Dokumention zum „Partyplaner"

erstellt im Rahmen des Moduls WBA2 im SS 2013

ausgearbeitet von Ngoc-Anh Dang (1108269) und Tin Tran (11075432)

eingereicht am 23.06.2013, FH Köln Campus Gummersbach
betreut von Herrn Prof. Dr. Kristian Fischer und den WBA2-Mentroren

Inhaltsverzeichnis

Teil I.
Aufgabenfindung

Die Aufgabe, die im Rahmen des Moduls Webbasierte Anwendungen gelöst werden soll, besteht darin, eine Anwendung zu entwickeln, die auf einem verteilten System basiert. Die synchrone und asynchrone Kommunikationsabläufe sollen durch RESTful Webservices und XMPP realisiert werden und die Oberfläche mit Java-Swing. Im Folgenden werden die Vorgensweise, Probleme, die während der gesamten Entwicklung aufgetaucht sind und schlieSSlich das fertige Ergebnis vorgestellt.

Zuerst wird die Idee für die Anwendung und das Konzept der Kommunikationsabläufe und Interaktion vorgestellt. Dann werden die projektbezogenen XML-Schemata, welche die Kommunikationsgrundlage für die Anwendung darstellt, erläutert. Danach wird auf die Umsetzung der REST-Architektur eingegangen: Welche Ressourcen wurden definiert? Was ist die Semantik der HTTP-Operationen? Wie werden die URIs konzipiert? AnschlieSSend wird die Umsetzung der XMPP-Thematik behandelt. Darauf folgt der die Beschreibung der Implementierung des Clients, welcher den synchronen und asynchronen Informationsfluss zusammenführt und die funktionale Grundlage der GUI darstellt. Zuletzt wird ein Fazit angehängt: Es beinhaltet eine kritische Sicht des gesamten Projektes und beleuchtet mögliche Alternativen zu einigen Entwicklungsschritten.

Alle Kapitel dieses Dokumentes haben im groben denselben Aufbau: Anfangs wird kurz auf die Theorie eingegangen [1], darauf folgt eine Beschreibung des Prozessablaufes und einige Problemefälle, die währenddessen aufgetaucht sind.

[1] Fachbegriffe, die bereits im Rahmen der WBA-Phase 1 behandelt wurden, werden nicht noch einmal erläutert.

Teil II.
Entwicklung der Anwendung

1. Konzept der Anwendung

Dieses Kapitel beinhaltet die Planung der Anwendung, bestehend aus der Idee und eine Skizzierung der Kommunikationsabläufe. Zur Veranschaulichung sorgt ein Szenario, das von der erfolgreichen Benutzung der Anwendung erzählt, und ein Anwendungsfalldiagramm.

1.1. Inhalt der Anwendung

Als Ergebnis eines Brainstormings entsteht die Idee eines sogenannten Partyplaners. Er ist eine Anwendung, mit der ein Nutzer Vorlagen (im Folgenden als Themes bezeichnet) zu verschiedenen Veranstaltungen erstellt und mit anderen Nutzern teilt, sodass alle Nutzer diese bearbeiten, erweitern und anwenden können (s. Abb. 1.3). Ein Theme stellt ein Design-Muster für eine Feier dar, welche etwa aus Rezepten, Outfitvorschlägen, Dekorationvorschlägen, Musikvorschlägen etc. besteht. Themes werden unter verschiedenen Genres eingeordnet (z.B. Hochzeit, Konzert, Geburtstag, Gottesdienste, ...), welche wiederum in Kategorien unterteilt sind (z.B. Diamanthochzeit, Kindergeburtstag, Freilichtkonzert, ...). Jedes Theme muss mindestens einer Kategorie und dementsprechend auch mindestens einem Genre angehören (s. Kap.2 „XML-Schemata"). Zudem besteht die Möglichkeit Themes zu kommentieren, bewerten und zu abonnieren. Alle selbsterstellten Themes hat der Ersteller automatisch abonniert. Bei Veränderungen in den abonnierten Themes wird der Nutzer benachrichtigt.

Aus dem ersten Feedbackgespräch mit den Mentoren ergab sich, dass mehrere Abonnement-Möglichkeiten integriert werden sollen, um mehr asynchrone Kommunikationswege zu schaffen: Nun sind neben den einzelnen Themes, ebenfalls ganze Genres bzw. Kategorien abonnierbar. Dabei wird der Abonnent sofort in Kenntnis gesetzt, sobald neue Themes einem Genre/ einer Kategorie hinzugefügt werden (s. Kap. 1.3 „Interaktionen und Kommunikation").

1.2. Szenario

Das nachstehende Szenario soll für ein besseres Verständnis des Partyplaners sorgen. Es stellt einen erfolgreich ablaufenden Anwendungsfall dar, in dem ein Nutzer alle wichtigen Funktionen, der Anwendung benutzt:

Hans Egon möchte demnächst einen Junggesellen Abschied für seinen besten Freund veranstalten. Als unkreativer Mensch, der sonst auch nie feiern geht, hat er überhaupt keine Idee, was so zu einer coolen Party gehört, geschweige denn, wie man eine organisiert. Bei seiner ersten Recherche stöSSt er dann auf den Partyplaner. Als Hans Egon nach der Installation die Anwendung genauer anschaut, entdeckt er in der Navigation unter dem Reiter Genres Junggesellenabschied. Er wählt diesen aus, in der Hoffnung Inspirationen zu finden. Einige der Designs sprechen auch wirklich ihn an, welche er daraufhin abonniert, sodass er direkt eine Benachrichtigung erhält, sobald es Neuigkeiten oder Veränderungen an diesen gibt. Nachdem er sich einen Überblick über die vorhandenen Themes verschafft hat, versucht er nun die besten Ideen in einem eigenen Theme zu integrieren. Da sein bester Freund ein riesen Fan von Filmen ist, erstellt er ein Party-Theme mit dem Motto Hollywood. Bei der Erstellung fügt Hans Egon zur Musik die Songs der Lieblingsfilme und zu den Rezepten ein paar Lieblingsgerichte seines besten Freundes hinzu. Damit möglichst viele Nutzer sein Theme finden, um es mit mehr Inhalt zu füllen, ordnet er es sowohl unter Jungesellenabschied, als auch unter Mottoparty ein. AnschlieSSend speichert er das Theme ab. Daraufhin bekommt Hans eine Nachricht, in welcher ihm nochmal bestätigt wird, dass andere Nutzer nun darauf zugreifen können.

Johann benutzt den Partyplaner bereits seit zwei Monaten regelmäSSig. Da er auch ein groSSer Fan von Filmen ist, stöSSt er auf Hans Egons Theme. Sofort fällt ihm eine fantastische Idee für die Outfits ein: Jeder Gast kann sich als eine bekannte Filmfigur verkleiden! Diesen Vorschlag möchte er mit Hans Egon teilen und lädt zum Theme ein Foto von sich im letzten Karnevalskostüm als Captain Jack Sparrow unter Outfits hoch.

Hans Egon freut sich über jede Benachrichtigung über diverse Kommentare, Bewertunge und Vorschläge der anderen Nutzer. Darunter ist auch die Benachrichtigung über Johanns hinzugefügtem Foto. Die Idee findet Hans Egon klasse und bestätigt sie, sodass diese Erweiterung für alle seine Abonennten sichtbar wird. Als nach einer Woche das Theme, seiner Meinung nach, ausreichend ausgearbeitet wurde, lässt sich Hans Egon vom Partyplaner eine Checkliste generieren, wo er Schritt für Schritt alles Organisatorische abarbeiten kann. Dabei findet er es sehr hilfreich, dass die Liste die groben Kosten mit angibt.

