



Fachhochschule Köln  
Cologne University of Applied Sciences

## Web-basierte Anwendungen 2: verteilte Systeme Dokumentation Phase 2

Laura-Maria Hermann  
11083968  
David Ferdinand Petersen  
11083571

23. Juni 2013

Dozent: Prof. Dr. Fischer

Betreuer: Renée Schulz, David Bellingroth und Christopher Messner

# Inhaltsverzeichnis

<b>I Einleitung</b>	<b>1</b>
<b>II Hauptteil</b>	<b>1</b>
<b>1 Vorbereitung</b>	<b>1</b>
1.1 Szenarioerstellung . . . . .	1
1.2 Visualisierung des Szenarios . . . . .	3
1.3 Kommunikationsabläufe . . . . .	9
1.4 Funktionen des Systems: . . . . .	10
<b>2 XML Schema</b>	<b>11</b>
2.1 Datentypen . . . . .	14
2.2 Attribute . . . . .	15
2.3 Struktur des Schemas . . . . .	16
<b>3 RESTful Webservice</b>	<b>16</b>
3.1 Ressourcen . . . . .	17
3.1.1 Findung von Ressourcen . . . . .	17
3.1.2 Eindeutiger Identifier und die Verbindungen zu Ressourcen	19
3.1.3 Implementierung der Ressourcen . . . . .	33
3.2 Verben . . . . .	33
3.2.1 GET . . . . .	33
3.2.2 HEAD . . . . .	34
3.2.3 PUT . . . . .	34
3.2.4 POST . . . . .	34
3.2.5 DELETE . . . . .	35
3.2.6 Weitere HTTP-Verben . . . . .	35
3.2.7 Zusammenfassung . . . . .	35
3.3 Realisierung des RESTful-Webservice Systems . . . . .	36
3.3.1 RESTful-Webservice . . . . .	37
<b>4 XMPP</b>	<b>40</b>
4.1 Recherche und Vorbereitung[6] . . . . .	40
4.1.1 Architektur: . . . . .	40
4.1.2 Streaming: . . . . .	40
4.1.3 Fat-Ping versus Light-Ping [5] . . . . .	41
4.1.4 Leafs . . . . .	42
4.1.5 Publisher versus Subscriber . . . . .	43
4.2 Realisierung des XMPP-Server Systems . . . . .	44
4.2.1 Benutzerrollen und Leaf-Nodes . . . . .	44
4.2.2 Probleme bei der Umsetzung des XMPP Servers . . . . .	46
4.2.3 Kombination: REST- und XMPP-Server . . . . .	47

4.2.4	Zusammenstellung der Leaf-Nodes . . . . .	49
4.3	GUI Entwicklung . . . . .	50
4.3.1	Prototyp . . . . .	51
4.3.2	Realisierung und Umsetzung des GUI . . . . .	53
4.3.3	Oberfläche des Interessenten . . . . .	54
4.3.4	Oberfläche des Veranstalters . . . . .	54
4.3.5	Probleme . . . . .	55
<b>5</b>	<b>Fazit und Ausblick</b>	<b>56</b>

# **Teil I**

## **Einleitung**

In dieser Dokumentation befinden sich die ausgearbeiteten Inhalte der Phase 2 des Moduls Web-basierte Anwendungen 2: verteilte Systeme. Die Modellierungs- und Codeaufgaben befinden sich im Java-Projekt des Repositorys, die schriftlichen Ausarbeitungen sind in dieser Dokumentation zu finden.

Zu Beginn war die Festlegung der Aufgaben, Ziele und persönlichen Erwartungen erforderlich. Im Groben bestand die Aufgabe darin, eine verteilte Anwendung mit synchroner und asynchroner Kommunikation zu ermöglichen.

# **Teil II**

## **Hauptteil**

### **1 Vorbereitung**

Im Folgenden werden Vorbereitungen und Rechercheergebnisse dokumentiert. Diese wurden im Rahmen der Meilensteine erarbeitet.

#### **1.1 Szenarioerstellung**

Zu Beginn, war es die Aufgabe ein konkretes Szenario zu entwickeln, welchem eine Problemstellung zugrunde liegt. Jene Problemstellung soll im Verlauf der zweiten Phase des Workshops mit Hilfe einer Anwendung gelöst werden. Hierbei ist drauf zu achten, dass sowohl synchrone als auch asynchrone Datenübertragung ermöglicht und angewendet wird.

Bei der synchronen Übertragung handelt es sich um RESTful Webservices, also beispielsweise gestellte Anfragen, auf dessen Antwort explizit gewartet wird. Dazu sollen im Laufe der Projektarbeit zunächst Datenobjekte identifiziert werden, aus dessen daraufhin Ressourcen gebildet werden. Auf diesen Ressourcen können dann Operationen im Sinne der synchronen Datenübertragungsart stattfinden (GET, PUT, POST, DELETE,...).

Bei der asynchronen Datenübertragungsart hingegen, wird vorwiegend die Publish-Subscribe Methode verwendet. Im Gegensatz zum synchronen Verlauf, wird an dieser Stelle nicht explizit auf die Antwort des Servers gewartet, sondern diese trifft in unbestimmter Zeit „irgendwann“ ein. Diese Art der Informationsbeschaffung ist deutlich effizienter, als bei der synchronen, da einerseits nicht zwingend auf eine Antwort gewartet werden muss (Zeitunabhängigkeit), andererseits, nicht ständig neue Anfragen gestellt werden, welche beispielsweise versuchen in regelmäßigen Zeitabständen neue Informationen zu erhalten (Resourcesparungen).

Im Folgenden wird das gewählte Szenario mitsamt seiner Problemstellung beschrieben und erläutert. Zunächst wurden mögliche Szenarien grob betrachtet und in Frage kommende ausgearbeitet. Nach der detaillierten Erarbeitung zweier Szenarien wurde Pro und Kontra abgewägt, sodass sich letztendlich für das "Sport-Szenario" entschieden wurde.

Die zugrundeliegende Problemstellung betrifft das Sportangebot und Sportveranstaltungen. So ist es, besonders in einer Kleinstadt interessant einen komfortablen Überblick über laufende Sportveranstaltungen zu erhalten. Dabei wird zunächst nicht explizit zwischen Training, in beispielsweise einem Verein, oder Wettkämpfen und Turnieren unterschieden. Zudem werden keine Filterungen bezüglich Alter und Geschlecht vorgenommen.

Die zu entwickelnde Anwendung soll den Anwender über bevorstehende Sportveranstaltungen, sowie allgemeine Sportarten informieren und benachrichtigen. Dazu werden, wie vorgegeben, sowohl synchrone, als auch asynchrone Datenübertragungsarten einbezogen. Der allgemeine Informationsabruft für Informationen bezüglich Sportveranstaltungen wird synchron realisiert. Diese Auskunft werden beispielsweise Sportarten, Sportgruppen (Rückschlagsport, Kampfsport, Schwimmen,...), Termine, Orte, Voraussetzungen, Preise etc. beinhalten. Veranstalter haben die Möglichkeit neue Veranstaltungen zu erstellen und benötigte Räume und Materialien zu reservieren, wobei diese ebenfalls Ressourcen darstellen.

Im Gegenzug dazu werden Abonnements sowie Benachrichtigungen zu einer abonnierten Sportgruppe, Sportart, Räumlichkeit oder Materialien asynchron verwirklicht. Wobei die an Sportveranstaltungen-Interessierten lediglich Ressourcen wie Sportart und Veranstaltung abonnieren können. Veranstalter hingegen können auch Räumlichkeiten und Materialien abonnieren. Dies erleichtert die Verfügbarkeitsprüfung und verhindert, dass Materialien oder Räumlichkeiten doppelt verplant werden. Ebenfalls werden für bereits bestehende Veranstaltungen Terminänderungen wie Ort- oder Zeitänderungen asynchron mitgeteilt indem der Veranstalter/Leiter/Trainer diese publiziert.

## 1.2 Visualisierung des Szenarios

Im Folgenden wird das gewählte Szenario näher beleuchtet und visualisiert.

Beim erstmaligen Aufruf der Applikation (Abb. 1) schickt das System eine Anfrage an den Server um die Basisinformationen über verfügbare Sportgruppen, Veranstalter oder Gebäude zu bekommen. Der Server liefert die angefragten Daten, welche vom System für den Nutzer lesbar dargestellt werden. Nun erfolgt die erste Interaktion des Benutzers mit dem System. Dieser wählt beispielsweise eine Sportgruppe aus. Alternativ bestünde auch die Möglichkeit speziell nach einem Veranstalter bzw. einem Gebäude zu suchen.

Erster Aufruf der Applikation.

Aktualisieren der Oberkategorien:  
Sportgruppen, Gebäude, Trainer

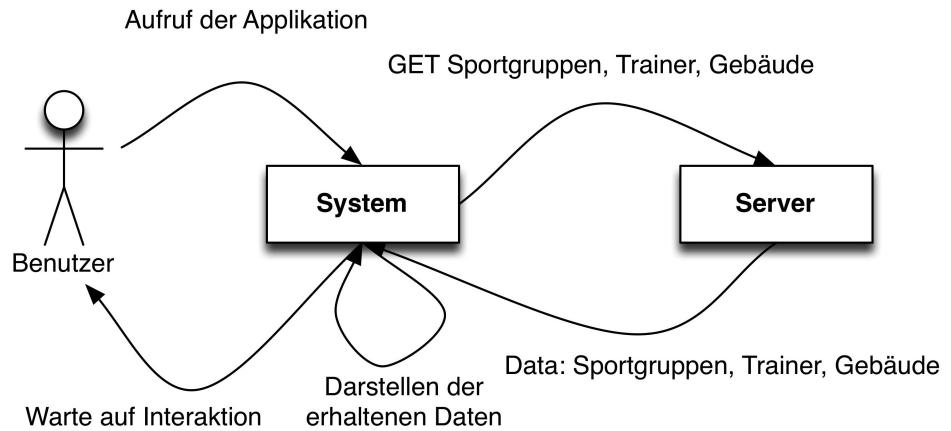


Abbildung 1: Erster Aufruf

Im Folgenden wird ein konkretes Szenario durchgespielt. Die Anwendung ist bereits gestartet und hat, wie oben beschrieben, die Basisinformationen bereits aktualisiert. In diesem Szenario möchte der Benutzer sich über eine Veranstaltung einer bestimmten Sportart als Interessent eintragen, um über Updates dieser informiert zu werden. Erneut wird dem Server eine Anfrage gesendet um zunächst alle Informationen bezüglich einer konkreten Sportgruppe (z.B. Kampfsportart) zu erhalten. Der Server übermittelt somit, sowohl die vorliegenden Informationen bezüglich jener Sportgruppe, als auch mögliche Sportarten dieser Gruppe. Daraufhin stellt die Anwendung diese Informationen sichtbar für den Benutzer dar, sodass alle Sportarten der gewählten Sportgruppenkategorie sichtbar sind. (Abb. 2)

Erste Interaktion mit dem System.  
Benutzer wählt (z.B.) Sportgruppe.  
Informationen zu dieser werden vom Server geholt.  
Alternativ könnte auch nach einem bestimmten Gebäude/Veranstalter gefragt werden.

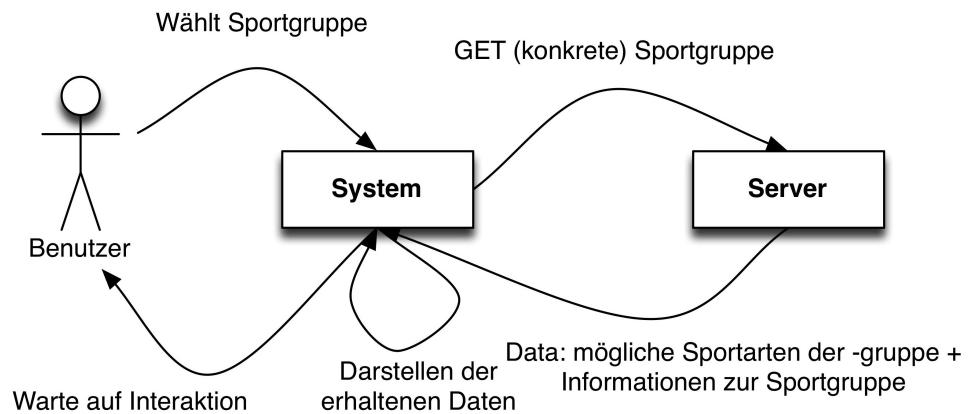
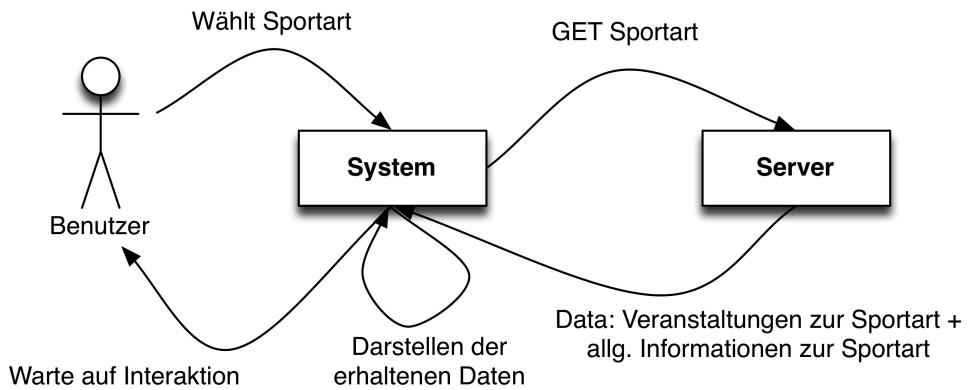


Abbildung 2: Szenario Teil 1

Nun kann eine konkrete Sportart gewählt werden (z.B. Judo, Teakwondo,...).  
Bei einer getätigten Auswahl sendet auch hier das System eine Anfrage an den  
Server und erlangt so die gewünschten Daten. (Abb. 3)

Der Benutzer wählt eine Sportart.

