt an gängige Formen. Prinzipiell wären hierbei alternative Auswahlmöglichkeiten und Varianten denkbar, wie beispielsweise die freie Definition des Genres oder des TV Senders als einfacher String. Hierbei hat der verwaltende Admin dann die Option den Inhalt frei zu bestimmen. Letztendlich wurde jedoch der eigentliche Nutzen abgewägt, sodass für den Serientracker folgende Restriktionen definiert wurden:

Tabelle 3.1: Allgemeine Restriktionen

Element/Attribut	Restriktion	Begründung
Overview (global)	Stringlänge $10 <$ und < 500	allgemeinen Informationen, kurze Inhaltsangabe
Title (global)	Stringlänge $1 <$ und < 80	gängige Titellänge
Name (list)	Stringlänge $2 <$ und < 80	treffende Bezeichnung, Name keine Beschreibung
Public (list)	Boolean True und False	feste Zustände
Episodenumber (episode)	Anzahl < 26	sinnvolle maximale Episodenanzahl
Seasonnumber (seasons)	Anzahl < 41	sinnvolle Begrenzung, Freiraum für Langzeitserien

Tabelle 3.2: Restriktionen des User Schemas

Element	Restriktion	Begründung
Username	Stringlänge 2 $<$ und $<$ 30	sinnvolle Namenlänge, verhindert Text
Lastname	Stringlänge 1 $<$ und $<$ 40	gängige Nachnamenlänge, eventuell Doppelnamen, verhindert Text
Firstname	Stringlänge 1 $<$ und $<$ 50	gängige Vornamenlänge, Mehrfachnahmen
Gender	Auswahl zwischen Male und Female	logische Auswahl, Vorgabe verhindert Schreibfehler
Age	älter als 13 und jünger als 121	Mindestalter zur Nutzung, logische Obergrenze
Location	Stringlänge $<$ 40	Stadtname, Land etc. Eingabe ist keine Adresse und lässt sich in Kürze ausdrücken
About	Stringlänge $<$ 200	optionale Kurzbeschreibung, nach oben begrenzt, zu viele Informationen nicht unbedingt von Interesse
Admin	Boolean ob True or False	Rechtevergabe nach Status, Auswahl nur in 2 Zuständen möglich

Tabelle 3.3: Restriktionen des Serie Schemas

Element	Restriktion	Begründung
Year	Jahreszahl 1900 < und < 2015	Jahreszeiten außerhalb unrelevant
Country	Auswahlmöglichkeit Ländern	Eingabefehler verhindern
Episoderuntime	Auswahl zwischen gängigen Episodenlängen	Serie hat feste Episodenlänge
Network	Auswahl bekannter Sender	unrelevante Sender entfallen, Eingabefehler verhindern
Airday	Auswahl des Tagnamen	Eingabefehler verhindern
Genre	Auswahl definierter Genres	einheitliche Schreibweise, Eingabefehler verhindern, sinnvolle Genre

3.1.5 Beispieldaten

Bereits während der Entwicklung der XML Schemas, wurden parallel XML Dateien mit entsprechenden Beispieldaten definiert. Durch den entsprechenden Validierungstest bietet sich zum einen die Möglichkeit entsprechenden Definitionen auf Korrektheit zu prüfen und zum anderen XML Typen zu entwickeln, die möglichst praxistaugliche Elemente und Attribute aufweisen. Zu jedem XML Schema wurde daher mindestens eine Beispieldatei erzeugt und die entsprechende Referenzierung untereinander getest. Speziell im Containerelemente Series wurde deutlich, wie flexibel die Datensicherung mit globalen IDs sein kann. Zum einen reicht das Einbinden eines Objekts in der simplen Form <serie serieID="ss_6127hdja"/>, zum anderen kann auch die gesamte Struktur einer Serie inklusive Informationen über Season und die einzelnen Episoden eingebunden werden.

Für die entsprechende Nutzung in der weiteren Entwicklung, wurden zudem korrekte Beispieldatensätze von Serien angelegt. Diese Repräsentieren von der obersten Ebene der allgemeinen Serieninformationen bis hin zur niedrigsten Ebene der einzelnen Episodeninfor-

mation vollständige Datensätze, wie sie in einem komplexen System dieser Form angelegt werden müssten. Aufgrund von Testzwecken, wurde sich jedoch auf wenige Beispielserien beschränkt. Zudem wurden entsprechende Daten innerhalb der Series.xml definiert und nicht in die entsprechenden Typen aufgeteilt. Diese Variante hatte zum Vorteil, dass die Informationen einer Serie (für Menschen) übersichtlich strukturiert und lesbarer waren, als wenn jede Beispielepisode per Hand in einzelne XML Files getrennt worden wäre. Aufgrund der Vielzahl von Informationen zeigte, sich aber bereits bei wenigen Daten der Nachteil, dass die Datei sehr komplex wurde. Darüber hinaus weist diese Variante auch die Schwäche beim möglichen Datenverlust auf. Sollte in diesem Fall das sammelnde Element verloren gehen, so wäre der Datenverlust größer gegenüber den seperaten Absicherungen in einzelnen Files.

In der Konzeptvorstellung wurde von der möglichen Einbindung einer Freundefunktion gesprochen. Die Umsetzung dieses Themas würde noch eine Änderung der XML Schemata der User mit sich ziehen. Ähnlich wie für die Abonnements von Serien, könnte man innerhalb des User Schemas ein komplexes Element *Friends* einbauen, dass beliebig viele User aufnimmt. Dabei könnten normale Objekte des bereits vorhandenen Userschemas eingebunden werden und eine Liste aller Freunde bilden.
Um die entsprechende Kommunikation zwischen den einzelnen Freunden zu gewährleisten und Meldungen wie Empfehlungen oder Benachrichtigungen zu verschicken, muss weiterhin das Message Schema erweitert oder alternativ ein neues Schema der *Friendmessage* eingeführt werden. Eine seperate Trennung beider Nachrichtenarten wäre in sofern sinnvoll, als das diese in der Funktionalität und Definition unterschiedliche Anwendungen haben. Die bisherigen Nachrichten werden von der Serverseite aus an User geschickt und beinhalten statische Nachrichten. Auch Empfehlungen könnte man mit festen Standartnachrichten wie *X empfiehlt dir Serie Y* umsetzen, jedoch wäre hierbei in der Regel eine persönlichere Komponente wie Userkommentare mit enthalten.

Sofern dieser Aspekt umgesetzt wird, müsste man hierbei die entsprechenden Varianten abwägen und dabei mögliche Szenarien konzeptionell durchspielen, um eine passende Einbindung hinsichtlich der Funktionalität zu finden.
Nach der Auseinandersetzung mit den vorhandenen Daten und entsprechender Definition der XML Schemas, folgt im nächsten Schritt die Entwicklung der synchronen Kommunikationskomponenten.

3.2 Ressourcen und die Semantik der HTTP-Operationen

3.2.1 Vertiefung

Als Vorbereitung für die Umsetzung der synchronen Kommunikationsvorgänge, steht die theoretische Auseinandersetzung mit REST im Mittelpunkt. Der erste Schwerpunkt dabei ist die Identifizierung vorhanderer Ressourcen des Serientrackers. Bereits beim Konzipieren der XML Schemas mit beispielhaften XML Datensätzen, musste überlegt werden, für welche Elemente es möglich ist Entitäten der realen Welt zu ermitteln.

Bei einer Ressource geht es, ähnlich wie bei XML Dateien, nicht darum wie die darin enthalten Informationen im letztendlichen Kontext repräsentiert werden, sondern welche Informationen diese enthalten. Entsprechende Objekte der Außenwelt werden beschrieben und wie die Wurzelelemente bei XML, stellen sie einen bestimmten Objekttyp dar. Eine identifizierte Ressource, ist eine Schnittstelle zur Außenwelt und sollte daher dem Kontext entsprechend gut durchdacht werden.
Die Auseinandersetzung mit den XML Schemas lieferte dabei einen Überblick über vorhandene Primärressourcen. Dabei handelt es sich um die Oberklassen der vorhandenen Objekttypen *Serie* und *User*.

Desweiteren ist es möglich vorhandene Subressourcen zu identifizieren, die sich dadurch auszeichnen, dass sie selbst Bestandteil einer Ressource sind. Aufgrund der komplexen Struktur einer Serie, die neben den allgemeinen Informationen noch die Informationen zu mehreren Staffeln und entsprechenden Episoden enthalten, wurde früh die systematische Aufteilung festgelegt. Da es auch innerhalb der Anwendung von Interesse sein kann, eine einfache Repräsentation der Staffelübersicht oder der Episodenübersicht einer Staffel zu ermöglichen, bietet es sich gerade bei diesen Typen an, diese Objekte als eigene Ressource zu designen. Dazu kommt, dass eine Episode zum Beispiel im Kontext einer Serie am meisten Sinn macht, durchaus aber auch für sich existieren kann. Zur Ordnung der einzelnen Elemente, gibt es entsprechende Listenressourcen wie Series, Seasons, Users und Episodes, welche alle Elemente des zugehörigen Typs aufnehmen und sammeln.

Ein Kernelement der Anwendung wird das Benachrichtigen der Benutzer über bestimmte Ereignisse sein. Diese Abonnements lassen sich in Listen verwalten. Gerade bei einer möglichen Kategorisierung wie beispielsweise *Serie mit dem Genre Drama*, wäre es durchaus interessant gewesen Listenressourcen mit entsprechenden Filtern zu definieren. Aufgrund der Vielzahl von Kategorisierungsmöglichkeiten, wird dieser Schritt aber nicht über einzelne Ressourcen stattfinden, sondern innerhalb der Anwendung durch die Abfrage bestimmter Elemente erzielt.

Die theoretische Auseinandersetzung führte zu folgenden Ressourcen, wobei die angegeben URI nur den charakteristischen Abschnitt widerspiegelt. Da für die Umsetzung an sich, die URI eher als eine ID in Form von Zeichen steht und für die letzendliche Auswertung keine semantische Korrektheit notwendig ist, wird auf eine Darstellung in Form mit Schema und Pfadangabe in der Übersicht verzichtet.