1.3. Interaktionen und Kommunikation

Aus technischer Perspektive ist der Partyplaner so zu sehen: Der Nutzer schickt mit Hilfe des Clients Anfragen an den Party-Server, der Dienste zur Bedienung der Anfragen zur Verfügung stellt (Client-Server-Prinzip). Dabei sind sowohl synchrone, als auch asynchrone Interaktionen zwischen Client und Server möglich. Synchrone Kommunikation bedeutet der Client schickt eine Anfrage (Request) an den Server und wartet so lange, bis er eine Antwort (Response) erhält bevor er weiterarbeitet (Reqest-Reponse-Prinzip). Bei einem asynchronen Kommunikationsablauf wartet der Client nicht auf die Antwort des Servers, er gibt ihm lediglich Bescheid, dass er Benachrichtigt werden möchte, sobald der Server etwas liefern kann. Bei diesem Projekt sind folgende Kommunikationsabläufe synchron bzw. asynchron:

Synchron:

- Suchanfragen nach vorhandenen Themes, Genres oder Kategorien
- Bestätigungsmeldungen bspw. beim Abonnieren/ Abonemment kündigen
- Bearbeitungsfunktionen, Kommentarfunktion, Bewertungsfunktion

Asynchron:

- Benachrichtigungen bei Veränderungen der abonnierten Themes
- Benachrichtigungen bei neu hinzugefügten Themes in abonnierten Genres bzw. Kategorien

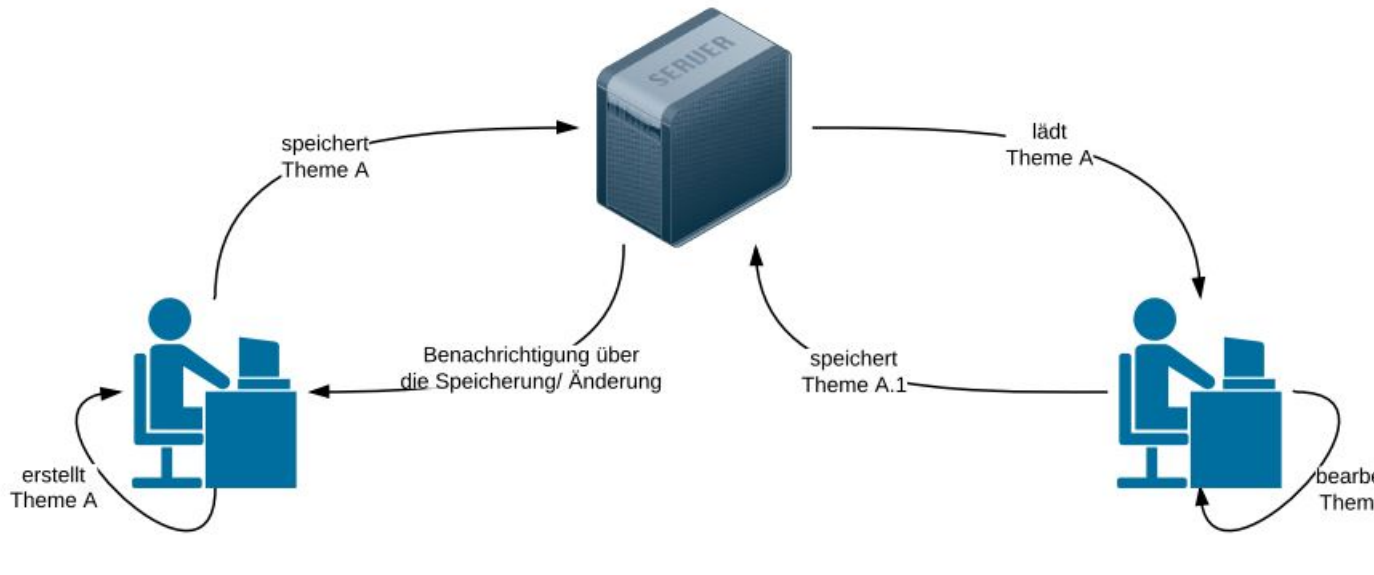

Abb. 1.1 Kommunikationswege Übersicht

Ein Nutzer erstellt ein Theme und speichert diesen auf dem Server ab. Ein Weiterer Nutzer lädt sich dieses Theme auf seinen Client und bearbeitet diesen. AnschlieSSend speichert er eine Kopie des ursprünglichen Themes ab. Der erste Nutzer bekommt eine Benachrichtigung über diese Änderung und kann diese Bestätigen. Die Änderung wird persistent und wird unter dem Namen des orginalen Themes für alle Nutzer sichtbar auf dem Server abgelegt [2]

Beim zweiten Feedback-Gespräch wurde angemerkt, dass zusätzlich zum obigen ein allgemeineres Diagramm erstellt werden soll. Daraus entstand Abb. 1.3 Die Einfärbung wurde noch nachträglich nach dem dritten Feedback-Gespräch hinzugefügt, damit optisch direkt erkennbar wird, welche Aktionen synchron (orange) bzw. asynchron (lila) ablaufen. Die durchgezogenen Verbindungen, sind Aktionen, welche die Akteure direkt ausführen können. Die gestrichelten Linien sind extends-Beziehungen (eine Aktion kann durch eine weitere erweitert werden). Zeigt eine gepunktete Linie von einer Aktion auf eine andere, setzt die eine die andere als Bedingung voraus. Richtet sich eine gepunktete Linie von einer Aktion auf mehrere andere Aktionen, gilt immer nur eine Bedingung.

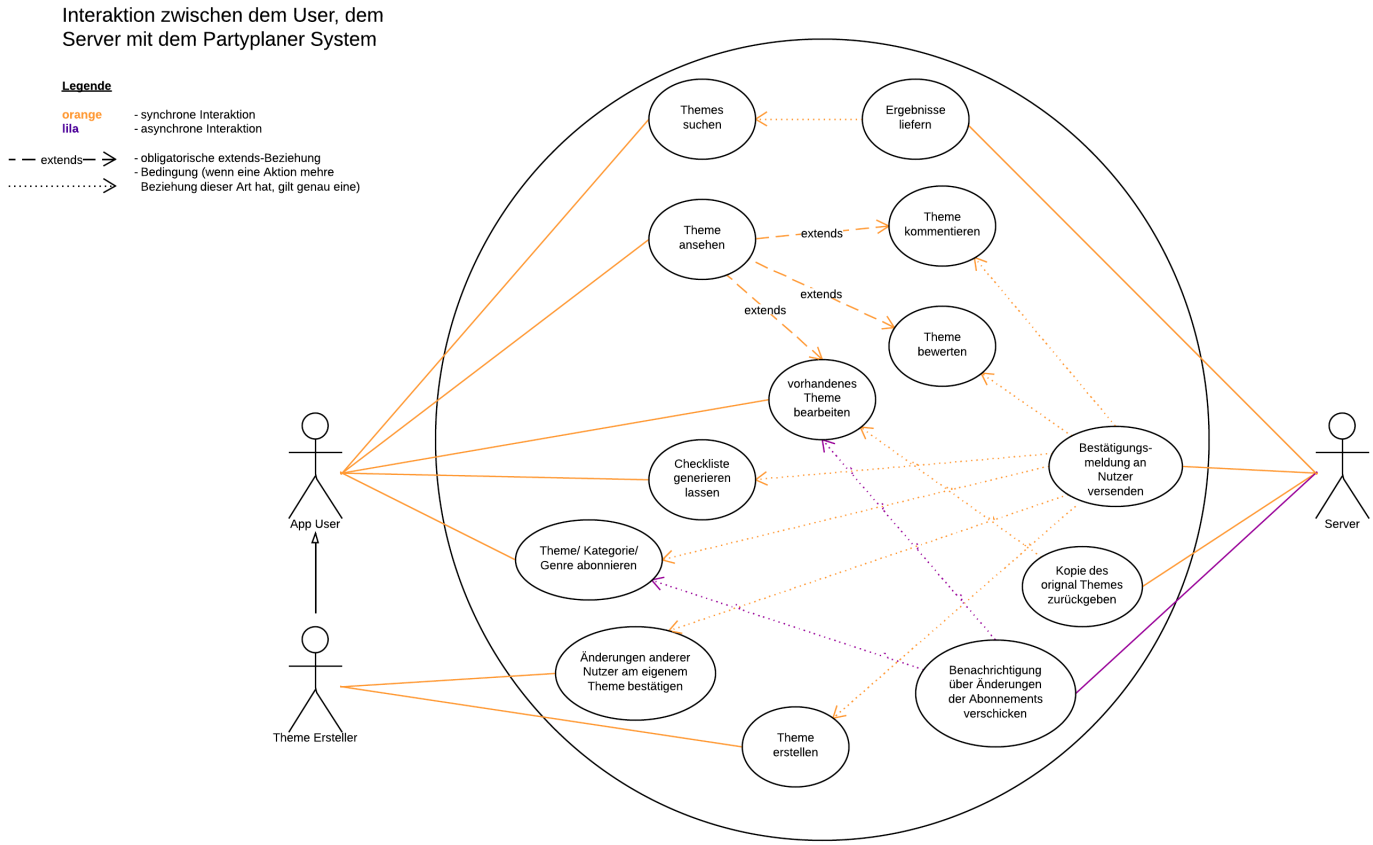

Abb. 1.2 Use-Case Diagramm

Der App-User hat die Möglichkeit im System Themes zu suchen, anzusehen, ein vorhandenes Theme zu bearbeiten, sich aus einem Theme eine Checkliste generieren zu lassen (FuSSnote: diese Funktionalität haben wir rausgenommen) und Abonnements abzuschlieSSen. Wenn sich ein App-User ein Theme ansieht, kann er optional anschlieSSend das Theme kommentieren (FuSSnote: Diese Funktion ist noch im XML-Schema und in der Datei integriert, jedoch nicht mehr im Client), bewerten oder bearbeiten. Als Theme-Ersteller kann man Änderungen anderer User an einem selbsterstelltem Theme Bestätigen. Der Server liefert zum Einen die angefragten Ergebnis-

[2] Diese Funktionalität wurde im späteren Verlauf der Entwicklung modifiziert, um das Konzept zu vereinfachen: Es wird nicht mehr unterschieden zwischen den eigenen Themes und der Themes anderer. Alle Themes, die abonniert wurden, darf man verändern

se für die Suchen der App-User und zum Anderen versendet er Bestätigungsmeldungen, sobald der App-User einen Kommentar gepostet, ein Theme bewertet, eine Checkliste generiert oder Abonnements abgeschlossen hat. AuSSerdem versendet der Server Benachrichtigungen, sobald Änderungen in den Abonnements des App-Users vorgenommen werden.