Das System holt die entsprechenden Veranstaltungen und Informationen zu dieser.



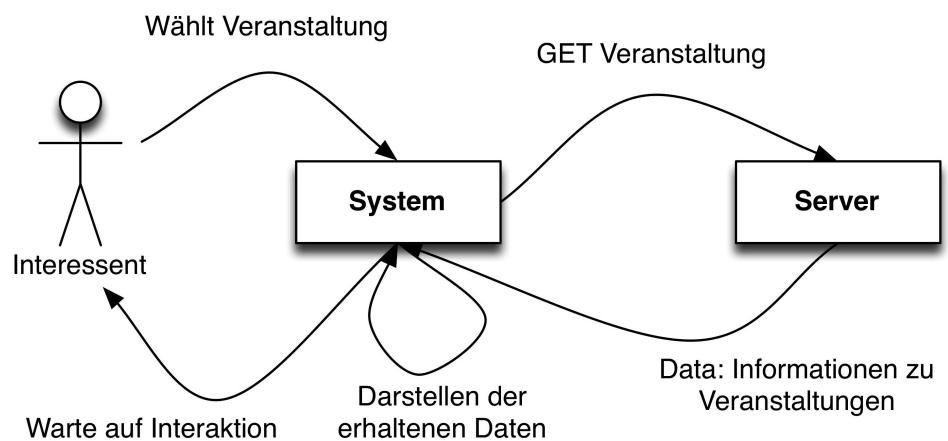
Informationen zur Sportart sind allgemein.

(Herkunft, Geschichte, allg. Voraussetzungen, Regeln,  
übliche Veranstaltungsorte,...)

Abbildung 3: Wahl einer konkreten Sportart

Der Benutzer hat nun die Möglichkeit eine bestimmte Veranstaltung zu seiner gewählten Sportart auszuwählen, um mehr Informationen zu dieser zu erhalten. (Abb. 4)

Der Benutzer wählt eine Veranstaltung.  
Das System holt die entsprechenden Informationen zu der Veranstaltung

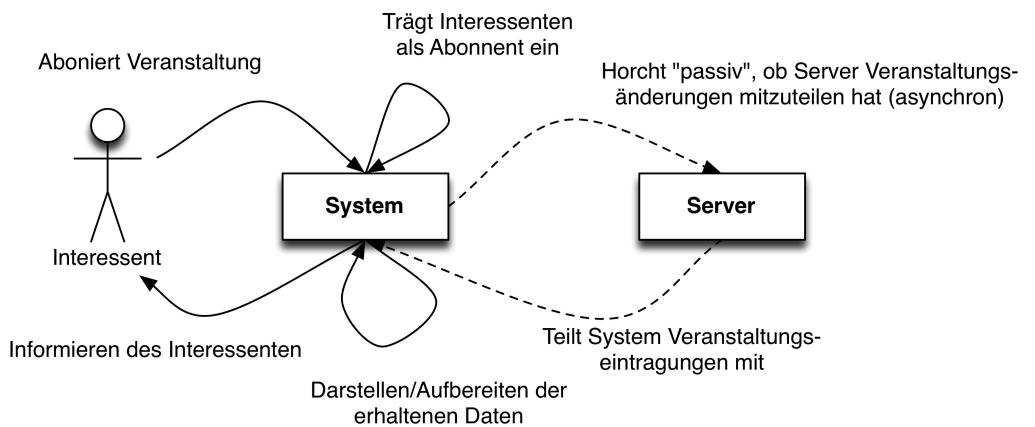


Veranstaltungen beinhalten:  
Art der Veranstaltung: Training/Übung, Turnier/Wettkampf  
Ort, Datum, Uhrzeit  
Voraussetzungen (Mitgliedschaft, Kosten,...)  
Niveau  
benötigtes Equipment  
Veranstalter (Name, Geschlecht, Kontaktdaten)  
weitere Informationen

Abbildung 4: Wahl einer Veranstaltung

Daraufhin kann sich der Benutzer als Interessent für diese Veranstaltung erklären. Das System trägt den Benutzer als Interessenten ein und benachrichtigt diesen asynchron, sobald vom Server Änderungen zu dieser Veranstaltung entdeckt wurden. (Abb. 5)

Der Benutzer tragt sich als Interessent für eine Veranstaltung ein.  
 Die Applikation trägt den Interessenten als Abonnenten ein.  
 Sobald der Server Änderungen der eingetragenen Veranstaltungen übermittelt, wird der Interessent benachrichtigt.



Alternativ möglich:  
 Sportart (o. Veranstalter,...) anstelle der Veranstaltung abonnieren.  
 In diesem Falle wird der Interessent bei Veranstaltungsänderungen  
 (oder neue Veranstaltungen bestimmter Trainer) informiert.

Abbildung 5: Interessenten-Erklärung

Abbildung 6 stellt dar, unabhängig vom obigen Szenario, wie ein Benutzer als Veranstalter agieren könnte, indem er eine Veranstaltung hinzufügt. Der Benutzer wird automatisch als Veranstalter eingetragen, sodass nach diesem gesucht werden kann. Der Veranstalter hat nun die Möglichkeit Abonnements zu bestimmtem Equipment oder Gebäuden abzuschließen. Sollte der Veranstalter bereits von einem oder mehreren Benutzern abonniert worden sein, so werden diese über die eine von ihm hinzugefügte Veranstaltung benachrichtigt. Ebenfalls werden die Veranstaltungslisten aktualisiert (auch asynchron möglich).

Benutzer als Veranstalter.

Der Veranstalter fügt eine Veranstaltung hinzu.

Abonnenten des Veranstalters werden benachrichtigt, die Veranstaltungsliste aktualisiert.

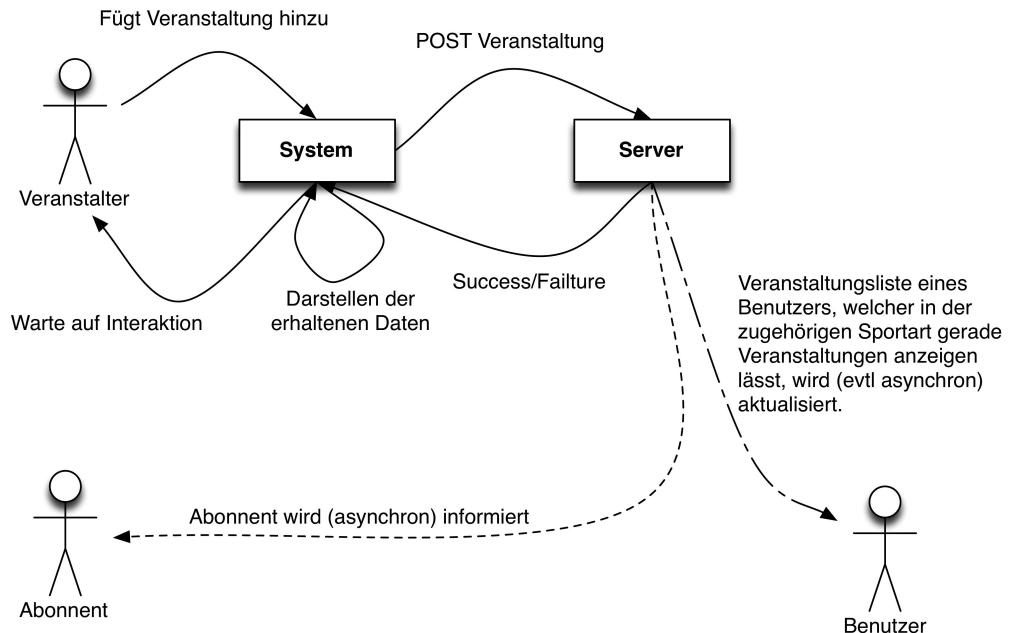


Abbildung 6: Agieren als Veranstalter

### 1.3 Kommunikationsabläufe

Als Vorüberlegung galt es sich gesondert auf das im Szenario dargestellte Problem, dem Informationserhalt von Sportarten und dazugehörigen Veranstaltungen, zu fokussieren und dieses zu erarbeiten. Um mögliche oder gewünschte Kommunikation zu erfassen, wurde jene zunächst grob zusammengefasst und, wie oben dargestellt, visualisiert.

Im Folgenden werden Kommunikationsabläufe beschrieben und erläutert.

- Sportgruppe, welche grob die Prinzipien des Sportes darstellt.
  - Allgemeine Informationen
  - Dazugehörige Sportarten

Zunächst zur Begrifflichkeit der Sportgruppen: Jene stellen eine Art Oberkategorie dar, um einzelne Sportarten besser und freundlicher zu gliedern und unterteilen. Beispiele hierfür wären Kampfsportarten welche beispielsweise die Sportarten Judo, Kung-Fu und ähnliche beinhalten.

Zu diesen Sportgruppen werden an dieser Stelle Beschreibungen der Sportgruppe geliefert. Prinzipien, Herkunft und Ideale könnten an dieser Stelle näher erläutert und betrachtet werden. Beispielhaft: „Unter Kampfsport definiert man einen Sammelbegriff für diverse Sportarten in welchen Kampfstile integriert werden.“

- Sportart
  - Allgemeine Informationen zu der Sportart (Herkunft, allg. Voraussetzungen, Geschichte, Regeln, übliche Veranstaltungsorte,...)
  - Laufende Veranstaltungen
- Veranstaltung
  - Ort der Veranstaltung
  - Datum, Uhrzeit
  - Voraussetzungen (Anmeldung erforderlich, Mitgliedschaft, Begrenzte Teilnehmerzahl (Minimale Teilnehmerzahl), Kosten, Niveau (Anfänger, Fortgeschritten, ...))
  - benötigtes Equipment
  - Veranstalter (Name, Geschlecht, Kontaktdaten)
  - Weitere Informationen

Unter jeder dieser Sportgruppen-Kategorie befindet sich eine Vielzahl von Sportarten, welche der jeweiligen Gruppe angehören. Diese werden aufgelistet und auch mit einer groben Beschreibung vervollständigt. Des Weiteren erhält man zu einer konkreten Sportgruppe Angaben zur Herkunft, Geschichte, Regeln und

andere nützliche Informationen. Das Wesentliche, welches zugleich die Problemstellung des Szenarios darstellt, ist allerdings an dieser Stelle, der Erhalt von Informationen über Veranstaltungen bezüglich einer konkreten Sportart. An dieser Stelle entschied man sich dazu nicht zwischen Training/Übung und Turnier/Wettkampf zu unterscheiden sondern alle möglichen Veranstaltungen zu listen. Zu einer konkreten Veranstaltung werden dann Informationen wie der Ort, die Uhrzeit, das Datum und eventuelle Voraussetzungen publiziert. Diese könnten beispielsweise eine notwendige Anmeldung oder begrenzte Teilnehmerzahl beinhalten.

#### 1.4 Funktionen des Systems:

Im Folgenden werden die Funktionen des Systems knapp beschrieben und erläutert. Wichtig ist an dieser Stelle, dass jene Funktionen derzeit nur auf das Minimum beschränkt wurden und eventuell bei Bedarf angepasst werden können. Auch handelt es sich hier nur um die Planung, sodass die eigentliche Funktion, welche erzielt wurde, erst am Ende beschrieben werden kann.

Zunächst wurde allerdings der Fokus auf den Erhalt der Informationen über Sportveranstaltungen gelegt. Hierbei gilt es besonders darauf zu achten welche Funktionen synchron und welche asynchron realisiert werden. Um die Unterscheidung an dieser Stelle nochmals genau deutlich zu machen, wurde sich mit beiden Interaktionsarten auseinandergesetzt, sodass eine sinnvolle Wahl getroffen werden konnte. Anzumerken ist noch, dass während des gesamten Verlaufs des Projektes einige Optimierungen, Anpassungen und Veränderung durchgeführt worden, da diese sinnig schienen.

- Synchron
  - Interessent:
    - \* alle Veranstaltungen zu einer Sportart(Trainer, Gebäude ...) liefern (GET)
    - \* Über Sportart/-gruppe informieren (GET)
  - Trainer / Veranstalter
    - \* hinzufügen einer neuen Veranstaltung (POST)
    - \* Änderungen an seiner Veranstaltung (PUT)
    - \* Löschen seiner Veranstaltung (DELETE)
    - \* Zugriff auf Ressourcen (Equipment/Gebäude)<sup>1</sup>
- Asynchron
  - Abonnements der Interessenten (Neue Events, Änderungen)
    - \* Sportarten

---

<sup>1</sup>Wie bereits erwähnt, wurde sich an dieser Stelle auf die Veranstaltung fokussiert. Im weiteren Verlauf könnte man zwecks Reservierung den Erhalt von Informationen zu Gebäuden und Equipment näher ausarbeiten.