Tabelle 3.4: Ressourcen des Serientrackers

Ressource	URI	Methode
Liste aller Serien	/series/	GET, POST
Einzelne Serie	/series/{id}	GET, PUT, DELETE
Liste aller Staffeln	/seasons/	GET, POST
Einzelne Staffel	/seasons/{id}	GET, PUT, DELETE
Liste aller Episoden	/episodes/	GET, POST
Einzelne Episode	/episodes/{id}	GET, PUT, DELETE
Liste aller User	/users/	GET, POST
Einzelne User	/users/{id}	GET, PUT, DELETE
Liste aller Listen	/lists/	GET, POST
Liste eines Users	/lists/{id}	GET, PUT, DELETE

Jedes Element, dass für die synchrone Kommunikation von Interesse ist, besitzt eine Ressource als einzelnes Element
(Form:/ressourcenname/{id}) und die Gesamtheit einer Liste (Form:/ressourcenname/). Bei den URI wird der Zugriff über entsprechende ID's auf die Liste deutlich und es zeigt sich eine Folge der einfachen URI. Durch die Konzipierung der Untertypen Staffel und Episode als einzelne Ressource, ist eine entsprechender Aufruf auf das bestimmte Element möglich. Vorstellbar aufgrund der allgemeinen Struktur wäre auch ein Pfad der
Form /series/{id}/seasons/{id}/episodes/{id}. Dieser setzt entsprechend vorraus, dass nicht nur die einfache Episoden ID, sondern auch die zugehörige Serien ID und Season ID bekannt ist.

3.2.2 Semantik der HTTP-Operationen

Nachdem die Ressourcen identifiziert sind, muss nun bestimmt werden, welche Operationen (HTTP-Methoden) unterstützt werden und welche Semantik sich dahinter verbirgt.

Dafür stehen die vier Methoden GET (Informationen auslesen), POST (Informationen anlegen), PUT (Informationen ändern) und DELETE (Informationen löschen) zur Verfügung. Nicht jede Ressource muss alle Methoden bereitstellen. Für das Projekt wurden die Ressourcen erneut betrachtet und die benötigten Methoden samt Semantik herausgearbeitet. Dabei entwickelte sich folgende Ergebnis:

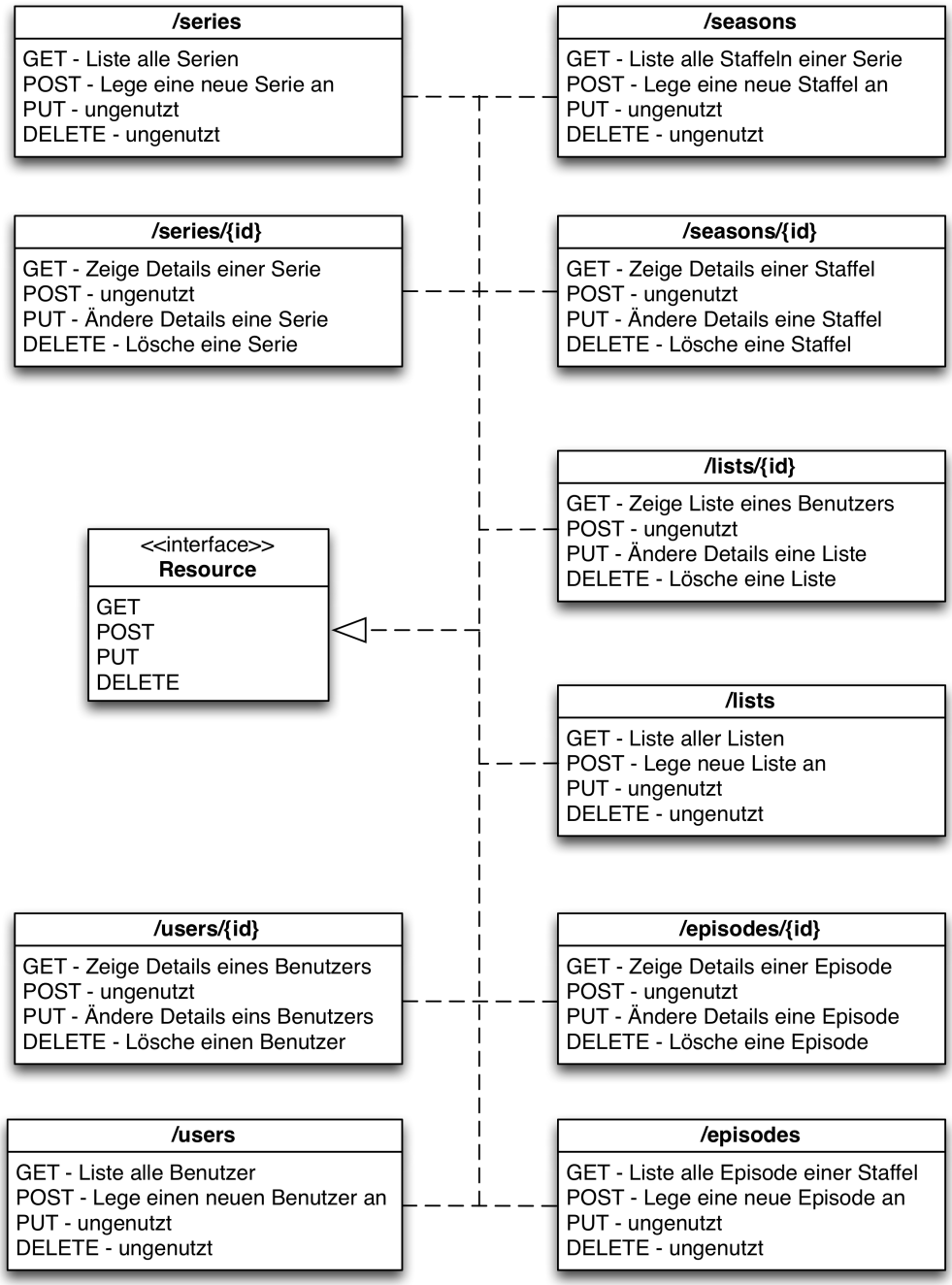


Abbildung 3.1: Bedeutung der http Methoden

3.3 RESTful Webservice

3.3.1 Vertiefung

Nach der konzeptionellen Planung der Ressourcen und HTTP - Operationen, geht es um die codebasierte Umsetzung. Definierte XML Schemas werden nach erfolgreicher Validierung in JAXB Objekte überführt, um eine Verwendung des Datenbestandes im Javaprojekt zu ermöglichen und aus den Schemadateien Java-Klassen zu erzeugen.
Zur Realisierung der synchronen Kommunikation wird Grizzly als Webserver der HTTP - Operationen verwendet. Zusätzlich wird zur Entwicklung des RESTful Webservices das Jersey framework implementiert, dass für JAX-RS APIs und Server als Referenzimplementierung dient.
Bereits im Rahmen der Ressourcenfindung, wurde sich Gedanken darüber gemacht, wie URI zur jeweiligen Ressource gestaltet werden können, damit ein logischer Request auf gewünschte Daten stattfinden kann.
Die Repräsentation einer spezifischen Ressource wurde im vorherigen Abschnitt mittels **Path-Parameter** realisiert, das heißt, der Parameter ID war Teil der eigentlichen URI.
Als Alternative können allerdings auch sogenannte **Query-Parameter** zum Einsatz kommen. Diese eignen sich meistens um Ressourcen noch weiter zu verfeinern und sind in der Regel optional.

Innerhalb des Serientrackers hat man es grundlegend mit 2 Typen von Daten zu tun, die bereits in vorherigen Überlegungen angesprochen wurden.Zum einen haben wir feste Objekte wie Serien, Season oder Episoden, die eine eindeutige ID erhalten und sich über diese auffinden lassen. Für diese Typen eignet sich besonders die Path-Parameter Anfrage. Diese Zeichnen sich durch die einfach Angabe des Pfades aus wie beispielsweise **@Path("serieID")** für die GET Operation auf die URI */series/{serieID}*.
Der zweite Typ sind die jeweiligen Listen, die sich durch die Elemente und Attribute der Serien entsprechend kategorisieren lassen. Eine Konzeptidee war das Abonnieren bestimmter Genres. Da Listen aber generell Serien unterschiedlichen Genres aufnehmen können, prinzipiell aber auf dem selben XML Schema *list* beruhen, wurde nach einer Möglichkeit entsprechende Objekte bei einem Request herauszufiltern.
Die Lösung dieses Problem wurde dann über eine Erweiterung der XSD, sowie der Abfrage per Query-Parameter innerhalb des ListsService erzielt.

```
1 @Path( "/lists" )
2 public class ListsService {
3
4 @GET
5   @Produces( MediaType.APPLICATION_XML )
6   public Response getGenreList(
7                 @QueryParam( "type" ) String type,
8                 @QueryParam( "name" ) String name)
9   [...]
```

Code 3.5: Auszug aus ListsService mit QueryParam

Als zusätzliche Parameter dienen hierbei *type* zur Abfrage, ob es sich um einer Userliste oder Genreliste handelt und der entsprechende *name* der Liste, falls es eine Genreliste ist. Genrelisten weisen hierbei nur Serien eines bestimmten Typs auf und werden speziell für diesen Zweck angelegt. Eine Abfrage auf diesen Typ erhält die Form /lists/?type=genre&name=action und würde bei erfolgreichem Request die entsprechende Liste mit den Serien vom Genre Action zurückgeben.
Weiterhin könnte exemplarisch ein Query-Parameter in diesem Projekt bei der Ressource Users zum Einsatz kommen, um nur die Benutzer der Gruppe Admin zu repräsentieren: /users/?group=admins. Aber auch bei einem Zugriff auf eine einzelnen Episode einer Staffel einer Serie können Query-Parameter genutzt werden, Beispiel: /episodes/?serie_id=1&season_id=2. Jedoch wären die Parameter an dieser Stelle obligatorisch.
Als Alternative zu den beiden Varianten bietet sich noch der **HTTP-Header** des Clienten an. Da diese Art von Parameterübertragung eher untypisch ist, wird an dieser Stelle nicht weiter drauf eingegangen.

Da jeder Request auch immer einen Response erfordert, ist es bei der Umsetzung notwendig, Statuscodes der HTTP - Operationen zu benutzen. Dabei viel die Wahl auf folgende Statuscodes bei zugehörigen Ereignissen.

Tabelle 3.5: Statuscodes der Webservice Ressourcen

Statuscode	Bedeutung
400	Bad Request - Request konnte aufgrund der Syntax nicht verstanden werden
404	Not Found - kein passender Treffer in der gefragten Ressource-URI
409	Conflict - Request aufgrund aktuellen Status nicht ausführbar
500	Internal Server Error - unerwartete Umstand, Request nicht durchgeführt

Wie diese Codes innerhalb der einzelnen Response-Methoden Verwendung finden, wird im folgenden an Beispielen dargestellt. Nach einigen Vorüberlegungen folgt nun die Umsetzung in Java und das finale Ergebnis.