Nun muss noch geklärt werden, in welchem Format die Datensätze übertragen werden und wie diese sinnvoll strukturiert werden sollen. Folgende Inhalte sollen für ein Theme übertragen werden:

- Allgemeine Informationen (Titel des Themes , welches Genre, welche Kategorie,..)
- Module der Vorlage (Dekoration, Outfits, Musik, Catering und Location)
- Interaktionen (Bewertungen und Kommentare)

Wie die Daten im Genauen strukturiert werden, wird im nächsten Kapitel behandelt.

2. XML-Schemata

Ein XML-Schema (auch XSD XML Schema Definition) definiert die Struktur von XML-Dokumenten. Die XML-Dokumente werden die Daten speichern, die zwischen Server und Client hin und her geschickt werden (Am Schluss des vorigen Kapitel wird kurz erläutert, welche Daten das beim Partyplaner konkret sind.). Desweiteren werden später aus den Schemata Java-Klassen für die Implementierung der Services generiert (s. Kap.3.2 „Implementierung der Webservices").

2.1. Aufteilung der Daten

Bei der Erstellung der Schemata kommt die Fragestellung auf, ob es sinnvoll ist Genre und Kategorien, neben Theme als eigene XSD-Schemata auslagern, oder lieber alles in ein Schema zu verschachteln. SchlieSSlich existiert in diesem Konzept eine klare Hierarchie, die man gut in einem einzigen Schema umsetzen kann: Es gibt mehrere Genres, die wiederum mehrere Kategorien, die wiederum mehrere Themes zusammenfassen. Letztenendes wurden drei verschiedene Schemata geschrieben: einen für die Genres, einen für die Kategorien und einen für die Themes.

Aus dieser Entscheidung ergibt sich direkt die nächste Frage: Wie werden dann die Zusammengehörigkeiten und die Einordnung der Themes kenntlich gemacht? Drei Lösungswege stehen hierbei zur Auswahl: Erstens man löst das Problem über IDs, welche die Beziehung deutlichmachen, zweitens man gibt das Problem auf die Implementierungsebene weiter und legt in den Schemata nichts dazu fest, oder drittens der Zugriff auf die Elemente werden im Rahmen des Hypermedia-Prinzips geregelt. Der hierarchische Aufbau des Pfades lässt sich gut auf die Hierarchie der Genres, Kategorien und Themes abbilden. Tatsächlich wird das Problem durch eine Kombinationen der drei Möglichkeiten gelöst. Die Genres, Kategorien und Themes erhalten IDs, die bereits die Beziehung zueinander suggerieren. Diese werden im Code ausgewertet (Näheres dazu im Kap.3 „Webservice"). Zudem sind Lese- und Schreib- Operationen auf die XML-Dokumente nur durch bestimmte URIs möglich, welche eine klare Angabe der Hierarchie erfordern. Wie genau die URIs aufgenbaut werden, kann man im Kap. 3.1 „URI Design" nachlesen.

Grund für diese Entscheidungen sind, dass die Genres und Kategorien fest vorgegeben, also nicht von Nutzern erstellt, modifiziert oder löscht werden können, und daher in eigenen XML-Dokumenten gespeichert werden sollen. Dadurch wird bei der Benutzung der Anwendung vermieden, dass ein sehr langes Dokument durchforstet werden muss - ganz egal, ob wirklich etwas bearbeitet wird, oder ob nur Daten geholt werden müssen um Inhalte anzuzeigen. Durchforsten meint im Konkreten, dass nicht nur jedes Mal geunmarshallt und wieder gemarshallt werden muss, sondern es müssen aufgrund des hierarchischen Zusammenhangs der Genres, Kategorien und Themes sämtliche Schleifen und Vergleiche durchlaufen werden. Dies wirkt sich auSSerdem negativ auf die Laufzeit aus.

2.2. Allgemeines zum Aufbau der Schemata

Einige der Elemente, die häufig im Schema verwendet werden, werden ausgelagert, also global definiert, und stehen gesondert am Ende des Hauptschemas. Dies ist beispielsweise beim complexeType Beschreibung der Fall. Eine Beschreibung besteht aus einem Titel, einem Bild und einem Text. Sowohl ein Outfit, als auch eine Location kann mit einer solchen Beschreibung ausgestattet werden. Alle übrigen complexTypes werden lokal definiert, da sie dann keinen eigenen Namen bedürfen, weil an keiner anderen Stelle darauf zugegriffen werden muss [3].

Während der Implementierung der Services kam ein Problem auf, dessen Lösung verlangte, alle Types global zu definieren. Diese Änderung wurde jedoch aus Zeitgründen nicht mehr unternommen. Mehr zu diesem Problem im Kap. 3.3.2 „Problemstellungen während des Entwicklungprozesses".

[3] Laut Feedback aus der Phase 1 von WBA2 empfehlenswert.

2.3. Zum Schema der Themes

Für die complexTypes wurde sequence anstatt choice verwendet, um eine gewisse Reihenfolge zu gewährleisten. Im Grunde genommen spielt das für den Nutzer keine Rolle, denn eigentlich bekommt er die XML-Dateien nie zu sehen, da aus Zeitgründen keine komplexe Oberfläche erstellt werden kann, soll der Nutzer später XML-Schnipsel an den Client übergeben, die dieser wiederum an die Webservices weitergibt. Um Übersichtlichkeit und Ordnung zu gewährleisten erscheint es sinnvoll an dieser Stelle feste Reihenfolgen festzulegen. Die Angabe der Bewertung wird als Attribut anstatt eines Elements gespeichert, da diese Information vorwiegend als Metadatum gewertet wird, als Information für den Nutzer (Gut bewertete Themes sollen zuerst angezeigt werden, damit der Nutzer in der Fülle der Themes schnell etwas Gutes findet [4].

2.4. Zu den Schemata der Genres und Kategorien

Wie bereits oben erwähnt sind sowohl Genres, als auch Kategorien fest vorgegeben. Nun stellt sich wieder eine ähnliche Frage: Ist es denn nun sinnvoll diese Informationen in einer XML-Datei unterzubringen? Mit derselben Argumentation, wie auch im voherigen Abschnitt werden für Genres und Kategorien wiederum verschiedene Schemata geschrieben. Die Daten über die Kategorien können schneller ermittelt werden, wenn im Voraus nicht mehrere Schleifen und Vergleiche durchlaufen werden müssen, somit die beträgt die Laufzeit O(1).

[4] Diese Prioritätenauswertung wurde aus Zeitgründen nicht mehr implementert.

3. RESTful Webservice

REST - Representational State Transfer - bezeichnet ein Programmierparadigma für Webanwendungen. REST es baut auf 5 Grund-Konzepten auf:

- URI-Konzept: Es gibt Ressourcen mit einer eineutigen Identifikation.
- Hyperlink-Konzept: Verknüpfungen von Ressourcen über Hypermedia.
- Feste-Schnittstelle-Konzept: HTTP (HyperText Transfer Protocol) stellt Standardmethoden zur Verfügung.
- Repräsentationsvarianten-Konzept: Es gibt unterschiedliche Repräsentationen für Ressourcen, wobei die unabhängig von der Repräsentation ist.
- Zustandsloses-Konzept: Die Kommunikation ist statuslos.

Laut der Aufgabenstellung ist vorgeben, dass eine RESTful-Architecture verwendet werden soll, dennoch sollen nochmal kurz die Vorteile von REST umrissen werden. Es gibt vier Prinzipien,die REST beschreiben, die zugleich die Vorteile sind:

- Prinzip der losen Kopplung: Die Systemelemente sind modularisiert, sie arbeiten miteinander über Schnittstellen. Diese Isolation sorgt für eine bessere Wartung, Erweiterbarkeit und Sicherheit.
- Prinzip der Interoperabilität: REST nutzt genormte Kommunikationselemente (XML, URI, HTTP, HTML, ...) und ist somit fast überall einsetzbar.
- Prinzip der Wiederverwendung: Bei REST gibt es nur eine Schnittstelle, welche mehrere Clients nutzen können. Dies mindert auSSerdem die Kosten in der Softwareerstellung.
- Prinzip der Performance und Skalierbarkeit: Da REST die Infrastruktur des Web nutzt, ist es möglich Anfragen schnell zu beantworten. AuSSerdem verzichtet REST auf den sitzungsbezogenen Status, was bedeutet, dass aufeinanderfolgende Anfragen nicht vom gleichen System beantwortet werden muss.[5]

Alternative Ansätze für die Kommunikation zwischen verteilten Systemen zu REST sind etwa Sockets, Shared Memory, RPC, CORBA oder RMI.