- \* Veranstalter
- \* Veranstaltungen
- Abonnements der Veranstalter
  - \* Equipment
  - \* Gebäude/Räumlichkeit

Da diverse Interaktionen mit dem System möglich sein sollen, scheint es unumgänglich zwischen den Gruppen der Interessenten und der Trainer beziehungsweise Veranstalter zu unterscheiden. So ergibt es sich, dass ein Interessent lediglich Informationen bei Ausführung der Anwendung erhält. Dem Trainer soll es zusätzlich ermöglicht werden, neue Veranstaltungen zu veröffentlichen, beziehungsweise zu erstellen, sowie bereits bestehende Veranstaltungen zu löschen. Hierbei ist jedoch wichtig, dass es einem Trainer nur möglich ist seine eigenen Veranstaltungen zu ändern oder zu löschen.

Asynchron soll allgemein lediglich das System der Abonnements verwirklicht werden. Auch an dieser Stelle wurde, aufgrund der verschiedenen Funktionalitäten, zwischen Interessent und Veranstalter unterschieden. So ist es dem Interessent lediglich möglich Sportarten, Veranstaltungen und Veranstalter zu abonnieren. Abonnieren bedeutet in diesem Kontext den Erhalt einer Benachrichtigung bei Veränderung einer abonnierten Ressource. Konkret wird dem Benutzer eine Benachrichtigung zugestellt sobald eine von ihm abonnierte Veranstaltung (oder Veranstalter bzw. Sportart) Änderungen darstellt. So wird man über mögliche Terminänderungen frühzeitig informiert.

Ein Veranstalter erhält die Möglichkeit Equipment und Gebäude/Räumlichkeiten zu abonnieren. Dies kann, wie auch bei der synchronen Interaktion, der Vereinfachung von „Reservierungen“ dienen. So erhält ein Trainer beispielsweise eine Benachrichtigung wenn ein von ihm abonniertes Equipment neuerdings (oder einmalig) doch verfügbar ist, sich also etwas geändert hat.

## 2 XML Schema

Entsprechend des ersten Meilensteins galt es ein projektbezogenes XML-Schema / -Schemata zu entwickeln. Dieses dient der Planung und Konzeption und soll besonders in Anbetracht der späteren Weiterarbeit mit JAXB valide sein.

Bevor ein XML-Schema für das bestehende XML-Dokument ausgearbeitet wird, soll zunächst einmal verdeutlicht werden, worum es sich eigentlich bei einem XML-Schema handelt. Ein XML-Schema hat die selbe Aufgabe, wie eine Document Type Definition, nämlich die Struktur eines XML-Dokumentes zu spezifizieren. Jedoch mit einigen Unterschieden. Beispielsweise ist das XML-Schema selber auch in XML geschrieben.

In XML-Schemata wird grundlegend zwischen zwei Arten von Typen unterschieden: Simple Types und Complex Types. Simple-Types sind Elemente (und Attribute), welche nur Text, also entsprechend keine weiteren Kindelemente oder Attribute enthalten. Complex-Types sind somit alle Elemente, welche

Kindelemente und/oder Attribute besitzen. Das Attribut selbst ist immer ein simple-type.

Im Folgenden werden die Elemente des XML-Schemas benannt, erklärt und erläutert. Des Weiteren werden die Restriktionen belichtet und die Verbindung beschrieben. Jedoch wird die genaue Beschreibung von Simple Types und Complex Types an dieser Stelle missachtet, da bereits eine Erklärung in Phase 1 erfolgt ist.

Zunächst aber noch zu einem Attribut, welches sich in beinahe allen Listenressourcen befindet: 'id'.

Bei der Vergabe für die Identifikation des Attributs 'id' der Elemente, um die Referenzierung und damit die Zugehörigkeit sicherzustellen, bietet sich die Möglichkeit an, innerhalb des Schemas nicht mit dem Attributwert (`type="id"`) zu arbeiten, sondern mit zum Beispiel einem Integer. Der Grund darin besteht, dass eine ID, in dem zu entwickelndem System, nicht eindeutig vergeben werden muss, es sei denn sie befindet sich im selben Elterelement. Dazu ein Beispiel:

Wenn es zwei unterschiedliche Veranstaltungen gibt, welche einer unterschiedlichen Sportart angehören, so können beide Veranstaltungen die ID 1 bekommen, da sie über die zugehörige Sportart und Sportgruppe angesprochen werden müssen. Somit ergibt sich die eindeutige Identifizierung bereits (siehe 3.1.2 auf Seite 19). Sollten jedoch in einer selben Sportart mehrere Veranstaltungen vorkommen, so dürfen dort nur eindeutige ID's vergeben werden.

Allerdings setzt dies natürlich voraus, dass eine Veranstaltung nur ausgewählt werden kann, wenn zunächst vorher die Sportart gewählt wurde. Dies entspricht zwar einer guten Struktur und ist auf den ersten Blick klar und verständlich, jedoch kann dies auch eine nachteilige Wirkung erzielen. Beispiele:

- Sollte das Gebäude bekannt sein, so muss der Umweg immer über den Ort geschehen.
- Sollte ein Veranstalter seine auszutragenden Veranstaltungen anzeigen lassen wollen, so muss dieser in jede Sportgruppenliste, danach jeweils in jede Sportartenliste nachschauen, um zu ermitteln, zu welcher Sportart eine Veranstaltung mit seiner eingetragenen Veranstaltungs-ID passt.

Auch wenn diese Arbeit vom System abgenommen wird, so ist dies nicht unbedingt effizient, da in jedem Fall der Index bzw. die ID des Elterelements bekannt sein und angegeben werden muss. So wird im späteren Verlauf das System mit Veranstaltungen arbeiten, sie muss jedoch bei jedem Arbeitsschritt die ID der beiden Elterelemente kennen, also wenigstens zwischenspeichern. Bei einer deutlich tieferen Schachtelung - wie sie zum Glück aktuell nicht vorliegt - sollte die Umsetzung der Hierarchie einem anderen Prinzip folgen.

Zunächst ist auch die im Folgenden vorgestellte Hierarchie zu beschreiben; Bei dem Element 'Sportverzeichnis' handelt es sich um die Wurzel des XML-Dokuments. Bereits bei Betrachtung des Schemas kann man die spätere Struktur der Anwendung erkennen. So werden die Sportgruppen, die Orte sowie die Veranstalter als eine Art Oberkategorie festgelegt zwischen welchen man später

wählen kann.<sup>2</sup> Betrachtet man die nicht vorhandenen Restriktionen, so wird ersichtlich, dass es jeweils genau ein Sportgruppen-Listen-, Orte-Listen- und Veranstalter-Listen-Element geben muss. Dies ist der Fall, da, auch eine leere Liste vorhanden sein darf. Allerdings kann lediglich die Veranstalter-Liste leer sein; die Sportgruppen (auch Sportarten), sowie Orte (auch Gebäude) sind vom System vordefiniert und weitestgehend nicht dynamisch.<sup>3</sup>

Die Sportgruppen-Liste stellt eine Liste aller Sportgruppen dar. Zunächst zur Begrifflichkeit: Unter Sportgruppen werden Oberkategorien von Sportarten verstanden, unter welche diese zusammengefasst werden können. Beispiele sind Kampfsport, Rückschlagsportarten, usw.. Sinnvollerweise enthält die Sportgruppen-Liste mindestens eine Sportgruppe, da das Vorhandensein der Sportgruppen essentiell für das System ist; Veranstaltungen gehören zu einer Sportart, ohne Sportarten gäbe es keine Veranstaltungen, welche nunmal das Hauptmerkmal des Systems darstellen. Eine maximale Anzahl von Sportgruppen ist an dieser Stelle und auch im weiteren Verlauf nicht gegeben (maxOccurs = "unbounded").

An dieser Stelle dienen Simple Types lediglich der genaueren Beschreibung der Ressource 'Sportgruppen'. Lediglich Informationen wie Namen und Beschreibungen werden geliefert. Durch Complex-Types wird erneut die Referenzierung zu einer Liste, in diesem Fall der Liste aller Sportarten (dieser konkreten Sportgruppe) realisiert. Des Weiteren werden im Complex-Type alle enthaltenen Simple-Type Elemente referenziert und Attribute definiert.

Auch bei dem Element 'SportartenM' handelt es sich um eine Listenressource. Um erneut die Restriktionen zu erläutern kann man eine Mindestanzahl von einer Sportart pro Sportarten-Liste erkennen. Dies begründet sich, wie bereits erwähnt, dadurch, dass eine leere Sportartenliste für die Anwendung unsinnig wäre und nicht zum gesetzten Ziel beziehungsweise zur geplanten Funktionalität führen würde. Logischerweise gibt es für eine Liste von Sportarten keine maximale Anzahl an konkreten Sportarten.

Eine konkrete Sportart hat sowohl Simple-Types als auch Complex-Types. Zunächst zu den Simple-Types. Durch Sie werden allgemeine Informationen, also simple Strings, über eine konkrete Sportart dargestellt. Diese sind zum Beispiel der Name der Sportart, eine kleine Beschreibung, eventuell notwendige Voraussetzungen, die Herkunft sowie Regeln. Das eigentliche Element 'Sportart' stellt einen Complex-Type dar. Dieser enthält die zuvor genannten Simple Types, sowie einige Referenzierungen. So wird von einer konkreten Sportart auf eine Liste von Veranstaltungen (zu dieser Sportart) referenziert.

Die Liste aller Veranstaltungen (einer Sportart) sind, wie wahrscheinlich vermutet, auch was ihre Anzahl betrifft, nicht nach oben begrenzt. Lediglich eine Liste muss vorhanden sein, denn ansonsten, würde wie bereits oftmals erläutert der eigentliche Sinn der Funktionalität verloren gehen.

Eine konkrete Veranstaltung wird durch eine Beschreibung<sup>4</sup>, einen kurzen

---

<sup>2</sup>Zu erkennen ist dies, da das Wurzelement 'Sportverzeichnis' lediglich diese drei Kindelemente anspricht.

<sup>3</sup>Ausnahmen können durch den Administrator am Server vorgenommen werden.

<sup>4</sup>Die Beschreibung kann auch als Name genutzt werden.

Informationstext, das Datum, an welchem diese stattfindet und die Uhrzeit beschrieben. Besonders an diesem Punkt wird im späteren Verlauf der Fokus der asynchronen Kommunikation liegen. Somit wird ein Benutzer augenscheinlich über eine Veränderung, beispielsweise der Uhrzeit, informiert. Zudem erschienen Angaben wie das Niveau sinnvoll zu implementieren. Somit können Angaben wie „Anfänger“ oder „Fortgeschrittene“ getätigten werden. Zudem ist es möglich nötige Voraussetzungen festzulegen. Diese könnten zum Beispiel sein, dass ein Tennisschläger mitgebracht werden muss. Die letzten beiden genannten Angaben zu einer Veranstaltung, sowie der Informationstext zu dieser, sind optional. Der Veranstalter muss diese nicht angeben; oftmals gibt es keinerlei Voraussetzungen, ein Niveau muss nicht vorgeschrieben sein bzw. ist in einigen Fällen auch nicht sinnvoll. Der Informationstext kann dazu dienen, detailliertere Informationen zu der Veranstaltung hinzuzufügen - sollte jedoch die Beschreibung ausreichen kann das Element ‘Info’ vernachlässigt werden. Diese Optionalitäten sollen dem Veranstalter mehr Flexibilität bieten, ihn jedoch nicht durch übermäßigen Informationsgehalt überfordern. Daher haben diese Elemente ein optionales Vorkommen, im Schema angegeben durch ‘minOccurs = “0”’.

## 2.1 Datentypen

Für alle Elemente des XML-Schemas sind weitestgehend vorgefertigte Datentypen verwendet worden. Dementsprechend wurden bei allen Elementen Standarddatentypen wie ‘String’ oder ‘Integer’ verwendet, außer bei dem Element ‘VTGeschlecht’ (Geschlecht des Veranstalters). Dieses Element hat einen eigenen Datentyp erhalten; die Wahl darf lediglich auf weiblich (w) oder männlich (m) fallen. Eine abweichende Angabe des Geschlechtes müsste dann mühselig von der späteren Programmlogik abgefangen werden; für diese Fälle sind selbstdefinierte Datentypen entworfen worden. Eine weitere Ausnahme stellen die Daten und Uhrzeiten dar. Der Einfachheit halber wurden aber auch hier vorgefertigte Datentypen (date, time) verwendet. Die Möglichkeit bestünde, einen eigenen Datentyp für jeweils Datum und Uhrzeit zu erstellen, jedoch wäre dies durchaus überflüssig, da die spätere Verwendung mit Java (JAXB) ein Objekt für XML-Daten und -Uhrzeiten unterstützt und die Handhabung der Daten einfach hält: ‘XMLGregorianCalendar’.