3.3.2 Umsetzung

Aufbauend auf die zuvor identifizierten URI, ging es nun darum die HTTP Methoden auf die einzelnen Ressourcen umzusetzen. Aus struktureller Sicht wurde dabei für jede Ressource eine eigene Service-Klasse angelegt, welche die jeweiligen Operationen bereitstellt. Innerhalb einer solchen, werden angedachte URI Pfade der Tiefe nach implementiert. Das Prinzip verdeutlicht folgende Übersicht zum Klassenaufbau des SeriesService und SeasonService:

```
1  /**
2   * Service for:
3   * GET      /series
4   * POST     /series
5   * GET      /series/{serieID}
6   * DELETE   /series/{serieID}
7   * PUT      /series/{serieID}
8   * GET      /series/{serieID}/seaons
9   * GET      /series/{serieID}/seaons/{seaonID}
10  * GET      /series/{serieID}/seaons/{seaonID}/episodes
11  * GET      /series/{serieID}/seaons/{seaonID}/episodes/{
        episodeID}
12  */
13 @Path( "/series" )
14 public class SeriesService {}
15
```

```
16 /**
17  * Service for:
18  * GET      /seasons
19  * POST     /seasons
20  * GET      /seasons/{seasonID}
21  * DELETE   /seasons/{seasonID}
22  * PUT      /seasons/{seasonID}
23  */
24
25  @Path( "/seasons" )
26 public class SeasonsService {}
```

Code 3.6: Klassenaufbau von BeispielServices

Wie bereits angesprochen, bestand die Möglichkeit zum Season und Episoden Zugriff auf unterschiedliche Wege. Da für jeden dieser Services eine mehrfache Definition aller Methoden nicht notwendig oder sinnvoll ist, wurde sich für eine Art Mittelweg entschieden. Die aufeinander aufbauende Variante im oberen Beispiel des SeriesService, erschien in sofern reizvoll, die die eindeutige Abfrage einer Episode über den jeweiligen Pfad realisiert werden könnte. Die zweite Variante stellt dabei eine höhere Komfortabilität dar, da zum Abrufen einer Season keine Serie ID bekannt sein müsste (die durch die XSD Definition jedoch vorhanden ist).
Daher wurden geplante POST, PUT und DELETE Methoden, jeweils nur auf den Haupttyp definierte und der Pfadzugriff in die Tiefe ermöglicht lediglich ein Holen der Daten, keine Manipulation. Diese sind in der jeweiligen Serviceklasse realisiert.

Zu Beginn wurden einzelne Methoden ausgearbeitet und auf Funktionalität in Verbindung mit dem Grizzly RESTServer getestet. Erste Versuche wurden mit Hilfe eines erstellten TestClients durchgeführt und lieferten schnell das gewünschte Ergebnis.
Innerhalb dieser Phase fand die gesamte Realisierung in der jeweiligen Methode definiert. So wurde beispielsweise das Marshalling und Unmarshalling jedes Mal erneut eingebunden.

```
1     @Path( "/{id}" )
2     @GET @Produces( "application/xml" )
3     public Serie getSingle(@PathParam("id") int id) throws
          JAXBException {
4         ObjectFactory of = new ObjectFactory();
5         Series series = of.createSeries();
```

```
6         JAXBContext jaxbContext = JAXBContext.newInstance(
              Series.class );
7
8         this.unMarshaller = jaxbContext.createUnmarshaller(); //
              Reading
9         Series rawSeries = (Series) unMarshaller.unmarshal( this
              .file );
10
11        List<Serie> series = rawSeries.getSerie();
12          for ( Serie serie : series ) {
13             if ( serie.getSerieID().intValue() == id ) {
14                 return serie;
15             }
16         }
17    return null;}
```

Code 3.7: GET Testmethode der SeriesID

```
1  public static void testGetSerie() {
2      String url = host + "/series/ss_0001wade";
3      System.out.println( "GET: " + url );
4      WebResource wrs = Client.create().resource( url );
5
6      ClientResponse response = wrs
7          .accept( MediaType.APPLICATION_XML )
8          .get( ClientResponse.class );
9
10     if ( response.getStatus() != 200 ) {
11       System.err.println( "Failed: HTTP error code: " +
             response.getStatus() );
12       return;
13     }
14     Serie output = response.getEntity( Serie.class );
15     System.out.println( "Output from Server..." );
16     System.out.println( output.getTitle() );
17   }
```

Code 3.8: Anfrage des Testclients nach Serie mit angegebener ID

Eine Abfrage der Liste lieferte im Idealfall das gesuchte Objekte.So ermöglichte obige Anfrage die Rückgabe der Serie mit angegebener ID.
Da diese Herangehensweise aus programmiertechnischer Sicht jedoch nicht besonderns effektiv und angebracht ist, fand im weiteren ein Refactoring des Webservices statt und der Klassenaufbau wurde angepasst. Ganz nach dem DRY Prinzip (don't repeat yourself) wurden wiederkehrende Informationen, wie das zuvor angesprochene Marshalling und Unmarshalling, auf eine eigene Klasse ausgelagert um Coderedundanz zu vermeiden und definierte Funktionen an mehreren Stellen zugänglich zu machen. Während der Entwicklung ergab sich daraus folgender Aufbau, bei dem einzelne Komponenten funktional miteinander Interagieren und im Zusammenspiel die synchrone Kommunikation des SerienTrackers ermöglichen.

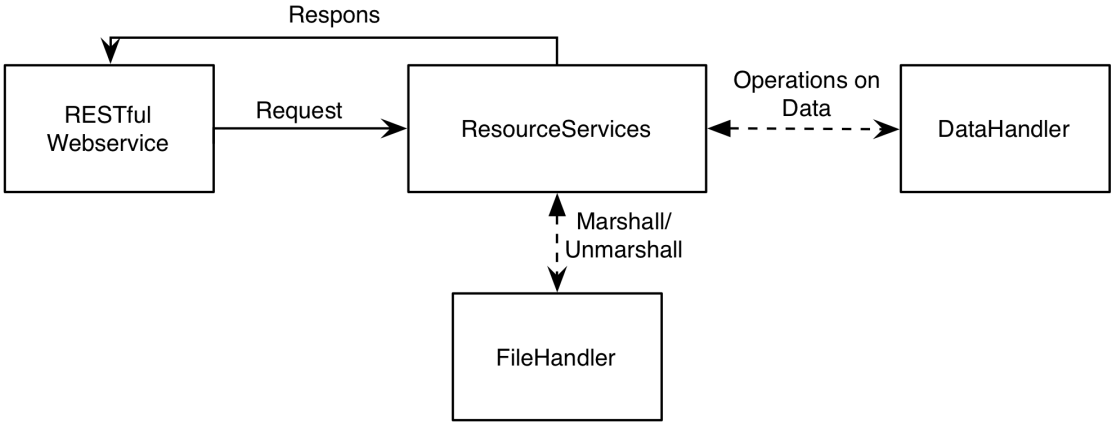


Abbildung 3.2: Funktionaler Aufbau der Komponenten zur synchrone Kommunikation

Über den RESTServer wird ein Request an die Serviceklasse einer Ressource geschickt. Die angesprochene Methode beginnt die Anfrageverabeitung und wird zuerst durch einen Logger erfasst *Logger.log(...);*. Diese Hilfsklasse wurde erstellt, um während der Entwicklung einzelne Schritte zu dokumentieren und erfolgreiche und nicht erfolgreiche Arbeitsschritte genauer bestimmen zu können. Die Serviceklasse greift dann gegebenfalls auf Methoden des FileHandlers und DataHandlers zu. Klassen in denen jeweilige Funktionen ausgelagert wurden. Der FileHandler kümmert sich um das Lesen und Schreiben der Daten in die XML Files durch Marshalling und Unmarshalling Operationen. Zudem überprüft sie, ob für entsprechende Objekte bereits XML Files zu Grunde liegen, ansonsten werden diese neu erstellt. Der DataHandler stellt die Methoden bereit, um auf vorhandene Daten zu operieren und den jeweiligen Request funktional umzusetzen. So wird biespielsweise ein Serienobjekt in der SeriesServiceklasse erstellt und durch eine Methode des Handlers *Serie serie = dh.getSerieByID(id);* durchsucht der zugehörige SeriesDataHandler vorhandene Elemente nach dem passenden Attribut und liefert die gesuchte Instanz zurück. Der Rückgabewert wird dann wieder vom Service geprüft und per Response zurückgeliefert. Gegebensfalls folgt die Rückgabe eines Statuscodes beim Eintreten eines des darin beschriebenen Ereignisses.

Die DataHandler wurden entsprechende der persistende Datenstruktur in die jeweiligen Typen aufgeteilt und übernehmen für zugehörige Ressourcen die Verarbeitung. Dadurch entwickelte sich der SeriesDataHandler, UsersDataHandler und ListsDataHandler. Im momentanen Entwicklungsstand des Projektes, verarbeitet der SeriesDataHandler vorerst auch nur die Informationen einer Serie.

Entsprechende Unterstützung einer Season oder Episode nach diesem Prinzip ist aus zeitlichen Gründen noch nicht implementiert, die Funktionalität der entsprechenden HTTP Operation findet noch auf eine andere Weise statt. Entsprechende GET Anweisungen könnten über die *series/serieID/seasons/seasonID/episodes* - URI ermöglicht werden. POST, PUT und DELETE Methoden sind innerhalb der Services umgesetzt.
Die Kommunikation zwischen Client und Server ermöglicht bisher einen Austausch zwischen den Serien und Userdaten. In Bezug auf Serien, wurde ein ListService erarbeitet, der jedoch noch nicht auf Funktionalität getestet werden konnte, da der zugehörige ListDataHandler noch Probleme aufweist. Für eine Lösung dieses Problems wäre dabei die Anpassung im Stile des UsersDataHandlers möglich. Dieser greift auf entsprechenden Fileordner auf der Database zu um die passende Userdatei zu erhalten oder legt alternativ neue Files für die einzelnen User an. Der ListsDataHandler geht diesen Schritt etwas anders an, indem er auf die lists.xml zugreift. Auch hier müsste ein Zugriff auf den gesamten Ordner realisiert werden und dann die einzelnen Dateien durchsucht werden. Momentan führt dieser Ansatz nicht zum erwünschten Zweck, da innerhalb der selbstdefinierten Database keine lists.xml existiert, die in der Zeit der Ausarbeitung als Testdatei bestand. Bei der Entwicklung der Testdaten wurde für jede Liste eine seperate Datei angelegt. Eine Zugriff auf eine XML File die mehrere Listen enthält wäre über die genre.xml oder user.xml vorstellbar. Dies wäre ein Grundansatz um dieses Problem entgegen zu gehen. Die entsprechende Pfadauslegung und anschließende Abfrage in Bezug auf die aktuelle Datenstrukturierung.
Zusätzlich zu den Services sei noch erwähnt, dass neben den jeweiligen RessourceServices ein zusätzlicher ImageService entwickelt wurde. Dieser arbeitet ebenfalls unter Verwendung der GET Methode und gibt dabei gespeicherte Bilder nach Filenamen zurück. Die Anwendung dieser Klasse, spielt jedoch erst in der Cliententwicklung eine Rolle.