Im Folgendem wird vorgestellt, wie REST in diesem Projekt umgesetzt wurden. Welche Ressourcen mit welchen Repräsentationen gibt es? Welche Standard-Methoden (HTTP-Methoden) greifen mit welcher Semantik auf die Ressourcen zu? Wie werden die URIs konzipiert? Daraufhin wird auf die Umsetzung des REST-Client und -Servers eingegangen.

3.1. Planung der REST-Architektur Ressourcen, HTTP-Methoden, URI-Design

Ressourcen sind die Elemente, die nach AuSSen getragen werden, dabei können sie verschiedene Repräsentationen haben. In diesem Projekt wurde mit zwei verschiedenen Repräsentationen gearbeitet - zum Einen mit einer reinen Textdarstellung (MIME-TYPE: text/plain) und zum Anderen mit XML (MIME-Type: application/XML). Andere mögliche Repräsentationen sind z.B. HTML, JSON oder PDF. In einem späteren Abschnitt wird nochmal näher darauf eingegangen, weshalb diese Repräsenationen gewählt wurden (s. Kap.3.2 „Implementierung der Webservices"). Die URI dient im WWW als eindeutige ID für die Ressourcen. Die HTTP-Methoden greifen über

[5] Stefan Tilkov: REST und HTTP - Einsatz der Architektur des Web für Integrationsszenarien. dpunkt.verlag 2011, 2. Auflage, S.1-4

die URIs auf die Ressourcen zu. Nachfolgend eine Übersicht zu den genannten Elementen:

Ressource	URI	Methode
Allgemein		
Liste aller Genres	/genres	GET
Genre	/genres/{genre_id}	GET
Liste aller Kategorien	/genres/{genre_id}/kategorien	GET
Kategorie	/genres/{genre_id}/kategorien/{kategorien_id}	GET
Liste aller Themes	/themes	GET , POST
Theme	/themes/{theme_id}	GET, PUT, DELETE
Liste aller User	/user	GET, POST
User	/user/{user_id}	GET
Themebezogen		
Liste aller Kommentare	/themes/{theme_id}/kommentare	GET, POST
Kommentar	/themes/{theme_id}/kommentare/{kommentar_id}	GET

Tab 3.1. Ressourcen, URI's und HTTP Methoden

Im GroSSen und Ganzen wird zwischen Primärressourcen und Listenressourcen unterschieden: Als Primärressourcen werden Genre, Kategorie, User und Kommentar definiert, da sie die Grundelemente für die Anwendung sind. Zu jeder Primärressource gibt es eine Listenressource - eine die Liste der Primärressourcen.

Die nachfolgende Tabelle beschreibt die Semantik der HTTP-Methoden auf die Ressourcen:

Methode	Ressource	Semantik
GET	alle	zeigt die Ressource an
POST	Liste aller Themes,Liste aller User, Liste aller Kommentare	ein neues Theme wird angelegt, ein neuer User wird registriert, User verfasst einen Kommentar zum Theme
PUT	Theme	das bearbeitete Theme wird überschrieben
DELETE	Theme	löscht ein Theme

Tab. 3.2 Semantik der Standard-Methoden

Genre, Kategorien und deren Listen haben KEINE PUT,POST und DELETE Methoden, um den Rahmen des Projektes nicht zu sprengen. Sind vom Entwickler Vorgegeben. Die Ressourcen User und Liste der User werden aus demselben Grund entfernt, da der Umfang des Projektes zu groSS wird.

Bezüglich des URI-Designs wurde zunächst entschieden, dass erstens die URIs der Primärressourcen immer aus dem Namen der Ressource und einer ID besteht (z.B. /genres/{genre_id}) und zweitens, dass die URIs der Listenressourcen alleine aus ihrem Namen bestehen (/genres). Wie zu sehen ist, wurde diese Entscheidung zurückgenommen. Nämlich ist die Hierarchie, die zwischen Gernes, Kategorien und Themes herrscht nicht deutlich gemacht. Der Zugriff über die URI soll bereits sicherstellen, dass nicht beispielweise auf eine Kategorie zugegriffen wird, welche gar nicht unter diesem Genre vorhanden ist (daher nun: /genres/{genre_id}/kategorien/{kategorien_id}). Bei den Themes wurde dies jedoch nicht umgesetzt, da auf Implementierungsebene zu viel Wirrwarr entstehen würde. Im Gegensatz zu der Kategorie, welche nur einem Genre angehören kann, kann ein Theme mehreren Kategorien und Genres angehören. Diese Prüfung über die Zusammengehörigkeit soll anhand der Daten in den Schemata geprüft werden. Dies vermeidet tief verschachtelte Schleifendurchläufe. Ein Gegenargument mag sein, dass der Nutzer durch die Struktur der URI und die der für ihn in der Anwendung sichtbaren Struktur verwirrt wird, jedoch ist an dieser Stelle dagegen zu argumentieren, dass die URI später für den Nutzer nicht sichtbar sein wird.

3.2. Implementierung der Webservices

Ein Webservice ist ein Dienst, der ein oder mehrere Server für Clients über das Web bereitstellt. Welche dies beim Partyplaner sind, wird in diesem Abschnitt vorgestellt. Vorerst wird jedoch auf allgemeine Aspekte eingegangen, die alle Services betreffen:

Als Grundlage für die Implementierung der Webservices dienen die validierten und auf Wohlgeformtheit geprüften XML- und XSD-Dokumente. Aus den Schemata werden mithilfe des XJC (von JAXB - Java Architecture for XML Binding - zur Verfügung gestellter binding compiler) Javaklassen generiert. Aus diesen Javaklassen können später Objekte instanziiert werden, dessen Daten aus den XML-Dateien gemarshallt (FuSSnote: marshalling: Umwandeln von strukturierten Daten in ein anderes Format, unmarshalling: umgekehrter Prozess) werden. Die @PATH-Annotionen legen fest, mit welcher URI auf welche Methode zugegriffen werden kann. die @GET-/@PUT-/@POST-/@DELETE-Annotationen weisen darauf hin welche Standardmethode in der nachfolgen Funktion umgesetzt wird. Die @Produces-Annotation besagt, welchem MIME-Type das erzeugte Objekt entspricht und die @Consumes-Annonation besagt, welchen MIME-Type die Eingabedaten für die Funktion besitzen müssen, damit die Funktion ausgeführt werden kann.

Die @PathParam-Annotation bezeichnet die Parameter in der URI, d.h. Teile der URI, die variieren können. In diesem Projekt werden die IDs der Genres, Kategorien und Themes als Path-Params mitübergeben. Somit wird eindeutig, welche Genre-, Katergorien- oder Theme-Ressource angesprochen werden soll.

Die @QueryParam-Annotation ermöglicht in URIs Anfragen mit Abfragen (Queries) aufzubauen, bspw. [...] ?genre_id=g0. Diese Mögichkeit des Aufrufs der StandardMethoden wurde anfangs integriert, später für den finalen Client wieder herausgenommen.

Für dieses Projekt wurden zwei WebServices impementiert: der GenresKategorienService und der ThemesService. Ursprünglich war noch ein dritter Service in Planung, der UsersService - wie auch der Liste der Ressourcen zu entnehmen ist (Kap.3.1). Dieser wurde jedoch aus Zeitgründen nicht mehr implementiert (hatte nur nebensächliche Funktionen. Der Kern, nämlich die Implementierung der synchronen und asynchronen Kommunikation, konnte auch gut ohne diesen Service umgesetzt werden. Der UserService sollte die Registrierten Nutzer verwalten und immer aufgerufen werden, sobald Daten über einen oder alle Nutzer gebraucht werden, z.B. wenn sich ein User registriert, abmeldet, etwas abonniert oder einen Kommentar verfasst.

3.2.1. GenresKategorienService

Der GenresKategorienService ist für die Verwaltung aller Daten der Genres und Kategorien zuständig. Hier eine Tabelle, die stichpunktartig darstellt, wie die HTTP-Methoden wie umgesetzt wurden [6]:

HTTP-Methode	Signaturen in Java	Beschreibung der Umsetzung
GET-Methode der Ressource Genres	getGenres();	Gibt eine Liste mit allen Genres zurück
GET-Methode der Ressource Genre	getGenre (String genre_id);	Gibt ein Genre-Objekt
GET-Methode der Ressource Kategorien	getKategorien (String genre_id);	Gibt eine Liste mit Kategorien zurück
GET-Methode der Ressource Kategorie	getKategorie (String genre_id, String kategorie_id);	Gibt ein Kategorie-Objekt

Tab. 3.3 Umsetzung der HTTP-Methoden im GenreKategorienService

[6] in der Endversion des Services, die Services für den KonsolenClient wurden anders umgestzt. Bei Bedarf kann man diese in Git unter Alt_GenresKategorienService bzw. Alt_ThemesService nachlesen.