Die eingangs angesprochenen ID's der Elemente weisen eine Unstimmigkeit gegen die übliche Handhabung von ID's in XML-Schemata auf; sie werden mit einem ‘String’-Datentyp versehen. Dies resultiert aus folgendem Grunde: Attribute mit dem Datentyp ‘ID’ müssen im gesamten XML-Dokument eindeutig sein. Wie des Öfteren erwähnt ist dies für das zu entwerfende System, aufgrund der Verschachtelungsstruktur, nicht umgesetzt. Zudem war es wünschenswert immerhin in der ID einen Präfix zu platzieren, welcher Auskunft darüber gäbe, um welch ein Element es sich handelt. Sollte entsprechend das erste Sportgruppen-Element angesprochen werden, eignete sich eine ID wie ‘SG0’. Diese unterscheidet sich von der Veranstaltungs-Element ID (‘V0’) insofern, dass allein durch die ID bereits erkennbar ist um welches Element es sich handelt (Sportgruppe/Sportart/Veranstaltung/Ort...). Im späteren Verlauf des Projek-

tes wurde dieser Präfix jedoch wieder entfernt; er würde die Handhabung mit den eigentlich interessanten Werten<sup>5</sup> erschweren, da die Zeichenkette auseinandergeschnitten werden müsste. Aus diesem Grunde ist der Datentyp der ID ein String geblieben (da das System schon relativ weit entwickelt worden war), obwohl lediglich ganzzahlige Werte als ID vergeben werden. Entsprechend musste, insbesondere bei der Entwicklung der grafischen Benutzeroberfläche, sehr oft auf Funktionen wie ‘String.toString()’ zugegriffen werden, was die Entwicklung des Systems dennoch glücklicherweise keineswegs einschränkte.

## 2.2 Attribute

Im gesamten Schema finden sich nur drei Attribute wieder:

1. Das Attribut ‘id’ zum identifizieren der Complex-Type Elemente,
2. das Attribut ‘entliehen’ um bei Equipment die Verfügbarkeit dieser angeben zu können,
3. das Attribut ‘deleted’ bei einer Veranstaltung.

Das erste Attribut, nämlich die ID, sollte soweit klar sein. Das Attribut ‘entliehen’ wird bei jedem Equipment-Teil benötigt, um feststellen zu können, ob es im Moment zur Ausleihe zur Verfügung steht, oder ob es bereits von einem anderen Veranstalter entliehen wurde. Dieses Attribut hat den Datentyp boolean; ist ‘entliehen = true’, so hat bereits ein Veranstalter dieses Equipment reserviert. ‘entliehen = false’ beschreibt, dass das betrachtende Equipment von Veranstaltern zur Ausleihe zur Verfügung steht.

Das Attribut ‘deleted’, welches, wie ‘entliehen’ bei dem Element Equipment, lediglich für Veranstaltungen verwendet werden kann, beschreibt, ob eine Veranstaltung gelöscht ist oder nicht<sup>6</sup>. Dies mag nun gegen gewöhnliche Datenhaltung widersprechen, doch ist dieses Attribut aber auch die Speicherung von gelöschten Veranstaltungen durchaus beabsichtigt. Die Löschung einer Veranstaltung (bzw. das Setzen des Attributes ‘deleted = true’), aber das Beibehalten der Veranstaltungsressource wird ‘logisches Löschen’[1, Seite 55] genannt. Es hat den Vorteil, dass der Client nach außen hin nur Veranstaltungen angezeigt bekommt, welche das ‘deleted’-Attribut mit dem Attributwert ‘false’ aufweisen, aber dennoch “gelöschte Veranstaltungen” nicht komplett vom Server gelöscht werden, sodass, auch nach Entfernen der Veranstaltung aus der angezeigten Liste, im Not- oder Fehlerfall immer noch noch auf die Daten zugegriffen werden können.<sup>7</sup>

---

<sup>5</sup>Nämlich der ID ohne Präfix, z.B: ‘54’.

<sup>6</sup>Auch hier wird der Datentyp ‘boolean’ verwendet.

<sup>7</sup>Natürlich erfolgt dieser Zugriff nicht vom Standardbenutzer/Client, sondern eher von einer administrativen Einheit.

## 2.3 Struktur des Schemas

Für das System wurde lediglich eine Schemadatei angefertigt. Sicherlich wären mehrere Schemata denkbar. So könnten Sportgruppen, Sportarten und Veranstaltungen ein Schema bekommen, Orte und Gebäude eins, und Veranstalter ebenfalls eins. Im Grunde bieten jedoch mehrzählige Schemata keinen wirklichen Vorteil gegenüber ein einzelnes Schema. Sie können durchaus für eine übersichtlichere Struktur verhelfen, jedoch im selben Zuge eventuell für Inkonsistenzen sorgen, nämlich, dann, wenn die Schemadatei zwar valide ist, jedoch mit der anderen Schemadatei nicht harmonisiert. Um jegliche Informationen zu bündeln und nicht unabhängig voneinander verwalten zu müssen, wurde lediglich eine Schema-Datei sowie eine XML-Datei verwendet. Für die bessere Struktur wurde auf die ‘Russian-Doll’-Darstellung (Für nähere Informationen Verweis auf Phase 1) verzichtet, sodass die eben erwähnten, so genannten ‘Oberkategorien’ (genauer genommen alle ComplexTypes) im gesamten XML-Dokument global ansprechbar sind. Dies hat nicht nur zu Folge, dass die Darstellung innerhalb des XML-Schemas deutlich übersichtlicher wird, da keine großen Verschaltelungstiefen entstehen, sondern auch, dass bei späterer Verwendung von JAXB, jedes Complex-Type-Element eine eigene Klasse zugewiesen bekommt. So kann von jedem Vaterelement ein JABX-Objekt gebildet werden, welches gleichermaßen individuell ansprechbar ist.

## 3 RESTful Webservice

Des Weiteren sollte ein RESTful Webservice in Java unter folgenden Bedingungen erstellt werden: Mindestens zwei Ressourcen müssen implementiert sein JAXB soll für das marshalling / unmarshalling verwendet werden. Die Operationen GET, DELETE und POST müssen implementiert sein und es sollen mindestens einmal PathParams und einmal QueryParams verwendet werden.

Der Kern eines jeden RESTful WebServices liegt in den fünf Prinzipien[3, Folie 132-136], welche im Folgenden genannt und kurz erläutert werden:

1. Eindeutige Ressourcen: Im Wesentlichen, geht es an dieser Stelle darum, sich mit allen möglichen Ressourcen auseinanderzusetzen um somit alle zu finden. Hierauf wird näher im Abschnitt 3.1 eingegangen.
2. Verknüpfung von Ressourcen: Es ist von Vorteil die genannten Ressourcen mittels Hypermedia miteinander zu verknüpfen. Dies wurde in der zu entwickelnden Anwendung beachtet, sodass alle Ressourcen auf sinnvolle Weise mit anderen verbunden sind. Dazu kommt, dass es sich erst um einen RESTful Webservice handelt, sobald Hypermedia eine enorme Rolle spielt[1]. Auch diese Thematik wird im Abschnitt 3.1.2 näher erläutert.
3. Standardverben: Es ist sinnig, besonders in Anbetracht der korrekten Kommunikation, die Standardverben für die Ressourcen zu verwenden. Auch an dieser Stelle wird auf Abschnitt 3.2 verwiesen.

4. Diverse Repräsentationen: Um eine möglichst weite Verbreitung der Anwendung garantieren zu können, ist es wichtig verschiedene Repräsentationen wie XML, HTML oder JSON vorzusehen. Da es im gegebenen Kontext weniger um die Verbreitung der Anwendung geht, wurde sich aufgrund der Einarbeitung in Phase eins für XML entschieden. Dennoch ist es sinnvoll, wenigstens eine weitere Repräsentation (wie z.B. HTML) zu unterstützen, da somit gewährleistet ist, dass auch Clients, die den vorgeschriebenen MIME-Type nicht unterstützen, immerhin anteilig Informationen, wenn auch nur einen Hinweis, bekommen. Da dies wie beschrieben für das aktuelle System zwar durchaus nützlich und realistisch wäre, jedoch nicht den Anforderungen und der Zielsetzung entspricht, wurde zunächst auf eine weitere Repräsentation verzichtet.
5. Statuslose/zustandslose Kommunikation: Statuslose Kommunikation beschreibt, dass jede Anfrage vom Server unabhängig von einer vorherigen Anfrage behandelt wird. Das heißt, der Server ‘vergisst’ den Client sofort wieder, nachdem die Anfrage gesendet wurde. Es wird demnach Serverseitig keinerlei Sitzungsstatus gehalten; dieser wird entweder Clientseitig realisiert, oder vom Client übertragen indem die Sitzung beispielsweise in eine Ressource überführt wird. Des Weiteren wird die Verbindung zwischen Client und Server nur so lange gehalten, bis der Client seine Antwort erhält, bzw. bis der Server mit der Abarbeitung der Anfrage fertig ist (zustandslose Kommunikation). Der Vorteil dieses Architekturmerkmals ist offensichtlich; der Client ist vom Server weitestgehend entkoppelt; es können mehrere Serverinstanzen den selben Dienst realisieren<sup>8</sup>, allerdings nicht, wenn einer der Server einen benötigten Sitzungsstatus halten sollte, welcher für die anderen Server unerreichbar ist.

### 3.1 Ressourcen

An dieser Stelle sollten, besonders um die Weiterarbeit zu vereinfachen, mögliche und vor allem notwendige Ressourcen beschrieben und erläutert werden. Hierzu bedarf es einer intensiven Auseinandersetzung mit diesen und deren Operationen, welche im späteren Verlauf noch erläutert werden.

Eingangs soll eine theoretische Auseinandersetzung mit Ressourcen im Kontext RESTful Webservices erfolgen. Das erworbene Wissen ist Grundlage um die nächsten Schritte erfolgreich absolvieren zu können. Es sollen projektbezogene Beschreibungen der für das Projekt benötigten Ressourcen und Operationen inklusive Begründung der Entscheidungen erstellt werden.

#### 3.1.1 Findung von Ressourcen

Da Ressourcen bei RESTful Webservices eine sehr wichtige Rolle spielen, wurden diese sehr ausführlich ausgearbeitet. Eine Ressource muss eindeutig identifizierbar sein und hat eine oder mehrere Repräsentationen. Zudem können

---

<sup>8</sup>Und somit Anfragen unabhängig voneinander verabreiten.

standard Verben, aber auch selbst definierte Verben auf diesen Ressourcen Anwendung finden und Ressourcen Verbindungen zu anderen eingehen.

Für das System, welches über Sportveranstaltungen und Informationen bezüglich diverser Sportarten liefert, sollen im Folgenden Ressourcen definiert werden, mithilfe dessen die weiter oben genannten Funktionen des Systems (1.4) ermöglicht werden können.

Zunächst wurde die grobe Gliederung des zuvor erstellten Kommunikationsablaufes (1.2) übernommen, da bereits dort gründlich das Konzept des späteren Systems ausgearbeitet und sich Überlegungen zu möglichen Ressourcen gemacht wurde. Diese Ressourcen werden im Weiteren als Primärressourcen bezeichnet; “frühe Kandidaten für persistente Entitäten”[1, Seite 35].

#### **Primärressourcen:**

- konkrete Sportgruppe
- konkrete Sportart
- konkrete Veranstaltung
- konkreter Ort
- konkretes Gebäude
- konkretes Equipment
- konkreter Veranstalter

Primär stehen, an dieser Stelle, nur die Veranstaltungen im Mittelpunkt. Hierbei jedoch ist es, besonders für den Benutzer, wesentlich komfortabler Veranstaltungen in gewissen Oberkategorien unterteilt zu erhalten. Somit werden jene in ihre jeweilige Sportart und deren dazugehörige Sportgruppe unterteilt. In weiteren Überlegungen, kam man zu dem Entschluss, vorhandenes Equipment und Veranstalter auch als Primärressourcen zu definieren, um diese - eventuell - zum Ende noch zu implementieren. Diese stellen jedoch lediglich eine Art Zusatzfunktionen dar und weichen vom eigentlichen Fokus, der Anzeige von Veranstaltungen, ab. Somit ist eine Art Filterung durch den Benutzer wesentlich einfacher und freundlicher gestaltet. Man erhält die Möglichkeit unmittelbar nach einem genauen Veranstalter zu suchen. Wohingegen ein Veranstalter somit die Möglichkeit erhält nach genauem Equipment zu filtern.

Das Equipment soll an dieser Stelle lediglich dem Veranstalter, welcher in diesem Falle ebenfalls Benutzer des Systems ist, dienen und seine Planung bezüglich seiner von ihm durchgeführten Veranstaltungen erleichtern. Des Weiteren soll es dem Nutzer ermöglicht werden, sich speziell nach Veranstaltern zu informieren und direkte Informationen bezüglich der von ihm angebotenen Veranstaltungen erhalten.

Damit nicht jede Ressource einzeln angesprochen werden muss, werden im Folgenden noch Listenressourcen definiert, welche die gesamte Menge der konkreten Ressourcen anspricht. Im Falle einer Abfrage nach allen Sportgruppen,

muss so nicht jede einzelne Sportgruppe mit ihrer eindeutigen ID bzw. URI angesprochen werden, sondern kann die Sportgruppen-Liste angesprochen werden, welche alle Sportgruppen liefert. Des Weiteren wird, in unserem System, erst durch die Listenressourcen das Hinzufügen eines neuen Elements mittels POST ermöglicht.