Im weiteren Rahmen der Dokumentation soll hierbei nicht auf jede einzelne Klasse eingegangen werden. Für einen kurzen Einblick in die Ausarbeitung, werden deshalb im folgenden nur kurz einzelne Aspekte der verschiedenen Methoden am Beispiel des SeriesService dargestellt.

```
1   @GET
2   @Produces( MediaType.APPLICATION_XML )
3   public Response getSeries() {
4     Logger.log( "GET series called" );
5     Series series = dh.getSeries();
6
7     if ( series == null )
8       return Response.status( 404 ).build();
9
10    return Response.ok().entity( series ).build();
11  }
```

Code 3.9: GET Methode /series

Im Vergleich zur vorherigen GET Methode, fällt die gewonnene Überischtlichkeit und Einfachheit auf. Der DataHandler dh führt die eigentliche Operation aus und liefert in dem Fall alle Serien zurück die vorhanden sind. Sind keine Serien auffindbar, wird der NOT FOUND Statuscode 404 als Response zurückgegeben. Bei Erfolg die gefundenen Entitäten.

```
1  @POST
2    @Consumes( MediaType.APPLICATION_XML )
3    public Response addSerie( Serie newSerie ) {
4      Logger.log( newSerie.getTitle() );
5
6      String id = "ss_" + Hasher.createHash( newSerie.getTitle()
           );
7
8      if ( dh.SerieExistsByID( id ) )
9        return Response.status( 409 ).build();
10
11     newSerie.setSerieID( id );
12
13     if ( ! dh.addSerie( newSerie ) )
14       return Response.status( 500 ).build();
```

```
15
16      URI location = null;
17      try {
18        location = new URI( RESTServerConfig.getServerURL() + "/
              series/" + id );
19      } catch ( URISyntaxException e ) {
20        e.printStackTrace();
21      }
22      return Response.created( location ).build();
23    }
```

Code 3.10: POST Methode /series

Die POST Methode zum Pfad /series erstellt eine neue Serie und fügt sie in die Liste ein. Zuerst wird eine neue ID der Serie erstellt. Dazu wird mit der eigenen Hilfsklasse Hasher eine eindeutige Kennung erstellt, wie sie im Abschnitt zur XML Entwicklung bereits angesprochen wurde. Es folgt die Kontrolle, ob die Serie anhand der ID bereits exisitert, wenn ja wird der 409 Conflict Statuscode ausgegeben, um dies zu vermeiden. Sofern es Probleme beim hinzufügen gab, wird dies vom Statuscode 500 informiert über serverseitige Probleme. An geplanter location im URI Pfad wird das Objekt anschließend erstellt.
Auch die beiden weiteren Methoden PUT und DELETE sind nach diesem Prinzip strukturiert. PUT ändert die Daten des bestehenden Objektes zu angegebener ID. Sofern die Serie existiert und Kontrollabfragen keinen Fehlercode zurückgeben, war die Änderung erfolgreich. DELETE entfernt nach gleicher Abfrage das gewünschte Objekte aus vorhandenem Datenbestand.

Dieser kurze Einblick in den RESTful Webservice, sollte einen Überblick über die Ausarbeitung der synchronen Kommikationsvorgänge schaffen.
Zu Beginn des Projektes wurden in der Konzeptidee diverse Möglichkeiten überlegt, die vom System unterstützt werden können. Für Serieninteressierte wurde beispielsweise das Favorisieren von Serien, Anlegen von Listen oder Markieren und Bewerten von Episoden angedacht. Die Auseinandersetzung mit dem Projekt, zeigte bei diesem Meilenstein, dass dieses Ziel vom entsprechenden Umfang her zu hoch ausgelegt war. Daher wurde der Fokus auf einen Teil dieser Möglichkeiten gelegt und nur ein Teil davon vertieft umgesetzt. Der Aspekt des Favorisieren und die Listen stellten für den weiteren Verlauf den größten Nutzen dar, weil anhand dieser Elemente die entsprechende Benachrichtigung innerhalb der asynchronen Kommunikation realisiert werden kann. Die weiteren Ansätze wurden zugunsten des Projektzieles außen vor gelassen, könnten aber in einem größer angelegten Projekt dieser Art eine Ausbaumöglichkeit darstellen.

3.4 Interaktion mittels XMPP Server

3.4.1 Vertiefung

XMPP steht für Extensible Messaging and Presence Protocol und war Bestandteil des 4. und 5. Meilensteins. XMPP ist ein offener Standard der verschiedene Erweiterungen beinhaltet. In diesem Projekt wird für die asynchrone Interaktion das Publish-Subscribe Paradigma verwendet, welches in der XMPP Spezifikation XEP-0060[1] definiert wird.

Für die weitere Planung, um die Erweitung in das Projekt einzubinden, war zunächst eine Recherche über die in der Spezifikation erwähnten Bestandteile der Erweiterung von Nöten. Im folgenden werden die Ergebnisse aufgelistet:

Node
Ein Node, auch *Topic* genannt, ist ein virtueller Speicherort für Informationen. Die Informationen können einem Node hinzugefügt werden sowie auch wieder ausgelesen werden. Ein Node wird durch eine unique Node ID gekennzeichnet.
Ein Untertyp des Nodes ist ein *Leaf Node*, welcher nur veröffentlichte Items beinhaltet.

Publisher
Die Publisher sind Instanzen, die Nodes mit *Items* befüllen können.

Subscriber
Die Subscriber sind Instanzen, die Nodes abonniert haben und die an den Node gesendeten *Items* empfangen können.

Item
Ein Item ist ein XML Fragment, welches einem Node zugeordnet werden kann.

Payload
Unter Payload ist zu verstehen, dass *Items* um Nutzdaten erweitert werden. Die Nutzdaten sind dabei durch ein XML Schema definiert und können beim Subscriber ausgelesen werden.

Service Discovery
Der Service Discovery ist ein Bereich der XMPP Entität, der den Umgang mit Anfragen und die Antwort bezüglich bestimmter Protokolle regelt. Er unterstützt dabei zum Beispiel das Auslesen von Informationen eines Nodes.

Item
Ein Item ist ein XML Fragment, welches einem Node zugeordnet werden können.

Auf dieser Basis konnten die Punkte nun auf das Projekt übertragen werden.

[1] `http://xmpp.org/extensions/xep-0060.html`

Die **Nodes** bzw. Topics werden die vorhandenen Serien sein. Ein Node wird durch die Serien ID eindeutig identifizierbar sein. Die Nodes können von den Serieninteressierten abonniert werden, sie sind somit die **Subscriber**.
Die Nodes sollten von einer administrienden Position angelegt werden, da das Löschen eines Nodes nur von Eigentürmer des Nodes, also dem Anleger, wieder gelöscht werden kann. Dies wäre ein Fall für den in der Konzeption schon erwähnten *Content-Admin*. In der weiteren Entwicklung kommt noch ein *Bot* zum Einsatz, welcher ebenfalls diese Aufgabe übernehmen wird.
Gleichzeitg sind diese beiden Instanzen auch die **Publisher**. Sie werden dafür zuständig sein, Nachrichten an den Subscriber über den Node zu übermitteln. Der Inhalt dieses **Items** wird die Informationen über die demnächst startende Episodenaustrahlung beinhalten.

Bei einer möglichen Implementierung der in der Konzeptvorstellung vorgestellten Freundefunktion, würde sehr wahrscheinlich weitere Nodes hinzukommen, d.h. jeder User würde ebenfalls durch einen Node repräsentiert werden können. Alternativ könnte auch die (Gruppen-)Chat Erwweiterung von XMPP als Einsatzmöglichkeit herangezogen werden.

3.4.2 Realisierung eines Clienten

Nach der Recherche sollte im Meilenstein 5 ein XMPP Client realisiert werden, der Leafs abonnieren, Nachrichten (mit Nutzdaten) empfangen und veröffentlichen und mögliche Eigenschaften eines Services anzeigen können soll.
Im Folgenden werden nun die Hauptbestandteile der Implementierung dokumentiert und welche Probleme dabei auftraten.

Als Serverumgebung wurde eine Openfire-Instanz eingesetzt und für die Übersetzung zwischen dem Java Client und dem Server die Bibliothek *Smack*.

Im Hinblick auf den User-Clienten wurde die Implementierung der folgenden Funktionen bereits modular aufgebaut, sodass sie später wiederverwertet werden können.

Für den Verbindungsaufbau müssen Hostname und Port des Openfire-Servers bekannt sein. In der weiteren Entwicklung hat es sich bewertet die Option `Connection.DEBUG_ENABLED = true` zu setzen. Es öffnet sich ein Debug-Fenster, welches die Interaktionen zwischen Client und Server darstellt.

```
1 /**
2  * Sets up a connection to the XMPP server.
3  *
4  * @param String hostname
5  * @param int port
6  * @return boolean
```

```
 */
public boolean connect( String hostname, int port ) {
  // Check if already connected
  if ( cn != null && cn.isConnected() )
    return true;

  try {
    //Connection.DEBUG_ENABLED = true;
    ConnectionConfiguration config = new
       ConnectionConfiguration( hostname, port );
    cn = new XMPPConnection( config );
    cn.connect();
    Logger.log( "Connection established" );
  } catch ( XMPPException e ) {
    return false;
  }

  return true;
}
```

Code 3.11: Auszug aus ConnectionHandler für den Verbindungsaufbau

Weiterhin muss für die Verbindung mit dem XMPP Server ein Benutzeraccount vorhanden sein, damit der Login erfolgreich aufgebaut werden kann.

```
/**
 * Login.
 *
 * @param String username
 * @param String password
 * @param String resource
 * @return boolean
 */
public boolean login( String username, String password, String
   resource ) {
  try {
    SASLAuthentication.supportSASLMechanism( "PLAIN", 0 );
    this.cn.login( username, password, resource );
    Logger.log( "Login successful" );
```

```
14    } catch ( XMPPException e ) {
15      Logger.err( "Login failed" );
16      return false;
17    }
18
19    // Init the Pub Sub Manager
20    this.psh = new PubSubHandler();
21
22    return true;
23  }
```

Code 3.12: Auszug aus ConnectionHandler für den Login

Hierbei ist anzumerken, dass im Bezug auf die Sicherheit das Passwort unverschlüsselt zwischen Client und Server transportiert wird. Im produktiven Einsatz sollte dieser Schritt nochmal überdacht werden und auf eine Verschlüsselung gesetzt werden.

Nachdem die Verbindung steht kann eine Instanz des PubSubManagers angelegt werden. Dieser wird ebenfalls in Smack bereitgestellt. Wegen es Modualren Aufbaus wurde auch hier eine Wrapper Klasse generiert, die unter dem Namen *PubSubHandler* zu finden ist. Mit Hilfe dieser Klassen können die weiteren Operationen für die Bearbeitung von Nodes durchgeführt werden.