3.2.2. ThemesService

Der ThemesService verwaltet nicht nur die Themes, sondern auch ihre Kommentare.

HTTP-Methode	Signaturen in Java	Beschreibung der Umsetzung
GET-Methode der Ressource Themes	getThemes();	Gibt eine Liste von allen Theme-Titeln als ein String zurück.
GET-Methode der Ressource Theme	getTheme (String theme_id);	Gibt Titel und Beschreibung eines Themes als String aus.
PUT-Methode der Ressource Theme	replaceTheme(String theme_id, Theme v_theme);	Verändert ein vorhanderes Theme.
POST-Methode der Ressource Themes	addTheme(Theme t);	Fügt ein theme in die Liste ein
DELETE-Methode der Ressource Theme	deleteTheme (String theme_id);	Löscht ein Theme aus der Liste
GET-Methode der Ressource Kommentare	getKommentare (String theme_id);	Gibt alle Kommentare zu einem bestimmten Theme aus.
GET-Methode der Ressource Kommentar	getKommentar (String theme_id, String kommentar_id);	Gibt einen bestimmen Kommentar zu einem Theme aus.
POST-Methode der Ressource Kommentare	addKommentar (String theme_id, Kommentar newKom);	Fügt einen neuen Kommentar zu einem Theme hinzu. Dabei wird die Nachricht erfasst, der Username des Verfassers und die Uhrzeit zu welcher der Kommentar gepostet wird.

Tab. 3.4 Umsetzung der HTTP-Methoden im ThemesService

In diesem Service gibt es noch eine Hilfsfunktion, nämlich setzeThemeDaten(Themes t);. Sie übernimmt das marshalling. Im GenreKategorienService existieren diese Hilfsfunktionen nicht, da nichts gemarshallt wird.

3.3. Jersey-Client und Grizzly-Server

Jersey ist ein Java Framework, mit dem in diesem Projekt der REST-Client entwickelt wurde. Grizzly ist das Framework, welches genutzt wurde um den HTTP-Server für den Partyclient einzurichten. Im folgenden Abschnitt wird vorgestellt, wie die Umsetzung des Clients und des Servers vonstatten gegangen ist. Für den Jersey-Client existiert eine Testklasse (RestClientTest), auf die diese Dokumentation jedoch nicht weiter eingehen wird. Sie testet lediglich jede HTTP-Operation auf jede Ressource.

3.3.1. Jersey-Client

Der REST-Client bietet Zugriff auf alle HTTP-Methoden. Bei einem Zugriff erstellt er jedesmal ein Webressourcen-Objekt, welches die geunmarshallten Objekte mit den Informationen aus den XML-Dateien zurückgibt. Je nach Anfrage baut der Client die URI dementsprechend mittels den Funktionen resource() und path() auf. AuSSerdem startet und beendet der REST-Client den REST-Server, sobald er instantziiert wird. Zudem stellt er die Funktion zur Verfügung den REST-Server zu beenden. Auf diese Methode wird später der Final-Client zugreifen. Der Final-Client führt den REST- und XMPP-Client zusammen. Wie und warum das funktioniert wird im

weiteren Verlauf erläutert (s. Kap.5 „PartyClient - Zusammenführung von REST- und XMPP-Services").

3.3.2. Problemstellungen während des Entwicklungprozesses

Während der Implementierung des Clients sind einige Problemstellungen aufgetreten: Das erste Problemstellung ist, dass die Umsetzung der Queryparameter für den finalen Client die Anwendung verkomplizieren würde. Bei der Umsetzung der Queryparameter wurden diese dazu genutzt, um IDs abzufragen und Primärressourcen anzusprechen, da diese Anwendung jedoch bereits GET-Methoden für die Primärressourcen enthält, würde diese weitere Funktion überflüssig werden (FuSSnote: zu dem Meilenstein, wo dies implementiert werden sollte, wurde dies auch getan und fehlerfrei vorgeführt. In den Dateien (Alt_GenreKategorienService und Alt_ThemesService sind diese noch enthalten.). Das zweite groSSe Problem ist, dass die PathParams nicht ausgewertet werden können, wenn mit ihnen ein Element angesprochen werden soll, das im Schema in einem complexType lokal definiert wurde. Bei der Fehlersuche wurde herausgefunden, dass das Unmarshalling von inneren Elementen nur funktioniert, wenn diese als root-Elemente im Schema gefunden werden können. Aus diesem Grund mussten die drei Schemata (KAP XY) nochmal umgeschrieben werden: Die complexTypes eines Genre, einer Kategorie und eines Themes werden nun global definiert, sodass die Definition dieses complexTypes nun auf derselben root-Ebene liegt, wie der root-Tag der Listenressource. An den Stellen, wo die complexTypes ursprünglich lokal definiert wurden, wird nur noch auf die globale Definition referenziert. Hier ein Beispiel dazu:

Ursprünglich:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="genres">
        <xs:complexType>
            <xs:sequence>
               <xs:element name="genre" minOccurs="1" maxOccurs="unbounded">
                     <xs:complexType>
                           [...Elemente eines Genre...]
                     </xs:complexType>
                     </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Jetzt:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" >
 <xs:element name="genres">
        <xs:complexType>
            <xs:sequence>
                <xs:element  ref="genre" minOccurs="1" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
        </xs:element>
</xs:schema>
```

3.3.3. Grizzly-Server

Der REST-Server, mittels des Grizzley-Frameworkes realisiert, wird direkt gestartet sobald eine Instanz erstellt wird. Die URL des Servers ist im Code fest einprogrammiert. Diese Stelle kann sicherlich noch verbessert werden, und zwar so, dass die URL dem Server-Objekt übergeben wird. Aus Zeitgründen ist dies jedoch noch nicht realisiert worden. AuSSderdem stellt der Server eine Methode zur Verfügung ihn zu terminieren. Diese Methode wird vom REST-Client ausgelöst.

4. XMPP

XMPP - Extensible Messaging and Presence Protocol - ist ein Kommunikationprotokoll für nachrichtenorientierte Middleware, die auf XML basiert. Wie auch bereits bei den RESTful Webservices wird zunächst erst einmal auf die Vorteile und Eigenschaften von XMPP eingegangen:

- *XMPP hat sich bewährt:* Hinter XMPP steht eine groSSe Community, die es stetig nutzt und weiterentwickelt. Dadurch wurde es bereits auSSerdem vielseitig getestet. Desweiteren wurde es vom IETF standardisiert.
- *Erweiterbarkeit und Interoperabilität:* Durch die vielseitige Nutzung wurde es weit über dem Instant Messaging (wo XMPP ursprünglich entstanden ist) hinaus weiterentwickelt und ist für viele weitere Anwendungsfälle nutzbar.
- *Dezentralisierung:* XMPP integriert im verteilten System mehrere Server und nutzt die Infrastruktur des World Wide Web (DNS - Domain Name System).
- *Sicherheit:* XMPP bietet unter anderem uthentifizierungsmöglichkeiten, Kodierungskanäle und Gegenwehr gegen vielen Formen von Malware.
- *Skalierbarkeit:* Mit dem Push-Modell hat XMPP Skalierbarkeitsprobleme gelöst, die bisher nicht lösbar waren.[7]

Zur Realisierung der asynchronen Nachrichtenübermittlung wird in diesem Projekt XMPP verwendet. Wenn ein Benutzer Abonnements abgeschlossen hat, soll er benachrichtigt werden, sobald es Neuigkeiten gibt (Push-Modell) - egal ob er währenddessen online ist, oder für eine Zeit offline war. Der XMPP-Server soll einen Publish-Subscribe Dienst zur Verwaltung der Abonnements des Nutzers anbieten. Über den XMPP-Client hat der Nutzer Zugriff darauf.

4.1. Open-Fire Server

Als XMPP-Server wird ein Openfire-Server eingerichtet. Auf diesem Server werden drei Benutzer angelegt: automatischerweise der admin und zusätzlich zwei Benutzer mit dem Namen user1 und user2 (die Passwörter sind jeweils adminadmin, user1user1 und user2user2). Die sonstigen Konfigurationen wurden so eingestellt, wie es aus dem YouTube Video der FH Köln Medieninformatik WBA2 - Phase 3 angeleitet wird. Mittels des Pidgin, ein Instant-Messenger, welcher XMPP-Server verwenden kann, wird sichergestellt, dass die User sich mit dem Server verbinden können.

4.2. Smack-Client

Der XMPP-Client baut auf dem Smack-Framework auf. Smack ist eine Java-Bibliothek, die Tools zur Verfügung stellt um XMPP-Clients zu bauen. Dieser Client soll dem Nutzer die Möglichkeit bieten den Publish-Subscribe Service zu nutzen. Er soll Topics, also Genres, Kategorien und Themes, subscriben (abonnieren) und unsubscriben können. Wenn ein Topic abonniert hat soll er über Neuigkeiten benachrichtigt werden.

Dafür muss für jedes Topic ein Node angelegt werden. Da in dieser Anwendung die Genres und Kategorien fest vorgegeben wurden, werden bereits bei der Initialisierung des XMPP-Clients die Funktionen des REST-Clients benötigt um Informationen aus den XML-Dokumenten zu extrahieren. Initialisieren bedeutet in diesem Zusammenhang zum Einen eben automatisch die LeafNodes für die Topics zu instanziieren und zum Anderen bereits vorhandene Topics zu finden (discover Nodes). Die LeafNodes für die Genres und Kategorien müssen einmalig beim aller ersten Start der Anwendung erstellt werden, danach werden sie bei jedem Start der Anwendung

[7] Perter Saint-Andre, Kevin Smith & Remko Troncon: XMPP - The Defiitive Guide. OREILY 2009, 1. Auflage, S.xi f.

vom ServiceDiscoveryManager wieder entdeckt (Daher ist dieser Teil im Code auskommentiert. Zum Debuggen, falls der Service nochmal ganz neu aufgesetzt werden soll, gibt es die Funktionen deleteAllNodes und unsubscribeAll, welche auch auskommentiert im Code vorhanden sind. Wurden die beiden Funktionen ausgeführt, muss die Erstellung der Genre- und Kategorie-Nodes wieder einkommentiert werden.)

Desweiteren werden die EventListener wieder hinzugefügt, da diese bei SchlieSSen des Clients immer gelöscht werden. EventListener sind Objekte, die bestimmte Funktionen ausführen, sobald Events gefeuert werden. Ein Event ist im Bezug auf dem Partyplaner z.B. wenn ein Theme geändert oder gelöscht wurde - eben Elemente, die beim publishen entstehen.

Zu Anfang steht erst einmal die Frage, ob CollectionNodes in den Service integriert werden sollen. Mittels CollectionNodes hat der Nutzer die Wahl eine Kollektion von Topics zu abonnieren. Da bei diesem Projekt die Funktion nicht angeboten wird, werden also keine CollectionNodes benötigt.

Nun wird genauer darauf eingegangen wie genau das zu einem Node Publishen beim Partyplaner aussieht: Bevor das Publishen implementiert wird gibt es einige konzeptionelle Fragen, die beantwortet werden müssen. Man kann beim Publishen, neben der reinen Mitteilung Es gibt etwas Neues., indem ein leeres Item versendet wird, auch ein Item mit Payload, also Nutzdaten, mitversenden (PayloadItem).

Hier ein Beispiel zum publishen mit und ohne Payload:

```
node.send(new Item());
node.send(new PayloadItem("test" + System.currentTimeMillis(),
          new SimplePayload("book", "pubsub:test:book", "")));
```

Das macht dann Sinn, wenn dem Nutzer zusätzlich mitgeteilt werden soll, wann, wo, was oder auch wie etwas verändert wurde. Das soll auch beim Partyplaner möglich sein. Werden beim Partyplaner Events gefeuert, wird der Payload der Items in einem Vektor mit dem Namen Benachrichtigungen gespeichert, sodass die GUI Informationen aus dem Payload anzeigen kann. Dabei ist zu beachten, dass der Payload in einer XML-Struktur angeben werden muss.

Eine weitere konzeptionelle Frage, welche beantwortet werden muss ist, ob die Items persistent gespeichert werden sollen. Mit persistenten Items gehen die Daten, die gepublisht worden sind, während der Nutzer nicht online war, nicht verloren. Der Nutzer kann die Nachrichten noch zu einem späteren Zeitpunkte erhalten. Auch diese Funktion soll im Partyplaner zur Vergügung stehen. Daher werden bei der Implementation die persistenten Items angestrebt.

Was die Vorraussetzung dafür ist, wird in einem weitereren wichtigen Aspekt für die reibungslose Benutzung deutlich: Die korrekte Konfiguration der Nodes. Man muss folgende Konfiguration einstellen: setPersistentItems(true);. Es gibt noch viele weitere Konfigurationen, jedoch wird nicht weiter auf andere eingegangen. Zur Bestimmung der notwendigen Konfigurationen dienen Informationen aus der Javadoc zur Smack API als Grundlage. In dieser Anwendung unterscheiden sich die Konfigurationen zwischen den Genre-, Kategorie- und Theme-Nodes nicht, da sie sich nicht viel in der Benutzung des publish/subscribens unterscheiden.

In diesem Dokument wurde bereits desöfteren erwähnt, dass der Nutzer auch dann benachrichtigt werden soll, sobald ein ganzes Theme gelöscht wird. Dazu werden ItemDeleteListener benötigt. Im Prinzip funktionieren sie genauso wie die ItemEventListener. Aus Zeitmangel konnten diese jedoch nicht umgesetzt werden.

4.3. Problemstellungen während des Entwicklungprozesses

Bei der Implementierung des Publish-Subscribe Dienstes sind einige Probleme aufgetreten: Das erste Hindernis war, dass Smack zwei Funktionen für das publishen zur Verfügung stellt, nämlich einmal publish() und einmal send(). Der Unterschied zwischen den beiden Funktionen ist, dass send() ein synchroner Funktionsaufruf und publish() ein asynchroner Aufruf ist. Bei send() wartet der Client auf die Benachrichtigung des Servers, ob er die published Items empfangen hat.

Bei publish() wartet der Client nicht auf die Antwort des Servers und fährt fort. Beim ersten Feedback-Gespräch zu dieser Problemstellung wurde von den Mentoren empfohlen publish() zu nutzen. Nach weiteren Gesprächen mit Kommilitonen und nochmals mit den Mentoren, wurde dies dann umgestellt. Mit dem von Smack zur Verfügung gestellten Debugger wurde ein weiterer Konflikt entdeckt [8]: Die ganze Zeit über wurde mit zwei verschiedenen JIDs (Jabber Idenfiers) gearbeitet. Teilweise wurde mit einer anderen JID abonniert, als gepublisht wurde. JIDs dienen als Adresse, womit Elemente beim XMPP angesprochen werden. Eine JID hat folgenden Aufbau: nameDesElemets@nameDesXMPPServers. Die eine JID, die im Server selbst für User generiert hat lautete user1@localhost/Smack. Die implementierte JID lautete user1@localhost. Der Fehler wurde folgendermaSSen behoben:

vorher:

```
private String getMyJID()
        {
                String id = Login.getUser() + "@" + PartyClient.con.getHost();
                return id;
        }
```

nachher:

```
private String getMyJID()
        {
                String id = PartyClient.con.getUser();
                return id;
        }
```

Der o.g. Debugger bietet die Funktion alle Stanzas mit zu lesen, die zwischem dem Client und dem Server hin und her geschickt werden. Stanzas sind die Nachrichten in einer XML-Struktur, die sich Client und Server zur Kommunikation hin und her schicken. Es gibt drei Arten von Stanzas <message/> als eine Nachricht, <presence/> zur Benachrichtigungen über Anwesenheiten und<iq/> für diverse Anfragen. Beim publishen ist folgender Nachrichtenaustauch beobachtbar: Der Client verschickt zum Server die Anfrage für einen Publish:

```
<iq id="2s674-54" to="pubsub.localhost" type="set">
  <pubsub xmlns="http://jabber.org/protocol/pubsub">
    <publish node="g0">
      <item>
        <payload>enter data here!</payload>
      </item>
    </publish>
  </pubsub>
</iq>
```

Der Server gibt dem Client Bescheid, dass er diese Anfrage erhalten hat:

```
<iq id="2s674-54" to="user1@localhost/Smack" from="pubsub.localhost" type="result
```

Der Server feuert daraufhin ein Event (!) :

```
<message id="g0__user1@localhost__9MpJQ" to="user1@localhost/Smack" from="pubsub.
  <event xmlns="http://jabber.org/protocol/pubsub#event">
    <items node="g0">
      <item id="4d010a3c-2b1c-4bdd-b9c4-733b1e0d336b1"/>
```

[8] Um den Smack Debugger zu starten, ist die entsprechende Zeile in der Klasse des PartyClients wieder einzukokmmentieren: XMPPConnection.DEBUG_ENABLED = true;