#### **Listenressourcen:**

- Liste aller Sportgruppen
- Liste aller Sportarten (einer Sportgruppe)
- Liste aller Veranstaltungen (einer Sportart)
- Liste aller Orte
- Liste aller Gebäude (eines Ortes)
- Liste mit Equipment (eines Gebäudes)
- Liste aller Veranstalter

Ersichtlich bei der Erstellung der Listenressourcen ist, dass im gleichen Zuge Filter angewendet worden sind und zudem eine Art der Verlinkung (Hypermedia) bereits erkennbar ist. Der Grund für diese Filterung ist, dass die gesamten Listen eventuell gar nicht relevant sind. Möchte ein Benutzer beispielsweise wissen, welche Sportarten eine gewisse Sportgruppe enthält, so muss hierfür nicht die gesamte Liste aller Sportarten angefordert werden, sondern lediglich jene, welche zu der entsprechenden Sportgruppe passen. Trotz dieser Filterkriterien, handelt es sich hierbei dennoch um Ressourcen.

#### **3.1.2 Eindeutiger Identifier und die Verbindungen zu Ressourcen**

Wie bereits bei einigen der Listenressourcen erkennbar ist, sind mehrere Ressourcen von der Existenz anderer Ressourcen abhängig. Diese Art der Ressourcenmodellierung ist durchaus beabsichtigt. Im Folgenden sind weitere Abhängigkeiten bzw. Verbindungen zu Ressourcen aufgelistet.

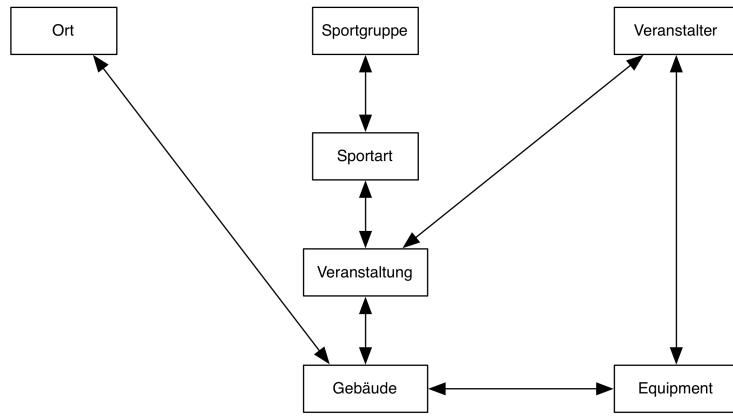


Abbildung 7: Strukturelle Anordnung der Ressourcen

### Liste aller Sportgruppen

- URI: /sportgruppen/
- Verbindung:
  - jede konkrete Sporgruppe

Die Ressource ‘Liste aller Sportgruppen’ enthält alle Sportgruppen. Entsprechend ist die Liste aller Sportgruppen mit jeder einzelnen Sportgruppe verbunden. Diese Ressource wird beim Start der Applikation angefordert.



Abbildung 8: Liste aller Sportgruppen

### Konkrete Sportgruppe

- URI: /sportgruppen/{id} /

- Verbindung:
  - Liste aller Sportarten (dieser konkreten Sportgruppe)

Die Ressource ‘konkrete Sportgruppe’ ist ein Teil von der Ressource ‘Liste aller Sportgruppen’. Sie ist verbunden mit den zugehörigen Sportarten, das heißt mit der Ressource ‘Liste aller Sportarten’, welche dieser konkreten Sportgruppe angehören. Diese Ressource wird angefordert, sobald aus der Listenressource ‘Liste aller Sportgruppen’ eine Sportgruppe ausgewählt wurde, zu jener der Benutzer mehr erfahren möchte.

*Beispiel: Kampfsport (/sportgruppen/1/) aus Sportgruppenliste (/sportgruppen/)*

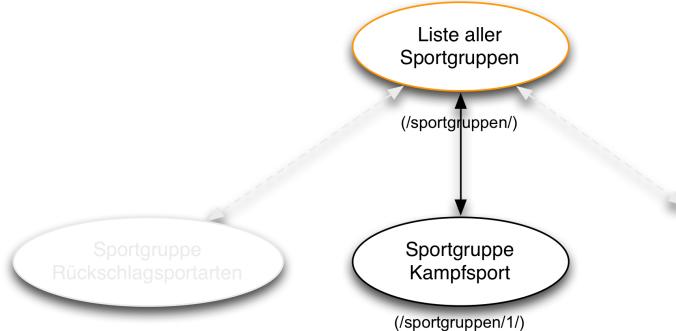


Abbildung 9: Konkrete Sportgruppe

#### Liste aller Sportarten (einer konkreten Sportgruppe)

- URI: /sportgruppen/{id}/sportarten/
- Verbindung:
  - jede konkrete Sportart (dieser Sportgruppe)

Die Ressource ‘Liste aller Sportarten’ ist eine Liste aller Sportarten, einer bestimmten Sportgruppe. Diese Liste enthält Verbindungen zu den einzelnen konkreten Sportarten.

*Beispiel: Judo, Karate, Boxen, Kung FU Kian, KuTaeKa-Do,... (/sportgruppen/1/sportarten/) aus Kampfsport (/sportgruppen/1/)*

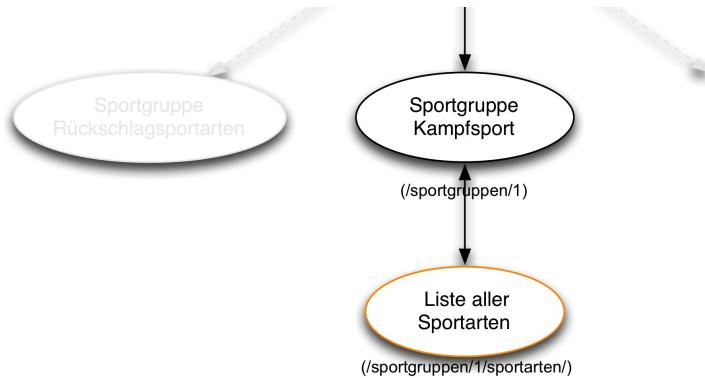


Abbildung 10: Liste aller Sportarten

### Konkrete Sportart

- URI: /sportgruppen/{id}/sportarten{id}/
- Verbindung:
  - Liste aller Veranstaltung (dieser Sportgruppe)

Die Ressource ‘Konkrete Sportart’ ist ein Teil von der Ressource ‘Liste aller Sportarten’, die einer zuvor gewählten Sportgruppe angehören. Sie ist verbunden mit den zugehörigen Veranstaltungen, bzw. mit der Ressource ‘Liste aller Veranstaltungen’, welche zu der gewählten Sportart gehören.

*Beispiel: Judo (/sportgruppen/1/sportarten/12/) aus Sportartenliste (/sportgruppen/1/sportarten/)*

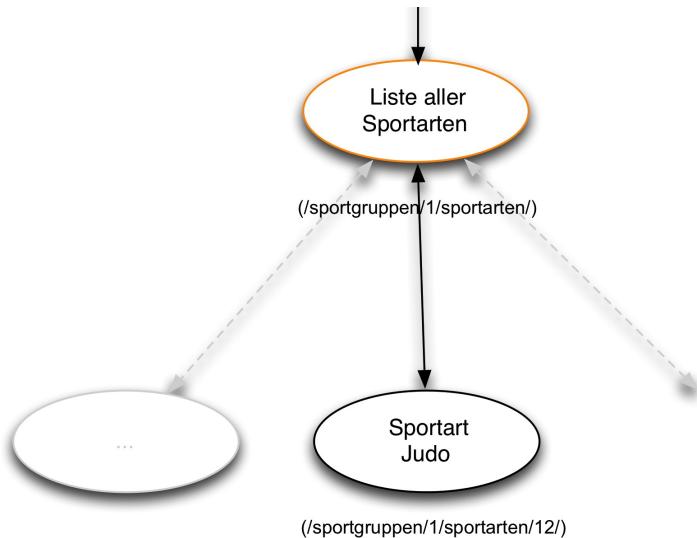


Abbildung 11: Konkrete Sportart

#### Liste aller Veranstaltungen (einer konkreten Sportart)

- URI: /sportgruppen/{id}/sportarten/{id}/veranstaltungen/
- Verbindung:
  - jede konkrete Veranstaltung (dieser Sportart)

Die Ressource ‘Liste aller Veranstaltungen’ ist eine Liste aller Veranstaltungen, einer bestimmten Sportart. Diese Liste enthält Verbindungen zu den einzelnen konkreten Veranstaltungen der zuvor spezifizierten Sportart.

*Beispiel: 1. Offene Kreiseinzelmeisterschaft U15, 1. Offene Kreiseinzelmeisterschaft U20, 2. Offene Kreiseinzelmeisterschaft U15, 3. Offene Kreiseinzelmeisterschaft U15,... (/sportarten/1/sportarten/12/veranstaltungen/) aus Judo (/sportgruppen/1/sportarten/12/)*

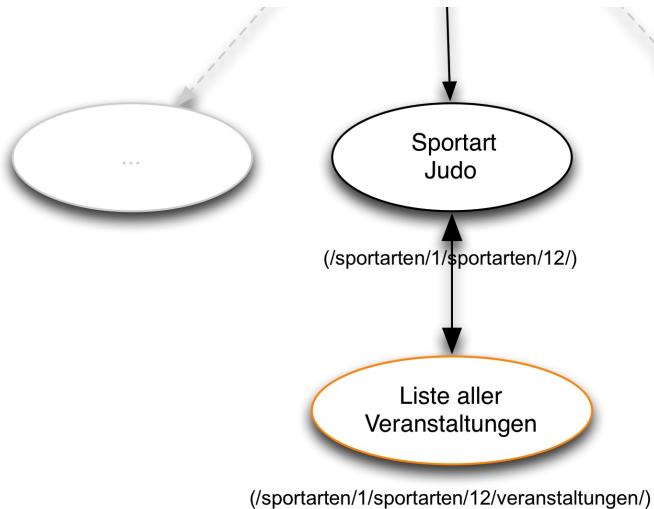


Abbildung 12: Liste aller Veranstaltungen

### Konkrete Veranstaltung

- URI: /sportgruppen{id}/sportarten/{id}/veranstaltungen/{id}/
- Verbindungen:
  - konkretes Gebäude
  - konkreter Veranstalter

Die Ressource ‘Konkrete Veranstaltung’ ist ein Teil von der Ressource ‘Liste aller Veranstaltungen’, die einer zuvor gewählten Sportart angehören. Eine Konkrete Veranstaltung ist verbunden mit dem Gebäude in welchem sie ausgetragen wird, sowie, mit dem zugehörigen Veranstalter, der die Leitung der Veranstaltung führt. Es bestünde die Möglichkeit, weitere Ressourcen zu einer Veranstaltung hinzuzufügen, wie zum Beispiel der Ort, in dem die Veranstaltung ausgetragen wird, oder das für die Veranstaltung benötigte Equipment. Der Verbindungen der Ressourcen wurden jedoch bewusst gewählt. Der Ort, an dem eine Veranstaltung ausgetragen wird, wird durch das Gebäude, mit welchem die Veranstaltung verbunden ist, ermittelt, da ein Gebäude fest an einem Ort geknüpft ist (siehe unten). Warum das Equipment keine Verbindung zu der Veranstaltung hat, wird im weiteren Verlauf erläutert.

*Beispiel: 1. Offene Kreiseinzelmeisterschaft U15 (/sportarten/1/sportarten/12/veranstaltungen/1) aus Veranstaltungsliste (/sportarten/1/sportarten/12/veranstaltungen/).*

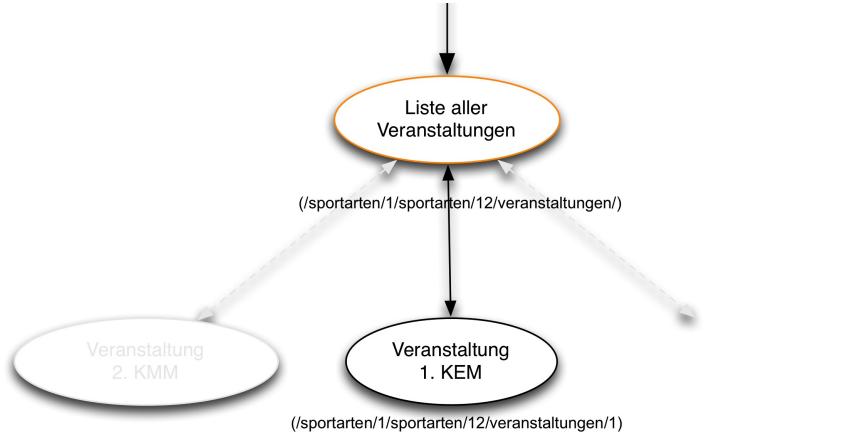


Abbildung 13: Konkrete Veranstaltung

### Liste aller Orte

- URI: /orte/
- Verbindung:
  - jeder konkreter Ort

Die Ressource ‘Liste aller Orte’ ist eine Liste aller konkreten Orte. Diese Ressource wurde definiert, um später die Möglichkeit zu bieten, nach einem bestimmten Ort zu suchen, bzw. sich zunächst einmal anzeigen zu lassen, welche Orte der Applikation überhaupt unterstützt werden.