Für das Anlegen eines Nodes gibt es verschiedene Dinge zu beachten. Während der Entwicklung traten an dieser Stelle die meisten Probleme auf.

Jeder Node kann mit einer standardmäßigen Konfiguration ausgestattet werden. Er lässt sich allerdings auch durch die in Smack mitgelieferte *ConfigureForm* konfigurieren.

```
1   public LeafNode createNode( String nodeID, String nodeTitle ) {
2     LeafNode node = null;
3
4     try {
5       // Node configuration
6       ConfigureForm form = new ConfigureForm( FormType.submit );
7       // Access
8       form.setAccessModel( AccessModel.open );
9       // Publish
10      form.setPublishModel( PublishModel.open );
11      // With payload
```

```
12      form.setDeliverPayloads( true );
13      // Delete message
14      form.setNotifyRetract( true );
15      // Persistent data
16      form.setPersistentItems( false );
17      // An frindly name
18      form.setTitle( nodeTitle );
19
20      // Create new node with configuration
21      node = (LeafNode) this.psm.createNode( nodeID, form );
```

Code 3.13: Auszug aus PubSubHandler.createNode() für das Anlegen eines Nodes

Der Code 3.13 war schlussendlich die für uns funktionierende Variante, womit die Verwendung des Nodes im Clienten funktionierte. Zusammengefasst:

Jeder kann den Node abonnieren, jeder könnte etwas auf den Node veröffentlichen, Nachrichten müssen einen Payload besitzen und es handelt sich um einen transienten Node.

Der nächste Punkt ist nun das Senden eines Items an einen Node. Hierfür bietet die Smack API zwei Wege: node.send(new Item()) oder node.publish(new Item()), zweiters arbeitet asynchron.
Bei der Implementierung hat die Variante über *send()* nicht funktioniert und warf den Fehler *bad-request(400)* raus. Aus diesem Grund wurde in der weiteren Entwicklung auf die *publish()* Variante gesetzt, siehe Code 3.14.

```
1   /**
2      * Sends a test item to the selected node.
3      */
4     private void sendPayload() {
5       // Create a new message object
6       ObjectFactory factory = new ObjectFactory();
7       Message message = factory.createMessage();
8       message.setContent( testNodePayload.getText() );
9
10      StringWriter notification = new StringWriter();
11      try {
12        JAXBContext jaxb_context = JAXBContext.newInstance(
              Message.class );
```

```
13        Marshaller marshaller = jaxb_context.createMarshaller();
14        marshaller.setProperty( Marshaller.JAXB_FRAGMENT, true );
              // Marshall without namespace
15        marshaller.setProperty( Marshaller.JAXB_FORMATTED_OUTPUT,
              true );
16        marshaller.marshal( message, notification );
17      } catch ( JAXBException e ) {
18        return;
19      }
20
21      // Get selected node
22      String selectedNode = (String) coboxExistingNodes.
           getSelectedItem();
23
24      // Publish item to node
25      PubSubHandler psh = this.ch.getPubSubHandler();
26      LeafNode node = psh.getNode( selectedNode );
27
28      node.publish(
29        new PayloadItem<SimplePayload>(
30          null,
31          new SimplePayload(
32            "message",              // Element name
33            "",                     // Namespace
34            notification.toString() // Payload
35          )
36        )
37      );
38    }
```

Code 3.14: Auszug aus XMPP Client für das Senden eines Items mit einem JAXB Objekt als Payload

Beim Senden von JAXB Objekten muss darauf geachtet werden, dass der Wert Marshaller.JAXB_FRAGMENT auf true gesetzt wird, damit kein Namespace mitausgegeben wird. Da durch das Veröffentlichen ein weiterer Namespace zum XML Objekt hinzugefügt wird, siehe Code 3.15, muss dieses vor der Verarbeitung durch den Marschaller entfernt werden.

```
1  public void handlePublishedItems( ItemPublishEvent<Item> items
       ) {
2      for ( Item item : items.getItems() ) {
3        String rawXML = ((PayloadItem<SimplePayload>) item).
             getPayload().toXML();
4
5        // Remove the the namespace
6        String xml = rawXML.replaceFirst( " xmlns=\"http://jabber
             .org/protocol/pubsub\"", "" );
7
8        // Get a JAXB object of the XML data.
9        Message payload = null;
10       try {
11         JAXBContext jaxbContext = JAXBContext.newInstance(
               Message.class );
12         Unmarshaller unmarshaller = jaxbContext.
               createUnmarshaller();
13         StringReader xmlString = new StringReader( xml );
14         payload = (Message) unmarshaller.unmarshal( xmlString )
               ;
15       } catch ( JAXBException e ) {
16         e.printStackTrace();
17       }
18       Logger.log( "Message Content: " + payload.getContent() );
19     }
20   }
21 }
```

Code 3.15: Auszug aus XMPP Client für das Verarbeiten eines veröffentlichten Items mit einem JAXB Objekt als Payload

Damit die Items einen Empfänger haben, kommen die Subscriber zum Einsatz. Die Smack Bibliothek bietet dafür die Methoden node.subscribe() bzw node.unsubscribe(). Außerdem muss über node.addItemEventListener ein Listener hinzugefügt werden, welcher die Verarbeitung der veröffentlichten Items übernimmt.
Hierbei muss beachtet werden, dass dieser Listener bei einer Kündigung auch wieder gelöscht werden müssen, da es sonst später zu mehrfachen Verarbeitung kommt.

Dieser Client wurde in der laufenden Entwicklung weiter als XMPP Debug Client ausgebaut um so die Richtigkeit des späteren User Clienten zu prüfen.

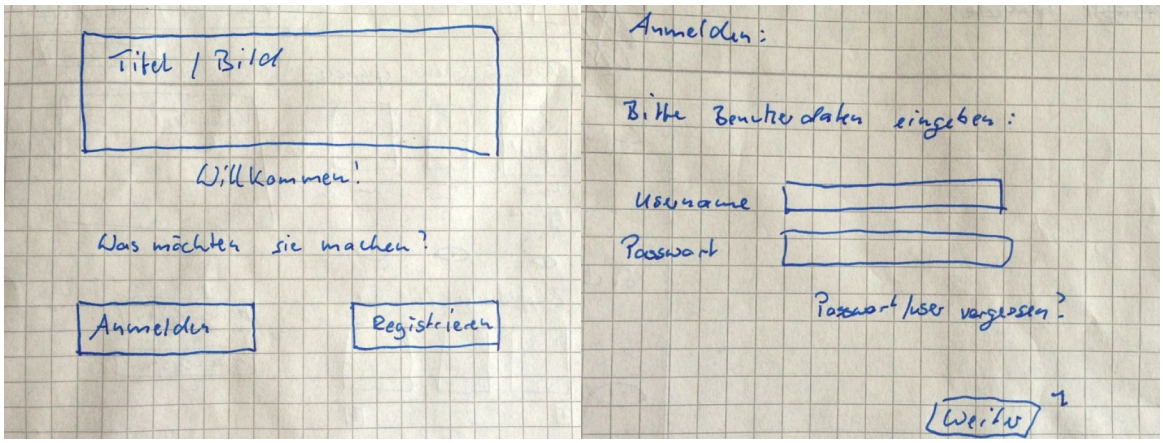
3.5 Entwicklung eines User-Client

3.5.1 Vorbereitung und Layoutentwicklung

Im finalen Meilenstein des Projektes, steht die Entwicklung eines grafischen Userinterface im Mittelpunkt, der die Funktionalität des Systems repräsentiert. Die Applikation soll die entsprechende Verbindung zum Server aufbauen und den Datenaustausch ermöglichen.

Vor der praktischen Umsetzung mit Java Swing, wurde ein mögliches Konzeptlayout erstellt und entsprechende Alternativen ausprobiert.Hierbei handelt es sich um erste Ideen wie die Anwendung aufgebaut sein kann und welche Elemente benötigt werden. Da in der letztendlichen Umsetzung aber hauptsächlich die Funktionalität im Mittelpunkt steht, wurde speziell darauf geachtet, dass alle Funktionalitäten in einem logischen Kontext eingebunden werden und anwendbar sind. Auf genaue Designkonzepte zum optimalen Aufbau wurde hierbei verzichtet, da sie im entsprechenden Projektkontext nicht an erster Stelle stehen sollten. In einem größer angelegten Projekt sollte hierbei eine intensivere Auseinandersetzung folgen mit entsprechender Validierung und Testdurchläufe.

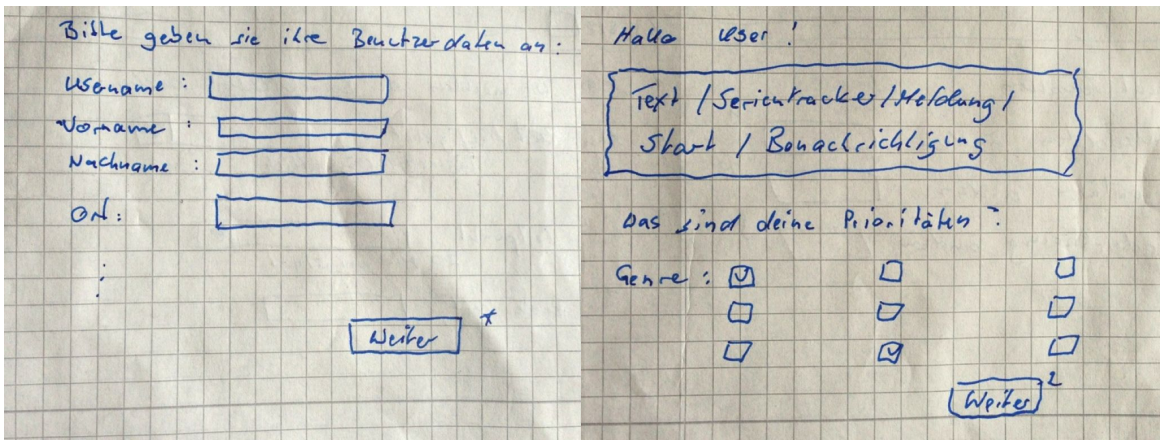
Die Applikation soll mit einer Startseite (Abb. 3.3a) beginnen, von der aus der Anwender die Möglichkeit hat sich zu identifizieren. Entweder er meldet sich mit bereits vorhandenen Daten an oder er registriert sich als neuer Benutzer. Sind bereits Accountdaten vorhanden, so folgt eine Eingabe des Usernamen und des Passworts (Abb. 3.3b). Sollte der User eine der beiden Informationen vergessen haben, so besteht die Option *Passwort/Username vergessen?*, wobei die funktionale Implementierung nicht stattfinden wird und dies eher als Platzhalter dient. Nach Eingabe der Daten und Prüfung auf Korrektheit, wird zur *Homeseite* weitergeleitet, die als Hauptseite dient und von der aus die verschiedenen Funktionen angesteuert werden.