```
    </items>
  </event>
  <headers xmlns="http://jabber.org/protocol/shim">
    <header name="pubsub#subid">6Nl3ckH3pIhd49zpU8OlT4gUQMw6qF4KMG3hjuyh</header>
  </headers>
</message>
```

Trotzdessen, dass dieses Event laut dem Stanza gefeuert wird, reagiert der ItemEventHandler keineswegs. Als Übergangslösung wird erstmal die handlePublishItems manuell innerhalb der publish-Methode aufgerufen - der Tatsache bewusst, dass dies nicht der Idee des Publish-Subscibe-Modell entspricht. Nach einer weiteren Änderung kann endlich eine Reaktion hervorgerufen werden:

vorher:

```
discovery_items = discovery_mgr.discoverItems( "pubsub.localhost" );
```

nachher:

```
discovery_items = discovery_mgr.discoverItems( "pubsub." + PartyClient.con.getHos
```

Obwohl, wenn man sich PartyClient.con.getHost()) ausgeben lässt, genauso localhost ausgegen wird, reagiert der Handler nur, wenn man den Überrgabewert für den DiscoveryManager auf diese Weise übergibt. Nun reagiert der ItemEventHandler zwar auch ohne ihn manuell aufzurufen, jedoch zeigt er die Benachrichtigungen nie beim ersten Mal. Bei einem weiterem Publish wird dann der erste und der zweite angezeigt und jeweils dabei immer mit zwei Benachrichtigungen. Kurz: Beim ersten Mal publishen erscheint nichts, beim zweiten Mal publishen erscheinen gleich vier Benachrichtigungen.

Desweiteren kommt keine Benachrichtigung mehr beim Publishen an, wenn der Nutzer ein Theme verändert oder neu erstellt, was noch nicht der Fall war, als die Handler-Funktion noch manuell aufgerufen wurde.

Ein weiteres ungelöstes Problem zurück: Zwar reagiert der ItemEventHandler nun, und die Benachrichtigungen werden in der GUI angezeigt, jedoch geschieht dies nur in der Oberfläche des Nutzers, der gepublisht hat. Ist ein zweiter User online, der dasselbe Topic abonniert hat, erhält er die Benachrichtigung nicht. Dabei wird in der Node-Konfiguration eingestellt, dass die Items persistent gepeichert werden sollen. Dementsprechend erhält der Nutzer auch keine Benachrichtigungen, falls welche gepublisht wurden während er offline war. Aufgrund der begrenzten Zeit, wurde die Ursachen für diese Probleme noch nicht entdeckt und diese Probleme konnten noch nicht gelöst werden.

5. PartyClient - Zusammenführung von REST- und XMPP-Services

In diesem Kapitel wird der finale Client vorgestellt - der PartyClient. Er soll die Schnittstelle für die GUI sein, um sowohl die REST-Services als auch die XMPP-Services zu nutzen. Zuvor wird jedoch auf den Konsolen-Client eingegangen, der implementiert wurde um einfacher die Methoden für den finalen Client zu entwickeln.

5.1. Konsolen-Client

Bei der Entwicklung der Methoden wurde zunächst ein Client mit einem Menü in der Konsole geschrieben (KonsolenClient). Auf diese Weise ist es einfacher Fehler in den Services zu finden und herauszufinden wie, welche Funktionen im Final-Client sinnvoll sind. Der groSSe Unterschied zum finalen Client ist, dass hier nur mit Strings gearbeitet wird, anstatt mit JAXB-Objekten. Dementsprechend wurde im WebService die Einstellungen @Produces(text/plain) bzw. @Consumes(text/plain) vorgenommen.

Nachdem ungefähr ersichtlich wurde, wie der Zugriff auf die Services aussehen müssen, wurde der finale Client implementiert, welcher entsprechend mit JAXB-Elementen arbeitet.

5.2. PartyClient

Der PartyClient bildet die funktionale Grundlage für die GUI. Die GUI ruft bestimmte Anfragen an die Services auf, die der PartyClient an den REST-Client und den XMPP-Client delegiert. Diese Trennung der Zuständigkeiten in verschiedene Klassen ist zum Einen für die Entwicklung angenehmer, zum Anderen sorgt diese Modularisierung für eine bessere Übersichtlichkeit, was unter anderem bei der Wartung des Systems von Vorteil ist.

Eine Alternative ist anstatt einen Client zu benutzen, der zwei unter einen Hut bringt, direkt die GUI über zwei Clients zu bauen. Jedoch wurde dagegen entschieden. Der erste Grund ist der, der Übersichtlichkeit: für die GUI ist es unkomplizierter Funktionen von einer Klasse aufzurufen, als Funktionen von zwei verschiedenen Klassen. Der zweite Grund ist, es gibt Funktionen, die ein Client unabhängig von den verschiedenen Services verwalten muss, etwa der Verbindungsauf und -abbau, und Funktionen, die die Ergebnisse der Services noch weiterverarbeiten (z.B. getGenresTitles()). Hierbei sind zwei etwas gröSSere Funktionen zu erwähnen, die der Partyclient übernehmen muss: die Umsetzung der PUT- und der POST-Methode.

Die Methoden nehmen die Eingabedaten der GUI entgegen. Dann erstellen bzw. modifizieren sie die betroffenen JAXB-Elemente so weit, dass sie dem REST-Client übergeben können. Der REST-Client nimmt die vorbereiteten Objekte an und führt schlieSSlich die HTTP-Methoden dazu aus. Aufgrund der bereits mehrmals erwähnten Schemata-Probleme ist die Umsetzung dieser beiden Methoden etwas unsauber. Dies bedeutet, dass manche Informationen nicht übergeben bzw. erstellt werden können. Bspw. falls mit der POST-Methode dem Theme ein Genre zugeordnet werden soll:

So sollte es aussehen:

```
t.getAllgemeines().getGenres().getGenre().add(g_id);
```

Allerdings verlangt die add-Methode folgende Parameter:

```
add(JAXBElement<String> e)
```

6. GUI mit Java-Swing

In diesem Kapitel wird zuerst die Planung der Umsetzung der Oberfläche für den Partyplaner vorgestellt und dann welches Ergebnis mit Java-Swing erzielt wurde. Java-Swing stellt eine Grafikbibliothek zum Programmieren von grafischen Oberflächen zur Verfügung

6.1. Planung

Zur Planung der Oberfläche wird zunächst in einem Papierprototyp die Gliederung und Struktur grob konzipiert.

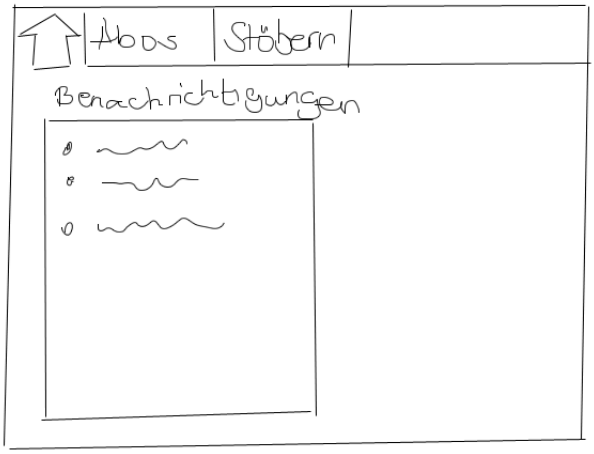

Abb. 6.1 Benachrichtigungen Fenster

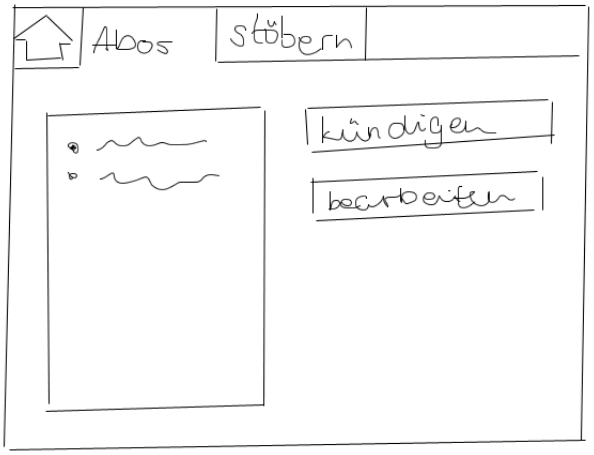

Abb. 6.2 Abonnement-Verwaltung Fenster

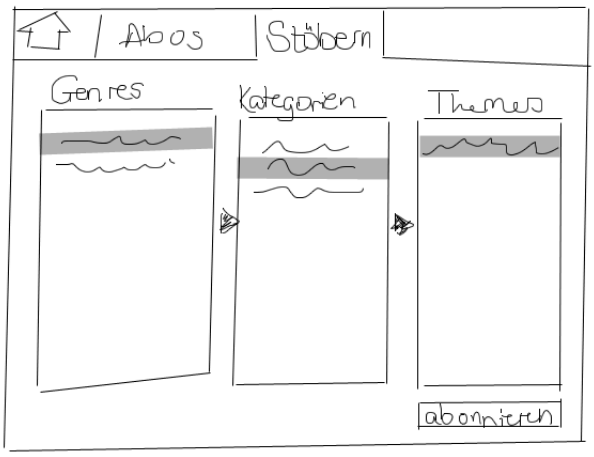

Abb. 6.3 Übersicht Fenster

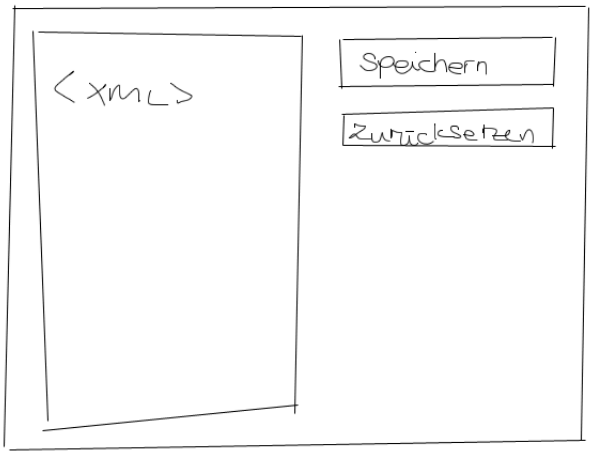

Abb. 6.4 - Theme bearbeiten/ erstellen Fenster

Wie auf den Abbildung erkennbar wird, ist die GUI folgendermaSSen aufgebaut:
Hauptmenü soll als Tabsystem strukturiert werden:

1. Tab: Dies ist die Startseite nach dem Einloggen der Anwendung. Der Nutzer hat eine Übersicht über Benachrichtigung seiner Abonnements.

2. Tab: Hier kann der Nutzer seine Abonnements einsehen und ggf. kündigen.

3. Tab: Hier kann der Nutzer nach neuen Abonnements Stöbern. Von links nach rechts ist eine Auflistung von den Genres, Kategorien und Themes. Abhängig davon, welches Genre der Nutzer auswählt, werden die passenden Kategorien angezeigt und abhängig von der Auswahl einer Kategorie werden die passenden Themes angezeigt.

4. Tab: Hier kann der Nutzer ein Theme bearbeiten oder ein neues erstellen. Zu sehen ist ein Editor, in welchem die Daten eingegeben werden sollen

6.2. Umsetzung

Bei der Umsetzung der Oberfläche kommen noch einige Aspekte hinzu: Das LogIn-Fenster, verschiedene Pop-Ups und ein Hilfe und Logout Button. Das LogIn-Fenster wird geöffnet sobald die Anwendung gestartet wird. Die LogIn-Daten werden dann dem PartyClient übergeben, der anhand dieser Daten die Verbindung zum XMPP-Server aufbaut. Die Pop-Up Fenster benachrichtigen den Leser ob eine Operation erfolgreich oder nicht erfolgreich durchgeführt wurde oder warnt

den Nutzer bei unzulässigen Operationen (bspw. wenn er den Abonnement kündigen-Button betätigt, ohne ein Topic ausgewählt zu haben.). Desweiteren gibt es ein spezielles PopUp-Fenster, das in einer eigenen Klasse realisiert wurde, nämlich das Fenster, in welchem die Informationen für jedes Theme angezeigt werden (ThemeInfo). Die Informationen werden aus den XML-Dateien geunmarshallt und in einem konkattiniertem String an Label-Elemente zurückgegeben. Ein Fehler in diesem Zusammenhang konnte noch nicht behoben werden: Manche Informationen werden als null angezeigt, obwohl diese in der XML-Datei diese Daten enthält. Das seltsame dabei ist, dass dieselben Informationen noch ausgegeben werden konnten, als die HTTP-Methoden noch text/plain zurückgegeben hat. Diese Tatssache lässt vermuten, dass dieses Problem dieselbe Ursache hat, wie das Problem mit den PathParams (s. Kap.3.3.2 „Problemstellungen während des Entwicklungprozesses"), nämlich, dass beim Unmarshalling Informationen nicht lesbar werden.

Der Hilfe- und Logout-Button sind fix, d.h. sie sind immer sichtbar, egal in welchem Menü sich der Benutzer befindet.

AuSSerdem wurde das Theme bearbeiten/erstellen - Menü etwas detailreicher gestaltet. Es gibt ein Dropdown-Menü, wo der Nutzer wählen kann, ob er ein abonniertes Theme modizifieren oder ein neues Theme erstellen möchte. Wählt er ein vorhandenes Theme aus, werden die Inhalte in dem Editor angezeigt. Der Editor weist neuerdings Tabs auf. Die Tabs stehen für die Module, aus welchen ein Theme besteht. Wählt der Nutzer die Option Theme erstellen aus, erscheint ein weiteres Dropdown, wo er dem neuem Theme ein Genre hinzufügen kann. Wählt er nun ein Genre aus, kann er wiederum eine passende Kategorie auswählen. An dieser Stelle wurde noch nicht realisiert, dass ein Theme zu mehreren Genres/Kategorien angehören kann. Desweiteren gibt es einen Zurücksetzen-Button, der die Inhalte des Editors löscht. Der Speichern-Button würde theoretisch ein publish auslösen, dies ist jedoch noch nicht umgesetzt.

Auch im Menü der Abonnements gibt es eine Neuerung: Die Abonnements sollten eigentlich in einem Dateiverzeichnis angezeigt werden, jedoch ist dies auch noch nicht umgesetzt worden. Ein weiteres Menü, das nachträglich hinzugefügt wurde, ist das Publish-Menü. Eigentlich ist dieses Menü kein Bestandteil des Partyplaners, es wurde lediglich hinzugefügt, da die Funktionen, die einen publish auslösen aus Zeitgründen noch nicht implementiert werden konnten (PUT und POST). In einem Dropdown-Menü kann man hier auswählen zu welchem abonniertem Topic man publishen möchte. Im darunter liegenden Editor kann man einen Payload mitgeben. Im Quellcode wird der Payload dann in eine XML-Struktur gepackt und als Item an den Server geschickt.

Teil III.
Fazit und Ausblick

Fazit

Zusammenfassend kann gesagt werden, dass zwei groSSe funktionale Aspekte und einige kleinere Design-bezogene Aspekte noch Ausarbeitung bedürfen.

- Publishing: Das Publishing, als einer der Hauptfunktion der Anwendung, funktioniert noch nicht wie gewünscht. Die Nutzer sollten immer benachrichtigt werden, ganz egal wer etwas gepublisht hat oder wann. AuSSerdem sollte der Nutzer erkennen was, wann und von wem geändert wurde, was bedeutet, dass der Payload nicht mehr vom Nutzer eingegeben wird, sondern vom Client erstellt wird.
- XML-Daten: Vermutlich sind die XML-Schemata nicht JAXB-gerecht aufgebaut worden, dass nicht alle Daten ausgelesen und wieder eingespeichert werden können.
- Ausgeblendete Komponenten der GUI verschieben die bereits eingeblendeten Elemente, wenn sie selbst eingeblendet werden. Dies ist ein layoutspezifisches Problem und kann vermutlich damit gelöst werden, dass das Layoutsystem eingeführt wird. Eine Alternative stellt z.B. ein Layout dar, welches absolute Koordinaten vorgibt.
- Die Abonnements werden noch nicht wie gewünscht in einer hierarchischen Struktur angezeigt. Dies ist sicher mit einer geschachtelten Schleife umsetzbar, welche zuerst die Beziehungen der Topics zueinander prüft und daraufhin die Reihenfolge erstellt.

Im GroSSen und Ganzen sind in der Idee und im Szenario zu Anfang viel mehr erwähnt worden, als tatsächlich umgesetzt wurde - wie z.B. die Checkliste, welche man sich generieren lassen kann. All diese Aspekte könnten noch aufgenommen werden, aber bedauerlicherweise stand nicht mehr Zeit zur Verfügung, sodass zwar eine lauffähige, jedoch nicht dem vollen Umfang an Funktionen entsprechende Anwendung entwickelt wurde.

Zuletzt noch kurze Anmerkungen zum Formalen: Leider ermöglichte uns die Software nicht in gegebener die Seitenzahlen dieses Dokumentes zu ändern (Deckblatt). AuSSerdem werden aus unbekannten Gründen weder das Logo, noch das Abgabedatum angezeigt werden.

Literatur

[1] Stefan Tilkov: REST und HTTP - Einsatz der Architektur des Web für Integrationsszenarien. dpunkt.verlag 2011, 2. Auflage, S.1-4

[2] Perter Saint-Andre, Kevin Smith & Remko Troncon: XMPP - The Defiitive Guide. OREILY 2009, 1. Auflage, S.xi f.