Abbildung 14: Liste aller Orte

## Konkreter Ort

- URI: /orte/{id}/
- Verbindung:
  - Liste aller Gebäude (des konkreten Ortes)

Die Ressource ‘Konkreter Ort’ ist ein Teil von der Ressource ‘Liste aller Orte’. Der konkrete Ort ist verbunden mit einem oder mehreren Gebäuden, also auch hier mit einer Liste aller Gebäude, welche sich in dem konkret gewählten Ort befinden. So könnte nach der Ortswahl die Suche nach einem bestimmten Gebäude erleichtert werden, im selben Zuge können Gebäude, in denen Veranstaltungen ausgetragen werden, ein Ort zugewiesen werden. Dadurch ist es möglich, über dem Gebäude in dem eine/mehrere Veranstaltung/en ausgetragen wird/werden, einen Ort zuzuweisen. Des Weiteren kann ein Veranstalter über den Ort ein Gebäude finden (siehe unten). Dadurch hat er ebenfalls die Möglichkeit zu schauen, welches Equipment er für das Gebäude buchen könnte, bzw. welches bereits verliehen ist.

*Beispiel: Grevenbroich (/orte/4/) aus Ortsliste (/orte/)*

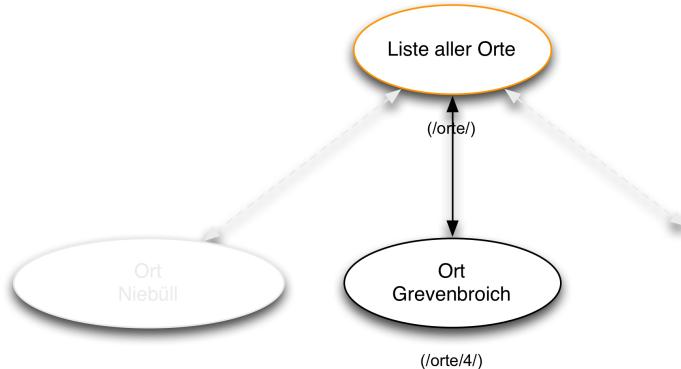


Abbildung 15: Konkreter Ort

## Liste aller Gebäude

- URI: /orte/{id}/gebaeude/
- Verbindung:
  - konkretes Gebäude

Die Ressource ‘Liste aller Gebäude’ ist eine Liste aller Gebäude eines bestimmten Ortes. Diese Liste enthält Verbindungen zu den einzelnen, konkreten Gebäuden des zuvor spezifizierten Ortes.

*Beispiel: Hans-Sachs Halle, Sporthalle 1, Tennishalle A, ... (/ort/4/gebaeude) aus Grevenbroich (/orte/4).*

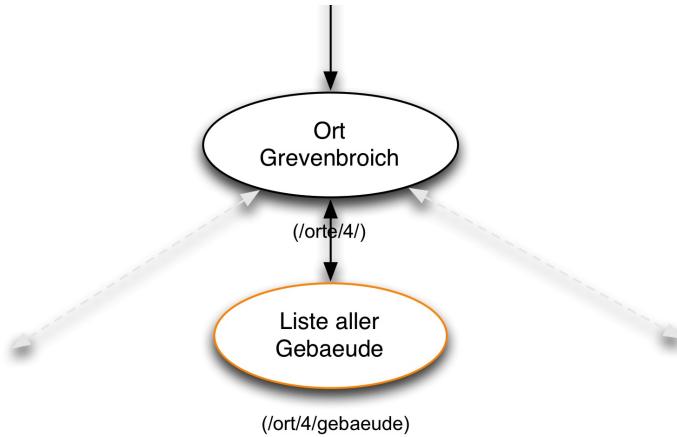


Abbildung 16: Liste aller Gebäude

### Konkretes Gebäude

- URI: /orte/{id}/gebaeude/{id} /
- Verbindung:
  - konkreterVeranstalter
  - konkrete Veranstaltung(en)
  - Liste mit Equipment

Die Ressource ‘Konkretes Gebäude’ ist ein Teil von der Ressource ‘Liste aller Gebäude’. Das konkrete Gebäude ist verbunden mit dem Equipment, welches in diesem Gebäude ist, bzw. zu diesem gehört. Des Weiteren ist das Gebäude mit keiner, einer oder mehreren Veranstaltung verbunden, sofern diese in dem konkreten Gebäude ausgetragen werden.

Auch hier wäre eine Listenressource möglich. Diese Listenressource wäre vergleichbar mit der bereits oben definierten Ressource ‘Liste aller Veranstaltungen’, mit dem Unterschied, dass diese nun nicht die Veranstaltungen einer Sportart, sondern einem Gebäude betreffen. Auf diese Listenressource wurde nun jedoch verzichtet, da im Regelfall eine Veranstaltung über die Sportart und

nicht über ein Gebäude gewählt wird. Zudem kann ein Veranstalter ein Gebäude für seine Veranstaltung reservieren.

*Beispiel: Hans-Sachs Halle (/orte/4/gebaeude/1) aus Gebäudeliste (/ort/4/gebaeude)*

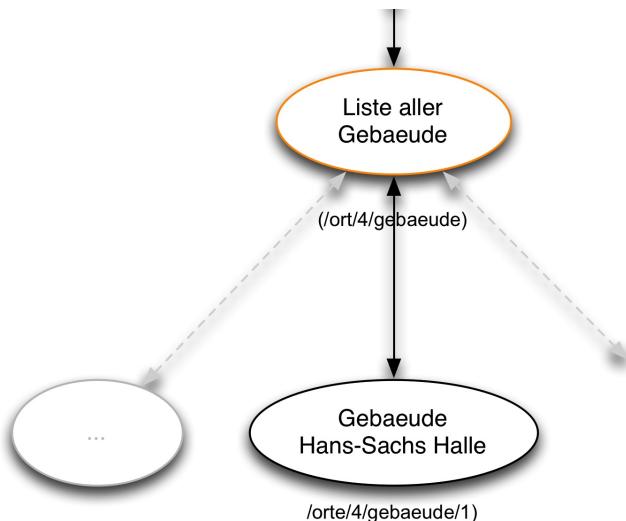


Abbildung 17: Konkretes Gebäude

### Liste mit Equipment

- URI: /orte/{id}/gebaeude/{id}/equipment/
- Verbindung:
  - konkretes Equipment

Die Ressource ‘Liste mit Equipment’ ist eine Liste welche jegliches Equipment eines bestimmten Gebäudes enthält.

*Beispiel: Judomatten, Fußbälle,... (/orte/4/gebaeude/1/equipment/) aus Hans-Sachs Halle (/ort/4/gebaeude/1)*

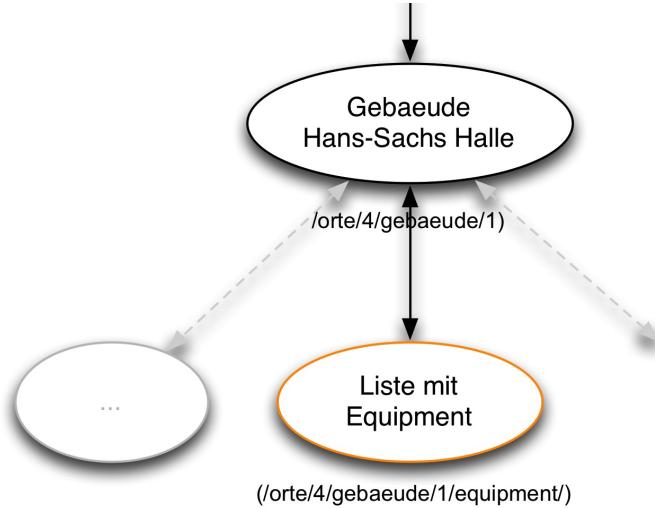


Abbildung 18: Liste mit Equipment

### Konkretes Equipment

- URI: /orte/{id}/gebaeude/{id}/equipment/{id}/
- Verbindungen:
  - konkreter Veranstalter
  - konkretes Gebäude

Die Ressource ‘Konkretes Equipment’ ist ein Teil von der Ressource ‘Liste mit Equipment’. Das konkrete Equipment ist verbunden mit dem konkreten Veranstalter, der es gerade verwendet/ausleiht. Es könnte gleichermaßen mit einer Veranstaltung oder Sportart verbunden werden, doch es wurde sich aus folgenden Gründen auf eine Verbindung zu einem konkreten Veranstalter geeinigt:

s

Sollte das Equipment an einer konkreten Sportart gebunden sein, so hätte dies den Vorteil, dass beispielsweise Fußball nicht für die Sportart Tennis reserviert werden können. Andererseits müsste jedoch zu jedem Equipment ein oder mehrere Sportarten zugewiesen werden, was bei beispielsweise “Sportmatten” einen nachteiligen Effekt auslösen würde. Da es im Endeffekt nicht relevant ist, von welcher Sportart (und auch Veranstaltung) ein konkretes Equipment reserviert wird, wird es dem Ausleihenden (dem konkreten Veranstalter) zugewiesen. Dies hat des Weiteren den Vorteil, dass, sollte ein gewisses Equipment zu jenem Zeitpunkt in Gebrauch beziehungsweise verplant sein, so kann direkt erkannt werden, an welchen Veranstalter sich der Interessent wenden muss, falls er es ebenfalls benötigt.

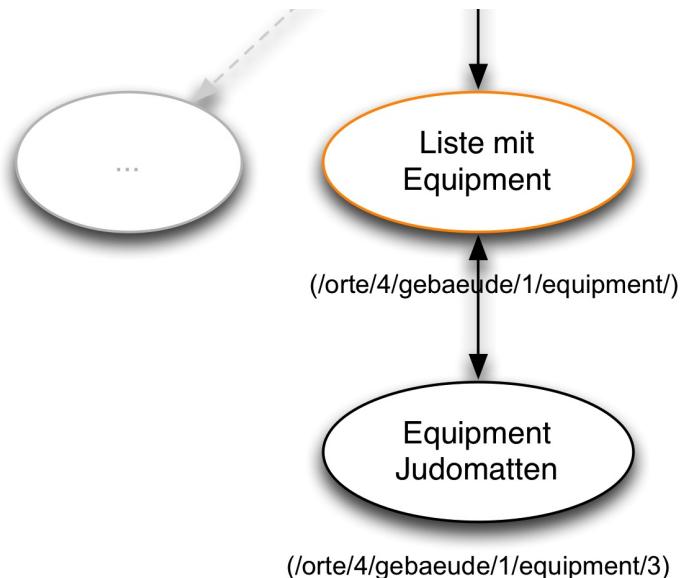


Abbildung 19: Konkretes Equipment

#### **Liste aller Veranstalter**

- URI: /veranstalter/
- Verbindungen:
  - konkreter Veranstalter

Die Ressource ‘Liste aller Veranstalter’ ist eine Liste aller konkreten Veranstalter. Diese Ressource wurde definiert, um später die Möglichkeit zu bieten, nach einen bestimmten Veranstalter zu suchen, bzw. zunächst erstmal anzeigen zu lassen, welche Veranstalter überhaupt registriert sind.



Abbildung 20: Liste aller Veranstalter

## Konkreter Veranstalter

- URI: /Veranstalter/{id}/
- Verbindungen:
  - konkrete Veranstaltung
  - konkretes Gebäude
  - reserviertes Equipment

Die Ressource ‘Konkreter Veranstalter’ ist ein Teil von der Ressource ‘Liste aller Veranstalter’. Der konkrete Veranstalter ist verbunden mit einem oder mehreren Gebäuden, Equipments und oder Veranstaltungen. Auch ein Veranstalter könnte später verfolgt werden; so kann dann ein Veranstalter abonniert werden, sollte es sich um einen favorisierten Veranstalter handeln. Dann besteht die Möglichkeit zu schauen, in welchem Gebäude dieser welche Veranstaltungen austrägt und auch, welches Equipment er gerade reserviert oder wofür er sich interessiert.

Interessenten einer Veranstaltung oder Sportart wurden nicht als Ressource identifiziert, da diese Benutzer keinen Teil des Systems darstellen und es auch keinerlei Informationen über Interessenten gibt, welche persistent in einer Ressource gespeichert werden müssten. Lediglich Authentifizierungsinformationen sowie die abonnierten Veranstaltungen oder Sportarten müssen gespeichert werden, doch diese Arbeit wird später vom XMPP-Server übernommen, sodass der Interessent keinerlei Bedeutung für den REST-Service darstellt.

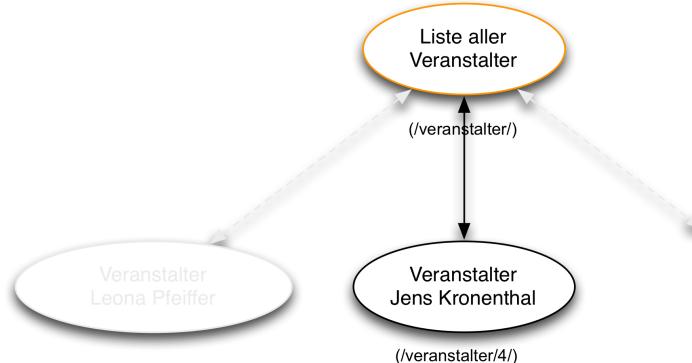


Abbildung 21: konkreter Veranstalter

Zur Veranschaulichung sind alle Ressourcen mit ihren Identifiern (URI) einmal in einer Tabelle aufgelistet.