(a) Startseite (b) Einloggen

Abbildung 3.3: GUI Layout Skizzen: Startauswahl und Einloggen

Entscheidet man sich für die Auswahl *Registrieren* wird ein neuer Account angelegt. Auf der ersten Seite (Abb. 3.4a) werden die Grundinformationen eingegeben, wobei hier nur Username und Passwort und Name notwendig sind und die zusätzlichen Informationen jederzeit ergänzt werden können. Nach Eingabe und Kontrolle folgt der zweite Teil der Registrierung, die Auswahl der Genreprioritäten (Abb. 3.4b). Da der Serientracker die Option anbietet, Informationen zu Serien eines bestimtmen Genres zu erhalten, dient dieser Schritt dazu bestimmte Genres zu abonnieren. Auch hier soll die Auswahl später geändert werden können.



(a) Registrieren (b) Prioritäten

Abbildung 3.4: GUI Layout Skizzen: Registrierung

Nach erfolgreicher Identifizierung des Users folgt der Homebereich (Abb. 3.5). Dies ist der eigentliche Ausgangspunkt für alle Aktivitäten und dient als Benutzerzentrale. Am oberen Bereich findet sich eine statische Menüleiste, die sich in diesem Aufbau durch alle weiteren Bereiche zieht und jederzeit zugänglich ist. Information zum angemeldeten User und die Optionen *Home*, *Favoriten*, *Einstellung*und*Logout*. Home führt jederzeite zur Hauptübersicht, Einstellung ermöglicht entsprechende Möglichkeiten zur Accountverwaltung und Logout meldet den aktuellen Benutzer ab und führt wieder zur Anmeldung. Die Option Favoriten war als Verwaltung zu den angelegten Lieblingsgenre geplant, wurde aber in weiteren Entwürfen in die Einstellungen mit eingebunden, da bis auf eine einfache Auswahl kein größerer Nutzen für den User angeboten wird. Für den Fall das Adminrechte vorhanden sind, wurde in einem

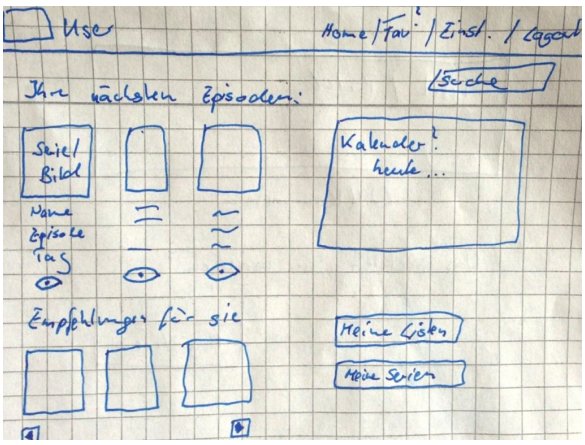
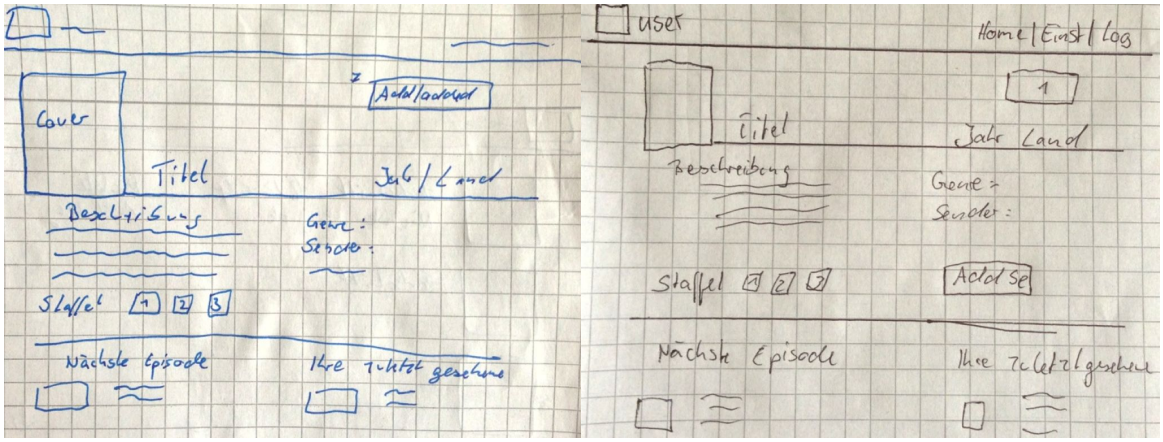


Abbildung 3.5: GUI Skizzen: Homeansicht

späteren Entwurf der zusätzliche Menüpunkt *Hinzufügen* eingebunden. Dort lassen sich neue Serien, Staffeln, Episoden und Genrelisten anlegen.

Da im Mittelpunkt des Konzeptes die Benachrichtigung über neue Episoden steht, werden dem User bereits nach einloggen die wichtigsten Informationen präsentiert. Neben einer Suche, werden die Ausstrahlungszeitpunkte der nächsten Episoden angezeigt. Neben dieser Übersicht sollten zudem direkte Benachrichtigungen in Form von Pop-Ups stattfinden, weshalb diese Ansicht vorallem der Erinnerung und Verwaltung dient. Inwiefern eine Darstellung der heutigen Serien in Form eines Kalender realisierbar wäre ist zur Zeit der Konzipierung noch unklar, sollte aber eher als *nice-to-have Feature* betrachtet werden.
Dazu gab es die Überlegung Serienempfehlung zu geben, basiernd auf abonnierten Genres. Buttons zu *Meine Serien* und *Meine Listen* führen zu entsprechenden Unterkategoerien. Meine Serien zeigt einer Auflistung der Serien über die man Benachrichtigungen erhalten möchte und die derzeit angeschaut werden. Meine Listen gibt einen Überblick über angelegte Listen zur Ordnung von Serien nach bestimmten Themen und bietet zudem die Möglichkeit eine neue Liste anzulegen.

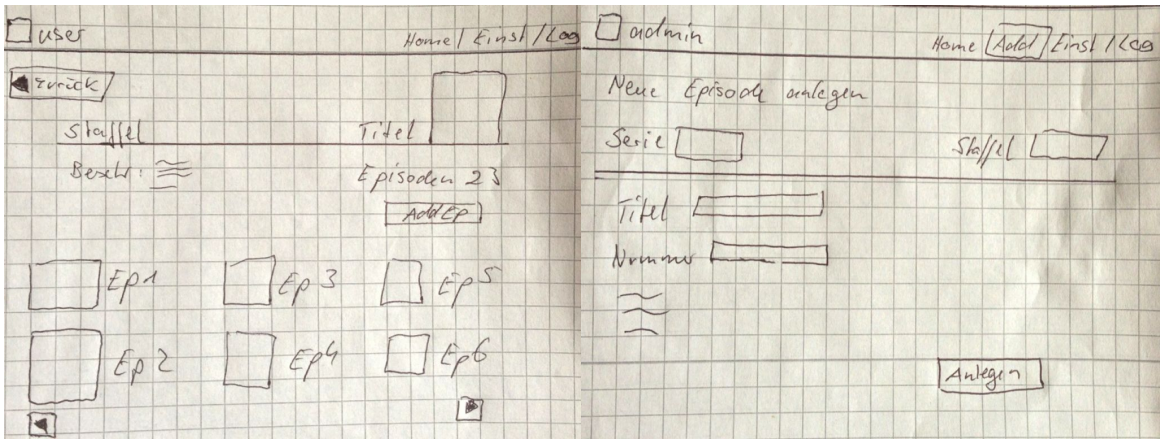
Nach Auswahl einer bestimmten Serie werden vorhandene Information präsentiert. Allgemeine Informationen, eine Übersicht zu vorhandenen Staffeln und eine Darstellung der nächsten Episode die Ausgestrahlt wird und gegebenfalls die Episode die der Benuter zuletzt gesehen hat. Hierbei würde die Verwaltung mit Hilfe der Seen List stattfinden, wobei auch hier die letztendliche Umsetzung eher unwahrscheinlich ist. Auf dieser Seite (Abb. 3.6a) kann die Serie einer bestimmten Liste hinzugefügt werden und wird damit abonniert. In einer weiteren Variante der Serienseite (Abb. 3.6b) handelt es sich um die Darstellung unter Adminrechte, wobei die *Add to List* Option nun zum Editieren dieser Seite führt und auch neben den Serien eine entsprechende Editierfunktion eingeführt wird.



(a) Serienseite User (b) Serienseite Admin

Abbildung 3.6: GUI Layout Skizzen: Serienübersicht

Die jeweilige Seasonseite (Abb. 3.7a) zeigt eine Vorschau zu vorhandenen Episoden und bietet dem Admin erneut die *Add Episode* Funktion (Abb. 3.7b). In welcher Form die Episoden aufgeführt werden ist zu diesem Zeitpunkt noch nicht festgelegt und wird bei entsprechender Umsetzung entschieden. Wird eine neue Episode angelegt (Abb. 3.7b) wird entsprechende Serie und Staffel referenziert und die einzelnrn Informationen eingegeben.



(a) Seasonseite (b) Episode anlegen

Abbildung 3.7: GUI Layout Skizzen: Seasonübersicht und Verwaltung

Bei der Planung der GUI, wurden die einzelnen Seiten nach entsprechenden Ideen konzipiert und weiterentwickelt. Die vorgestellten Entwürfe sollten einen kurzen Einblick in die Planungsphase geben und dienten in dieser Form keiner finalen Vorlage, nach der die Entwicklung stattfindet. Aus diesem Grund wird hier im weiteren nicht auf jede einzelne Unterseite eingegangen.

Während bei den ersten Varianten vorallem der mögliche Aufbau und die Darstellung der einzelnen Elemente im Vordergrund stand, wurden in späteren Versionen speziell auf entsprechende Funktionalität und Verweisen geachtet. Dabei entstand unter anderem die Adminansicht, welche die Kategorien um verwaltende Optionen erweitert. Ob jede Vorstellung realisierbar bzw. im Projektkontext notwendig ist und wie sich entsprechendes Layout letztendlich entwickelt hat, wird in der Umsetzung dargestellt.

3.5.2 Umsetzung

Da im Voraus schon auf eine ordentliche Projektstruktur und dementsprechend wiederverwertbare Klassen gesetzt vorhanden waren, konnten diese für den User-Client wieder genutzt werden.

Für die GUI sollte *Swing* verwendet werden, welches standardmäßig bei der Komponentenpositionierung auf fixe Pixel setzt. Als Alternative stand ein GUI Builder im Raum oder ein ordnentlicher Layoutmanager.
Die Wahl fiel dabei auf den Layout Manager namens MigLayout[2].

Das Ziel für den User-Client war in erster Linie die Basisfunktionen verfügbar zu machen. In Bezug auf zuvor erstellte Konzeptskizze, zeigte sich, dass viele Funktionen angedacht wurden, die für die letztendliche Realisierung keine bedeutende Rolle spiele und aus diesem Grund ersetz wurden. Die gesamte Home Seite stellt nun eine funktionale Übersicht über vorhandene Serien dar. Eine Auswahl führt zu einer kurzen Vorschau, die in der Detailansicht dann die Informationen einer Serien mit Staffeln und Episodenvorschau hervorruft. Alle Funktionen wie Listenerstellung oder Übersicht, sowie Episodendetails konnten aus zeitlichen Gründe nicht mehr umgesetzt werden. Dadurch ergaben sich nun bei Abgabe folgende Funktionen, die anhand von GUIScreenshots einen Einblick in die Umsetzung geben sollen:

- Registrierung eines neuen Accounts über XMPP und REST API
- Anmeldung via XMPP und REST API
- Auslesen der Benutzerdaten über REST API
- Auslesen von Serien über REST API
- Auslesen von Staffeln über REST API
- Auslesen von Episoden über REST API
- Anlegen einer neuen Serie über REST API
- Abonnieren einer Serie über XMPP PubSub Extension
- Empfangen von Notifcations über XMPP PubSub Extension

[2] `http://www.miglayout.com/`

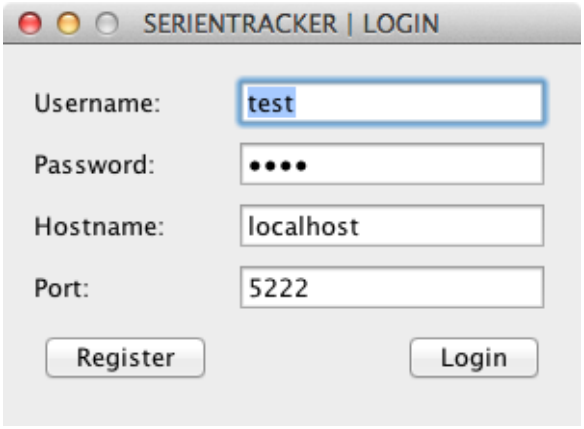


Abbildung 3.8: Anmeldung via XMPP und REST API

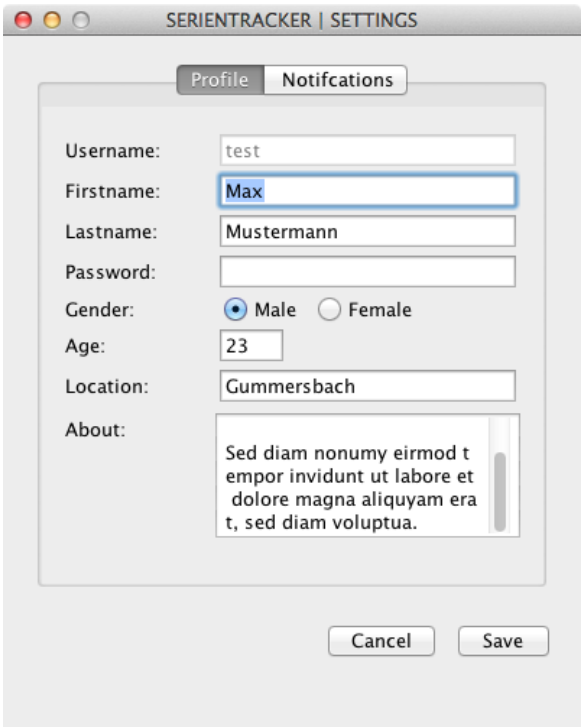


Abbildung 3.9: Auslesen der Benutzerdaten über REST API

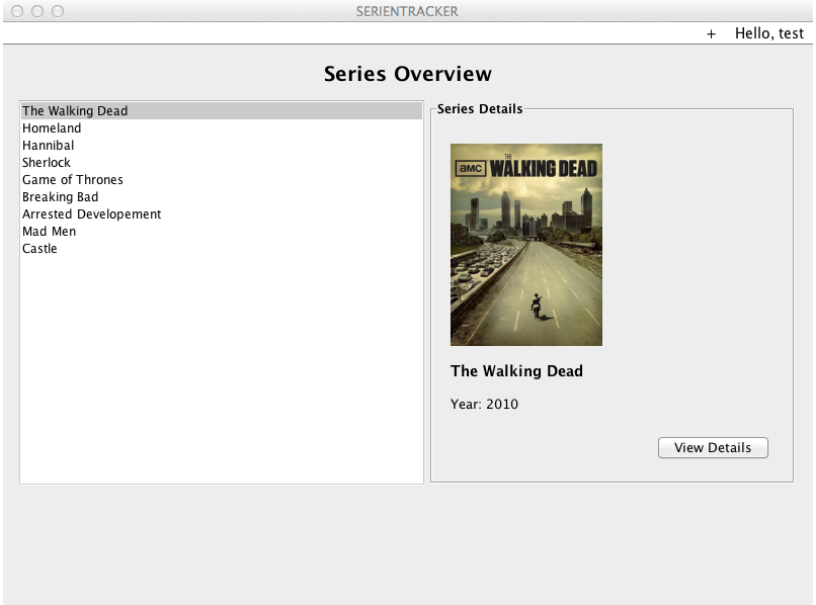


Abbildung 3.10: Auslesen von Serien über REST API

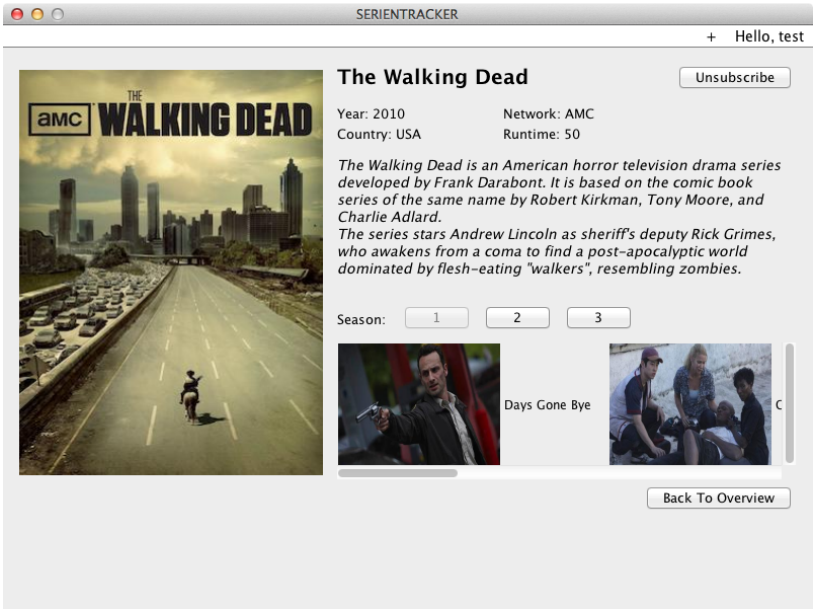


Abbildung 3.11: Auslesen von Staffeln und Episoden über REST API

Abbildung 3.12: Anlegen einer neuen Serie über REST API

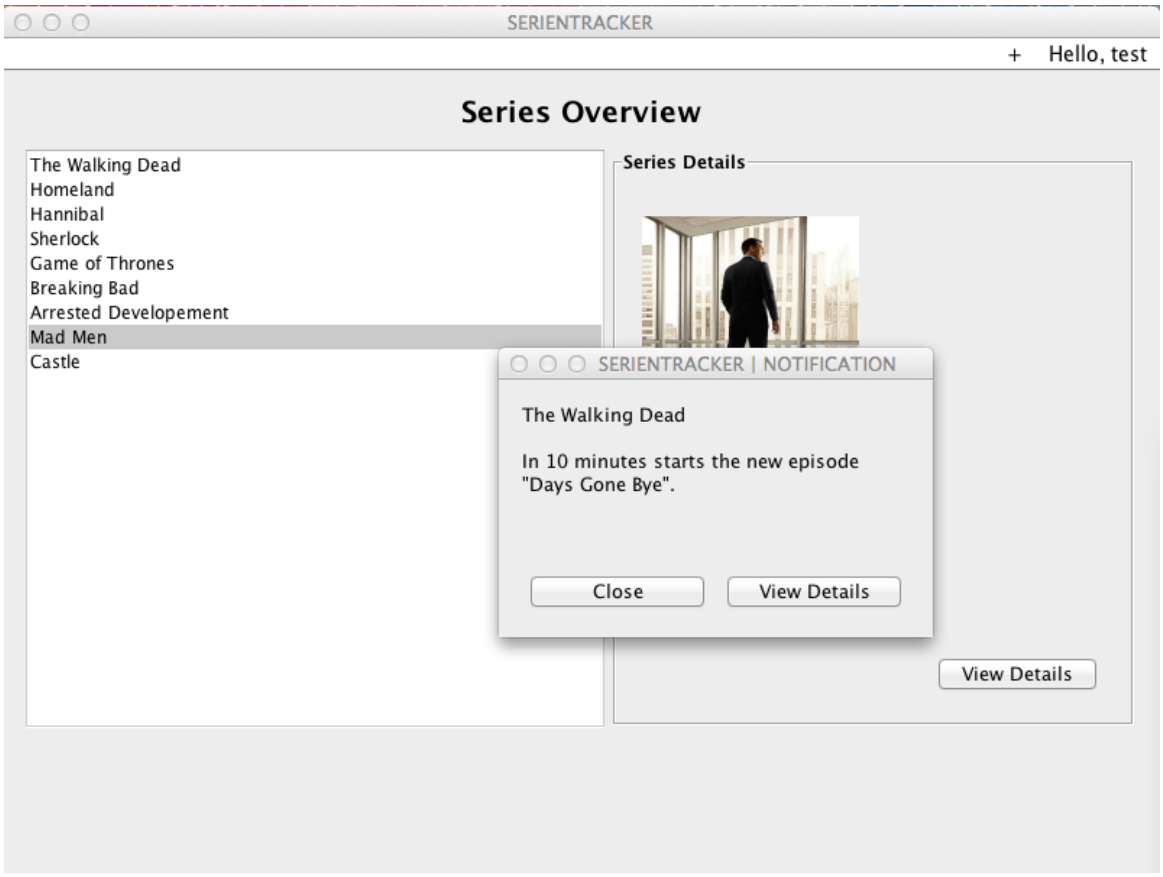


Abbildung 3.13: Empfangen von Notifcations über XMPP PubSub Extension

Übersicht Packages

Zum Schluss eine Übersicht der nun vorhanden Packages und deren Funktionen im Zusammenhang mit dem User-Client.