Ressource	URI
Liste aller Sportgruppen	/sportgruppen/
Konkrete Sportgruppe	/sportgruppen/{id}
Liste aller Sportarten (einer Sportgruppe)	/sportgruppen/{id}/sportarten/
Konkrete Sportart	/sportgruppen/{id}/sportarten/{id}/
Liste aller Veranstaltungen (einer Sportart)	/sportgruppen/{id}/sportarten/{id}/veranstaltungen/
Konkrete Veranstaltung	/sportgruppen/{id}/sportarten/{id}/veranstaltungen/{id}/
Liste aller Orte	/orte/
Konkreter Ort	/orte/{id}/
Liste aller Gebäude (eines Ortes)	/orte/{id}/gebaeude/
Konkretes Gebäude	/orte/{id}/gebaeude/{id}/
Liste mit Equipment (eines Gebäudes)	/orte/{id}/gebaeude/{id}/equipment/
Konkretes Equipment	/orte/{id}/gebaeude/{id}/equipment/{id}/
Liste aller Veranstalter	/veranstalter/
Konkreter Veranstalter	/veranstalter/{id}/

Abbildung 22: Tabellarische Darstellung der Ressourcen

Die verschachtelten Verbindungsbeziehungen (siehe URI's) ergeben sich aus mehreren Gründen. Zum Einen ist es effizienter, nur benötigte Informationen anzufordern, sodass der Server entlastet wird. Es wäre nicht von Nöten, alle Sportgruppen und deren Sportarten anzufordern, wenn der Benutzer doch nur eine bestimmte Sportart sucht. Bei fast allen Ressourcen ist ein solches Verhalten erkennbar; es macht keinen Sinn, alle Veranstaltungen aller Sportarten anzufordern, wenn nur Veranstaltungen zu einer Sportart gesucht werden/jedes Equipment anzufordern, wenn nur das Equipment in einem Gebäude gewünscht ist, usw.. Zum Anderen erzeugen diese Verbindungen eine schöne aber auch logische Struktur. Es gibt keine Veranstaltungen, die mehreren Sportarten zugewiesen werden können, es gibt keine Sportarten, die mehreren Sportgruppen zugewiesen werden können, es gibt kein Equipment, welches nicht an ein bestimmtes Gebäude gebunden ist, es gibt kein Gebäude, welches nicht an einem bestimmten Ort gebunden ist, usw. Natürlich gibt es auch hier sicherlich Ausnahmen, wie z.B., dass eine Veranstaltung eben doch mehrere Sportarten betrifft und/oder gleichzeitig in mehreren Gebäuden stattfindet. Doch diese Ausnahmen werden hier nicht berücksichtigt, da dies eher selten der Fall ist. Als Lösung hierfür müsste die Veranstaltung zwei mal eingetragen werden.

Unabhängig von den Verschachtelungen und die dahergehende Strukturierung, sind natürlich auch Ressourcenverbindungen möglich, welche nicht aus der URI ersichtlich sind. So kann beispielsweise lediglich ein einzelnes Gebäude angefordert werden, welches mit jeder einzelnen Veranstaltung verbunden ist, welche in diesem Ausgetragen wird. Diese Verbindungen wurden oben bereits an der jeweiligen Ressource vermerkt. Zusätzlich zu den vermerkten Verbindungen, ist jede Ressource mit der jeweilig übergeordneten verbunden. Somit ist natürlicherweise eine Veranstaltung nicht nur mit dem Gebäude und Veranstalter, sondern auch mit der Sportart verbunden. Diese Verbindungsstruktur ergibt

sich, wie bereits erwähnt, aus den URI's der einzelnen Ressourcen und wurden aus diesem Grund nicht mit unter dem Punkt "Verbindung(en)" der Ressourcen aufgeführt.

### 3.1.3 Implementierung der Ressourcen

Im Anschluss galt es die Ressourcen zu implementieren. Hier wird auf das GitHub (Package: Ressourcen) verwiesen, in welchem sich die Implementierungen befinden. Hier kann man auch besonders die Implementierung der Methoden erkennen. So wird bei der Listen-Ressource für Veranstaltungen lediglich ein POST<sup>9</sup> ausgeführt, was ein neues Element in dieser Liste erstellen kann. Zur Erstellung wird keine ID benötigt; diese wird vom Server vergeben. Bei einer konkreten Ressource hingegen werden PUT und DELETE verwendet welche auf eine bestehende ID zugreifen um somit ein konkretes Element zu ändern oder zu löschen.

Wie in den Vorgaben beschrieben galt es das Marshalling und Unmarshalling, also das Umwandeln der Datenformate, mittels JAXB zu realisieren. Auch an dieser Stelle wieder ein Verweis ins GitHub - (Un-)Marshalling.

## 3.2 Verben

Wie bereits Eingangs erwähnt, ist es möglich auf den eben definierten Ressourcen Operationen anzuwenden. Diese Operationen werden im Folgenden REST/HTTP-Verben (kurz: Verben) genannt. Die HTTP-Verben beschreiben, wie sich eine Ressource bei Anfragen verhält bzw. was bei einer ausgeführten Operation für Zustandsänderungen geschehen. Für den REST-Architekturstil werden Standardverben vergeben. Unter anderem sind diese Standardverben folgende:

GET HEAD PUT POST DELETE

Damit der Architekturstil nicht gebrochen wird, werden auf den eben definierten Ressourcen ausschließlich diese vordefinierten Verben angewendet. Dementsprechend fällt die Option, eigene Operationen für die Ressourcen zu definieren, in dieser Projektarbeit weg.

Bevor die einzelnen Ressourcen ihre Verben zugeteilt bekommen, soll zunächst einmal klargestellt werden, worum es sich bei den eben genannten Standardverben genau handelt.

### 3.2.1 GET

Mit dem Verb GET ist es möglich, eine Ressource aus dem Web anzufordern. GET liefert dem Client dazu die entsprechend angeforderte Repräsentation der Ressource. Bei der GET-Anfrage findet keine Zustandsänderung statt, es werden lediglich Daten abgefragt. Da die häufigsten Anfragen im Web sich auf den lesenden Zugriff beziehen, spielt GET eine wichtige Rolle und ist entsprechend für das gesamte Web optimiert und beinhaltet zudem zusätzliche Spezifikationen wie das 'bedingte' GET. Das 'bedingte' GET (genauer: conditional GET)

---

<sup>9</sup>Siehe nächstes Kapitel für die Bedeutungen von HTTP-Verben.

fordert eine Repräsentation nur unter bestimmten Bedingungen an.[1, Seite 52] So kann beispielsweise eine Ressource nur dann angefordert werden, wenn sich diese seit einem gewissen Zeitpunkt geändert hat. Dies ist insofern nützlich, wenn eine Ressource angefordert werden soll, welche sich seit dem letzten Zugriff nicht verändert hat; sie muss nicht nochmals angefordert werden, sondern kann z.B. aus dem Cache geladen werden. Der Begriff 'Idempotenz' stellt zudem ein wichtiges Konzept für die HTTP-Verben dar. Mit 'Idempotenz' wird beschrieben, dass das mehrmalige Aufrufen einer Operation stets das selbe Ergebnis erzielt.[1, Seite 54] Dies ist bei der GET-Anfrage natürlich der Fall: Wird eine Ressource zwei mal hintereinander angefordert, wird diese (sofern sie sich nicht inzwischen geändert hat und kein conditional GET implementiert wurde) genau so repräsentiert, wie als wenn sie nur ein Mal angefordert wurde.

Das HTTP-Verb 'GET' sollte demnach für jede Ressource definiert werden, da GET das Standardverb der HTTP und Rest-Architektur ist.

### 3.2.2 HEAD

Anstelle der bedingten GET-Anfrage lässt sich auch das 'HEAD'-Verb verwenden um die Datenübertragungsmenge zu minimieren. Eine HEAD-Anfrage liefert lediglich den Header der zu repräsentierenden Ressource. Anhand dessen, kann der Client über die Metadaten der Ressource herausfinden, wann diese das letzte mal bearbeitet wurde. Sollte sich die Ressource seit der letzten Anfrage nicht verändert haben, so muss diese nicht erneut vom Server abgefragt werden.

### 3.2.3 PUT

Mit dem HTTP-Verb PUT wird eine Ressource, sofern sie bereits vorhanden ist, aktualisiert, andernfalls wird eine neue Ressource erzeugt. Auch PUT verfolgt das Konzept der Idempotenz; wird die PUT Anfrage erneut gesendet, so wird die Ressource entweder erneut aktualisiert oder nach dem Hinzufügen einer neuen Ressource, diese mit den selben Daten aktualisiert. Es ist also auch hier nicht relevant, ob die PUT-Anfrage ein mal oder 20 mal ausgeführt wurde.<sup>10</sup> PUT wird in dem zu entwickelnden System primär dazu genutzt Ressourcen zu aktualisieren. Das Hinzufügen einer Ressource wird von dem HTTP-Verb 'POST' übernommen. Der Unterschied hierbei ist, dass eine PUT-Anfrage die Ressource selber bestimmt (Beispiel: PUT .../veranstaltungen/23).

### 3.2.4 POST

Wie bereits beim HTTP-Verb PUT beschrieben, dient die POST-Anfrage dazu, eine neue Ressource anzulegen. Im Gegensatz zu der PUT-Anfrage wird bei der POST-Anfrage die Ressource nicht vom Client bestimmt, sondern vom Server. Somit werden oftmals vom Client lediglich Listenressourcen angesprochen, in denen dann der Server eine neue Ressource erstellt, welche die übermittelten

---

<sup>10</sup>Sollte die Anfrage nur einmal gesendet werden und geht dabei verloren, so macht dies natürlich schon einen Unterschied.

Informationen vom Client enthält. Der Client spricht eine Listenressource wie „..../veranstaltungen“ an, der Server erzeugt daraufhin beispielsweise die neue Ressource „..../veranstaltungen/24“ . Im Gegensatz zu den bereits kennengelernten HTTP-Verben, ist die POST-Anfrage nicht idempotent. Das heißt, wird eine neue Ressource mittels POST angelegt, so führt eine weitere POST-Anfrage dazu, dass erneut eine Ressource (mit anderer URI) angelegt wird.

Mit einer POST-Anfrage ist es zudem möglich, andere, selbst definierte Operationen durchzuführen. Dies wird jedoch, um den REST-Architekturstil zu wahren, nicht weiter beachtet und demnach auch nicht im System umgesetzt.

### 3.2.5 DELETE

Eine DELETE-Anfrage sorgt dafür, dass eine Ressource gelöscht wird. In diesem Fall wird die konkrete URI der zu löschenenden Ressource angegeben. Das Löschen einer Ressource führt im engeren Sinne nicht dazu, die Ressource wirklich zu löschen, sondern diese lediglich als gelöscht zu markieren. Es biete sich somit die Möglichkeit, Ressourcen beispielsweise mit dem Attribut 'gelöscht' zu versehen. Wird eine DELETE-Anfrage auf diese Ressourcen gestartet, so wird das Attribut den Wert 'true' zugewiesen<sup>11</sup>. Auch DELETE ist idempotent; das mehrmalige Setzen des Attributes 'gelöscht' auf 'true' bei mehreren Anfragen auf die selbe Ressource hat keinen unterschiedlichen Effekt.

### 3.2.6 Weitere HTTP-Verben

Natürlich wurden mehr, als die oben gelisteten Verben für HTTP spezifiziert. Da diese jedoch etwas komplexer sind und mit hoher Wahrscheinlichkeit nicht für das auszuarbeitende Projekt genügend Relevanz aufweisen, wurden sie der Einfachheit halber nicht gelistet. Mit den fünf oben definierten HTTP-Verben sind bereits ausreichende Operationen für einen strukturierten, durchdachten und komplexen Webservice möglich.

### 3.2.7 Zusammenfassung

Im Folgenden werden alle verwendeten Verben tabellarisch gelistet.

---

<sup>11</sup>In der GET-Anfrage sollten demnach keine Ressourcen, die den Attributwert 'true' aufweisen, gelistet werden.

Ressource	Verben
Liste aller Sportgruppen	GET, HEAD
Konkrete Sportgruppe	GET
Liste aller Sportarten (einer Sportgruppe)	GET, HEAD
Konkrete Sportart	GET
Liste aller Veranstaltungen (einer Sportart)	GET, HEAD, POST (um Veranstaltung hinzuzufügen)
Konkrete Veranstaltung	GET, PUT, DELETE
Liste aller Orte	GET, HEAD
Konkreter Ort	GET
Liste aller Gebäude (eines Ortes)	GET, HEAD
Konkretes Gebäude	GET
Liste mit Equipment (eines Gebäudes)	GET, HEAD
Konkretes Equipment	GET, PUT
Liste aller Veranstalter	GET, HEAD
Konkreter Veranstalter	GET

Auf Grundlage der Tabelle wird nun die Wahl der Methoden erläutert und begründet.

Zunächst zu den Listenressourcen, für welche die Methoden GET und HEAD implementiert wurden. Durch einen Aufruf der Methode HEAD kann geprüft werden, ob Veränderungen an der Ressource vorgenommen worden sind. Mittels GET würden die Veränderungen, falls vorhanden, geholt. Wie bereits erkennbar kann man lediglich eine konkrete Veranstaltung ändern oder löschen (PUT und DELETE). Dies hängt besonders mit dem eigentlichen Ziel der Anwendung zusammen. Die Liste aller Veranstaltungen wird durch die Methode POST ergänzt, sodass eine neue Veranstaltung erzeugt wird.