de.fhkoeln.gm.serientracker.client
In diesem Paket befindet sich die Main-Klasse für den User-Client.

de.fhkoeln.gm.serientracker.client.gui
In diesem Paket befinden sich die Klassen für die Realisierung der GUI für den User-Client.

de.fhkoeln.gm.serientracker.client.utils
In diesem Paket befinden sich die mehrer modulare Klassen, die Aufgaben, wie Login, Subscription oder Session-Verwaltung übernehmen.

de.fhkoeln.gm.serientracker.jaxb
In diesem Paket befinden sich vom JAXB Generator generierten Objekte.

de.fhkoeln.gm.serientracker.utils
In diesem Paket befinden sich zwei Helfer-Klassen, zum einem der Hasher, zum Generieren eines MD5 Hashs sowie der Logger.

de.fhkoeln.gm.serientracker.webservice
In diesem Paket befindet sich die Main-Klasse für den REST Server, sowie die Klasse für die Servereinstellungen.

de.fhkoeln.gm.serientracker.webservice.data
In diesem Paket befinden sich die Data-Handler für die Ressourcen. Aufgrund des modularen Aufbaus wurden diese aus den Service-Klassen extrahiert. Sie sind das Bindeglied zwischen dem Service und dem File-Handler, siehe utils Paket.

de.fhkoeln.gm.serientracker.webservice.resources
In diesem Paket befinden sich die Services für die Ressourcen auf welche über die Clienten zugegriffen wird.

de.fhkoeln.gm.serientracker.webservice.utils
In diesem Paket befindt sich eine Helfer-Klasse, dem File-Handler, welche die Verbindung zwischen Dateisystem und JAXB Objekt implementiert.

de.fhkoeln.gm.serientracker.xmpp
In diesem Paket befindet sich die Main-Klasse für den XMPP-Debug-Client, sowie die Klasse für die Servereinstellungen.

de.fhkoeln.gm.serientracker.xmpp.gui
In diesem Paket befinden sich die Klassen für die Realisierung der GUI für den XMPP-Debug-Client.

de.fhkoeln.gm.serientracker.xmpp.utils
In diesem Paket befinden sich die mehrer modulare Klassen, die Aufgaben, wie Verbindungsaufbau oder Subscription-Verwaltung übernehmen.

de.fhkoeln.gm.serientracker.bot
In diesem Paket befindet sich die Main-Klasse für den Bot-Client. Siehe hierfür Abschnitt "Zusatz: Bot-Client".

de.fhkoeln.gm.serientracker.bot.jobs
In diesem Paket befinden sich die Klassen, die die Jobs übernehmen, wenn der Cronjob das Event ausführt.

de.fhkoeln.gm.serientracker.bot.utils
In diesem Paket befindt sich eine Wrapper-Klasse, welche die Verbindung zwischen der Quartz Scheduler API und dem Clienten implementiert.

3.5.3 Zusatz: Bot-Client

Für das Veröffentlichen von Items, also den Notifications mit der Info über die bevorstehende Episodenaustrahlung, wurde ein Bot Client angelegt.
Der Bot-Client übernimmt dabei die Aufgabe pro Serie die Nodes anzulegen. Dazu scannt er die Datenbestand und wird zur jeder Serie drei Nodes anlegen, da ein User selbst entscheiden kann, ob er 5, 10 oder 15 Minute vorher informiert werden möchte. Die Node ID ist dabei im Format series:{Serien ID}:{[5|10|15]} definiert.
Gleichzeitig scannt er den Datenbestand nach den Episoden ab und prüft ob das Austrahlungsdatum in der Zukunft liegt.
Liegt es in der Zukunft wird ein Event angelegt. Dazu wurde die Quartz Scheduler Bibliothek[3] eingesetzt. Mit ihr ist es möglich sogenannte Cronjobs anzulegen. Die beiden entwickleten Jobs *ProfilerJob* und *NotificationJob* übernehmen dabei die Verarbeitung.

Aus zeitlichen Gründen konnte das Senden der automatisieren Notifcations nicht mehr implementiert werden. Zum Testen der Notifcation kann deshalb der XMPP Debug Client genutzt werden.

[3] `http://quartz-scheduler.org`

4 Projektreflektion

In der zweiten Projektphase des Sommersemesters 2013 im Modul Web-basierte Anwendungen 2, ging es um die Realisierung eines verteilten Systems. Zu Beginn dieser Dokumentation wurde das Konzept des SerienTrackers vorgestellt. Das wurden verschiedene Möglichkeiten zum Informationsaustausch angedacht und die verschiedenen Kommunikationsmöglichkeiten überlegt. Es folgte eine Vorstellung der einzelnen Entwicklungsphasen mit Ergebnissen, Ideen und einigen Alternativansätzen.

Wie bereits an manchen Stellen während der Ausarbeitung angesprochen, verlief die Entwicklung teilweise etwas anders als geplant. Zu Beginn stand bereits die Frage offen, ob die Einbindung eines Freundefeatures zeitlich realisierbar ist. Spätens im 3. Meilenstein des RESTful Webservice, wurde dieser Aspekt bereits zurückgestellt und auch andere Features woe Bewertungen oder Fehlermeldungen sind nach Ablauf der Projektphase nicht umgesetzt. Zugunsten zahlreicher Features viel die Konzentration deshalb auf einige wenige und diese wurden zur Vertiefung in den Fokus gerückt.
Im Laufe des Projektes machte sich der recht knappe Zeitplan deutlich bemerkbar,speziell bei den späteren Meilensteinen, bei welchen vor der Ausarbeitung der Grundlagenstoff vertieft werden muss. Eine kontinuerliche Beschäftigung mit den jeweiligen Inhalten ist in diesem Fall notwendig und vereinfacht einen Teil der Organisation. Ein simples Aufschieben von Aufgaben war auch aufgrund der Präsenztermine nicht möglich, was jedoch zum Fortschritt beitrug. Diese Herangehensweise war prinzipiell schon aus vorherigen Projekten bekannt, in diesem Fall aber unvermeindlich und soll auch in späteren Projekten angestrebt werden.
Die grafische Darstellung zur Codefrequenz auf GitHub, zeigt noch, dass innerhalb des Projektes speziell in der mittleren Phase deutlich mehr Potential steckte.

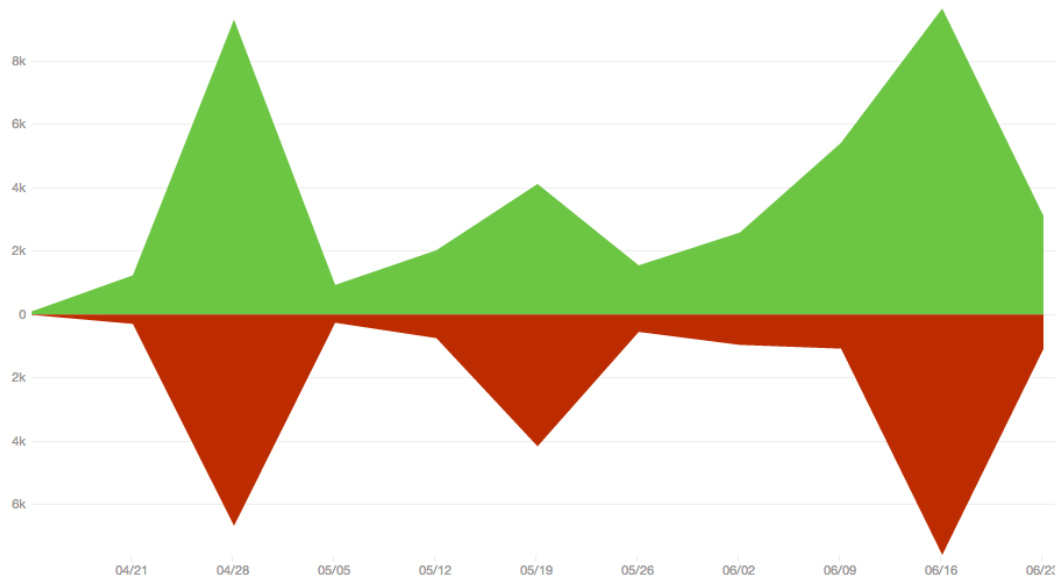


Abbildung 4.1: Codefrequency GitHub

Auch die Verwendung von GitHub als Versionskontrolle stellte zur Projektentwicklung eine gute Unterstützung dar. Austausch von Codes und anderen Dateien fand auf diesem Weg in einer sehr effektiven Form statt und soll für zukünftige Projekte erneut verwendet werden. Regelmäßige Absprachen und gemeinsames Arbeiten, förderte dabei den Lernfortschritt.
Was das finale Ergebnis angeht, so wurde keinesfalls das zuvor angestrebte Ergebnis erfüllt. Dieses war jedoch deutlich zu hoch angesetzt, was den Umfang anging und zwischendurch auftretende Probleme wirkten sich im Zeitrahmen als Hindernis auf.
Dennoch wurde sich den Anforderungen entsprechend mit den jeweiligen Projektzielen beschäftigt, sodass das die Lernziele umgesetzt und verstanden werden konnten.

In Anbetracht auf zukünftige Projekte, speziell das Entwicklungsprojekt interaktive Systeme im nächsten Semester, brachte dieses Projekte lehrreiches Wissen und zeigte Stellen auf, an denen noch mehr Potential steckt.

5 Literatur- und Quellenverzeichnis

- Stefan Tilkov: REST und HTTP, 2. Auflage 2011, dpunkt.verlag
- Bill Burke: RESTful Java with JAX-RS, 1.Auflage 2009, O'Reilly Verlag
- Peter Saint-Andre, Kevin Smith & Remko Troncon: XMPP The Definitive Guide, 1. Auflage 2009, O'Reilly
- A. Vohra, D. VohrA: XML Developement with Jaca Technology, 2006, Apress Verlag
- Codebeispiele und Links zur Phase 2 aus dem Medieninformatik Wiki als vertiefende Informationsquelle

Anmerkung zum Datenbestand:
Verwende Abbildungen innerhalb des Beispieldatenbestandes, entstammen der Seite http://trakt.tv/. An Bildquellen besteht kein Recht, doch ist kein Gebrauch über die private Nutzung im Rahmen dieses Projektes hinaus vorgesehen.

6 Autorenübersicht

Themenbereich	Dennis Meyer	Dominik Schilling
Konzept	50%	50%
Projektbezogene XML Schemata	70%	30%
Ressourcen und Semantik der HTTP - Operationen	50%	50%
RESTful Webservice	40%	60%
XMPP Konzeption	40%	60%
XMPP Cliententwicklung	30%	70%
Client	50%	50%

Abbildungsverzeichnis

Tabellenverzeichnis

Codeverzeichnis

Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.

Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsgbehörde vorgelegen.

Gummersbach, 23. Juni 2013

Dennis Meyer
Dominik Schilling