Denkbar wäre gewiss auch die Implementierung der POST-Methode bezüglich Equipment, welche allerdings an dieser Stelle nicht berücksichtigt wurde, da zum Einen der Fokus auf der Veranstaltungsressource liegt und zum Anderen diese Funktion nahezu so selten vorkommt, wie das Hinzufügen einer Sportgruppe oder Sportart, sodass diese Aktivität von einem Administrator übernommen werden sollte.

### 3.3 Realisierung des RESTful-Webservice Systems

Im Folgenden wird die Herangehensweise der Realisierung des RESTful-Webservices besprochen. Insbesondere wird die Umsetzung des Programmcodes und gegebenen Bibliotheken betrachtet. Dennoch sei auch hier auf das JavaDoc verwiesen<sup>12</sup>, welches ausführlich jede Methode und Klasse beschreibt. Teilweise sind in der JavaDoc sogar mehr Informationen zu finden, als die Kapselung es erlaubt; die Methoden werden nicht nur in ihrer reinen Funktionalität beschrieben, oft erläutert die JavaDoc, was wirklich hinter den Methoden geschieht. Aus diesem Grunde wird in dieser Dokumentation deutlich weniger auf die Inhalte, welche aus der JavaDoc entnommen werden können, eingegangen.

---

<sup>12</sup>Zu finden in dem Projektordner/docs

### 3.3.1 RESTful-Webservice

An dieser Stelle soll unmittelbar bereits auf den RESTful-Webservice eingegangen werden, da das Schema bereits erläutert wurde und die Vorgehensweise mit und vom Unmarshalling und Marshalling mittels JAXB aus der Dokumentation der ersten Projektphase entnommen werden kann.

Für die Anfertigung eines REST-Services wurden die ‘Jersey’- und die ‘HTTP-Server’-Bibliothek verwendet. Bei dem HTTP-Server handelt es sich nicht um Grizzly, sondern um die Standardbibliothek, gegeben von Java Sun. Es wurde auf diese zurückgegriffen, da die Dokumentation des Grizzly-Servers sich als unbrauchbar erwies. Dennoch liegt der Fokus nicht auf dem Server selbst, sondern auf dem Service, nämlich die Funktionen, die später vom Server realisiert werden soll. Beim Start des Servers sucht dieser nach Services, welcher er realisieren kann und stellt diese dann unter die gegebene Adresse bereit. Die für den REST-Services relevanten Java-Dateien finden sich in dem Package ‘restService’. Jedoch sei zu beachten, dass hier lediglich der Server und der Client realisiert werden. Der eigentliche Service wird in dem Package ‘ressourcen’ umgesetzt. In diesem Package finden sich alle Ressourcen wieder, wie sie eingangs definiert worden sind. Dementsprechend sind auch die auf die Ressourcen anzuwendenden Verben in je einer Methode realisiert. Als Beispiel wird die Java-Klasse ‘SportgruppenListe’ im Folgenden verwendet. In dieser Klasse befindet sich eine Methode. Im Konstruktur wird das Unmarshalling realisiert, was für diesen Teil der Dokumentation nicht relevant ist. Die Methode ‘getSportgruppen()’ beschreibt, was geschehen soll, wenn ein Client auf einen Server, der diesen Service umsetzt, mit einer GET-Anfrage zugreift. Interessanterweise fehlt hier die bei den Ressourcen genannte HTTP-Methode ‘HEAD’. Sie müsste theoretisch in einen eigene Methode realisiert werden. Das kann sie durchaus auch, doch der verwendete Webserver übernimmt diese Arbeit. Wird eine HTTP-HEAD Anfrage an den Server gesendet, kann dieser bereits ableiten, dass diese Anfrage sich auf die Ressource bezieht, auf der auch eine GET-Operation realisiert wurde. Der Server behandelt die HEAD-Anfrage sogesehen als einen Teil der GET-Anfrage und liefert dementsprechend nur den benötigten Header der angesprochenen Ressource. Alle anderen HTTP-Methoden sind zu jeder Ressource so umgesetzt, wie sie Eingangs im Kapitel Ressourcen definiert worden sind. Dennoch sind die Methoden PUT, DELETE und POST zu erwähnen.

**POST** Wie bereits bekannt, ist das HTTP-Verb ‘POST’ nur für die Ressource ‘VeranstaltungenListe’ definiert. Dies macht insofern Sinn, da nur dem Ressourcenobjekt einer Liste, eine weitere Veranstaltung hinzugefügt werden darf, da diese ja alle zugehörigen Veranstaltungen enthält. Wie bereits im Kapitel Verben erwähnt, wird die ID der neu angelegten Ressource, innerhalb der Listenressource, vom Server festgelegt<sup>13</sup>. Wird vom Client das Attribut-ID gesetzt, so wird der Server dieses überschreiben. Selbiges geschieht natürlich mit dem Attribut ‘deleted’ - so wird verhindert, dass eine neu hinzugefügte Veran-

---

<sup>13</sup> Die ID wird mit jeder angelegten Ressource inkrementiert.

staltung, bereits bei der Erstellung gelöscht ist; dies würde keinen Sinn machen.

**PUT und DELETE** Auch die HTTP-Verben PUT und DELETE wurden so umgesetzt, wie im Kapitel Verben beschrieben. Für diese Methoden benötigt der Client die eindeutige Veranstaltungs-ID, welche sich, nach dem Ändern oder Löschen der Veranstaltung nicht ändern darf und nicht ändern wird. Natürlich wird es später in der grafischen Benutzeroberfläche gar nicht die Möglichkeit geben, die ID der Veranstaltung zu ändern, doch das System soll nach Möglichkeit auf mit anderen Clients funktionieren können. Dementsprechend wird, wenn das neue Veranstaltungs-Objekt eine geänderte ID aufweist, stets wieder die alte ID bekommen. Des Weiteren ist es nicht möglich, eine Ressource mittels PUT erzeugen zu lassen. Dieser Vorgang wäre natürlich möglich, jedoch könnte dies zu Inkonsistenzen mit dem zu entwickelnden System führen, da die Identifier der Veranstaltungen sich inkrementell verhalten. Wird nun mittels PUT eine Veranstaltung mit selbstdefinierter ID erzeugt, so müsste diese ID umständlich vom System noch verarbeitet und ggf. abgefangen werden. Aus diesem Grunde wurde das Hinzufügen einer Veranstaltung mittels dem HTTP-Verb PUT unterbunden. Die Operation für das Löschen einer Veranstaltung wurde wie im Kapitel zum Schema erläutert, als logisches Löschen umgesetzt. So wird lediglich das Attribut ‘deleted’ auf ‘true’ gesetzt, und damit wird es für GET-Anfragen nicht weiter berücksichtigt.

**Pfad-Parameter und Abfrage-Parameter** Pfad-Parameter (im Weiteren ‘PathParam’ genannt), sind Zusätze zu einer URI, welche einen gesonderten Pfad ermöglichen. In dem Falle des umzusetzenden Systems ermöglichen Path-Params die Realisierung von unterschiedlichen Pfaden durch die Angabe der ID der geforderten Ressource. So kann eine Listenressource oder konkrete Ressource so umgesetzt werden, wie es im Kapitel Ressourcen, hinsichtlich der Verschachtelung der URI, (siehe dazu die Tabelle) gezeigt wurde. Dies hätte dann die Struktur: [Server]/Listenressource/konkrete Ressource/Listenressource/..., wobei die konkrete Ressource alsPathParam, angegeben wird. Da sich die Ressourcen durch die ID der Elemente identifizieren, werden somit die in der XML-Datei befindlichen ID's der Elemente alsPathParam verwendet um die entsprechenden Ressourcen anzusprechen. Beispielhaft könnte folgende URI entstehen:

`'127.0.0.1/sportgruppen/21/sportarten/4/veranstaltungen/324'`

Abfrage-Parameter (im Weiteren QueryParam genannt) ermöglichen eine Spezifizierung der Anfrage des Clients. So können mit QueryParams auf sehr einfache Art und Weise, die angeforderten Informationen nach Filterkriterien organisiert werden. Die Verwendung von QueryParams war für das zu erstellende System nicht vorgesehen, da die Elemente des XML-Schemas kaum Attribute aufweisen, für dessen Verwendung QueryParams nötigt wären. Die beiden Attribute

‘deleted’ (von einer Veranstaltung) und ‘entliehen’ (von einem Equipment) sollten durch die Programmlogik bei der Abfrage realisiert werden; es werden z.B. bei der GET-Abfrage keine Veranstaltungen berücksichtigt, die das Attribut ‘deleted = true’ aufweisen. Nach dem Attribut ‘entliehen’ eines Equipment, zu Fragen, sollte nicht notwendig sein, da der Veranstalter, welcher dieses Equipment entliehen hat, ebenfalls eine Verbindung zu diesem Equipment aufweisen muss. Des Weiteren würde es kaum Sinn machen, die Ressourcen mittels QueryParam anzusprechen, denn dann würde die URI von eben, so aussehen:

```
'127.0.0.1/sportgruppen/query?id=21/sportarten/query?id=4/veranstaltungen/query?id=324'
```

Für die Verschachtelung der einzelnen Elemente, würden sich die QueryParam somit ebenfalls nicht eignen, da PathParams hierfür deutlich übersichtlicher sind. Dennoch stellte sich im Nachhinein heraus, dass sich die QueryParams doch noch nützlich erweisen können. Sie wurden für das Attribut ‘deleted’ verwendet, da sie für die spätere Umsetzung der grafischen Benutzeroberfläche einen deutlichen Vorteil bieten; die explizite Abfrage nach demQueryParam ‘deleted = true’ erleichtert das Bestätigen eines Löschvorgangs einer Veranstaltung, sowie das Auffinden von gelöschten Veranstaltungen.

**Probleme bei der Umsetzung des RESTful-Webservices** Aufgrund von einigen Missverständnissen musste leider nahezu der gesamte RESTful-Webservice einmal umgeschrieben werden. Ein Problem bestand zunächst darin, dass aus der Aufgabenstellung des dritten Meilensteins nicht klar wurde, wie der Service getestet werden soll und ob die Entwicklung eines Clients dazu gehört oder nicht. So wurde zunächst ein Client mitentwickelt, welcher jedoch nach dem nächsten WBA-Beratungstermin wieder verworfen wurde bzw. die Entwicklung zunächst eingestellt wurde, da dieser gar nicht Bestandteil des Meilensteins war. Da sich der Client jedoch erst in einer frühen Entwicklungsphase befand, entschied sich dieser Fehler als nicht derartig gravierend; es wurde folgend mit einem vorgefertigten Client gearbeitet, welcher die HTTP-Methoden bereits kennt und umsetzt. Das wesentliche Problem bei dem dritten Meilenstein, beriet der Teil der Aufgabenstellung, der da lautete: “JAXB soll für das marshalling / unmarshalling verwendet werden.”[4, Zuletzt aufgerufen: 22.06.2013]. Aus diesem Satz ging hervor, JAXB serverseitig zu verwenden, denn sonst würde diese Information nicht bei der Erstellung des Services stehen. Dementsprechend wurden zunächst jegliche HTTP-Methoden bereits serverseitig unmarshalled. Das Ergebnis einer GET-Anfrage bestand dann nur aus einem Plaintext, der vom Server übermittelt wird; es wird kein XML-Format übergeben, der Server hat dieses bereits verarbeitet. Am vorletzten möglichen Beratungstermin ging dann hervor, dass das Unmarshalling hauptsächlich auf Clientseite übernommen werden soll. Darauf hin wurde jede einzelne Anfrage, außer die POST-, DELETE- und PUT-Operation, (da diese eine übergebene XML-Struktur verarbeiten und das XML-Problem ja nicht Clientseitig, sondern Serverseitig bestand), abgeändert.

## 4 XMPP

In einem weiteren Meilenstein ging es darum, die Realisierung der asynchronen Kommunikation grob zu planen. Hierzu sollten Leafs definiert und die Publisher beziehungsweise Subscriber eindeutig festgelegt werden. Zudem galt es den Payload zu bestimmen und den XMPP Server einzurichten.

Zur Realisierung dieser Aufgaben waren einige Einarbeitungen und Recherchen von Nöten. Es soll klargestellt werden, was XMPP eigentlich ist und wie man es verwendet.

Im Folgenden werden die Rechercheergebnisse dokumentiert und beschrieben.

XMPP (Extensible Messaging and Presence Protocol), bedeutet im weitesten Sinne übersetzt, ein erweiterndes Nachrichten und Anwesenheitsprotokoll und basiert auf dem XML Format; XMPP ermöglicht es XML Daten zu versenden und erhalten. Dies geschieht, als Besonderheit, in Echtzeit. Einer der Services, welcher für das zu entwickelnde System besonders wichtig erscheint, sind die Notifications. Diese ermöglichen die “one-to-many”-Benachrichtigungen. Im Allgemeinen bedeutet dies das publishen, welches später realisiert werden soll, was bedeutet, dass ein Veranstalter über das System vielen Abonnenten eine Veränderung mitteilt.

### 4.1 Recherche und Vorbereitung[6]

#### 4.1.1 Architektur:

Die Architektur, welche sich hinter XMPP verbirgt nennt sich dezentral. Das heißt, dass es mehrere Dispatcher gibt, von welchem Informationen erhalten werden können, statt, wie bei einer zentralen Architektur lediglich einen Dispatcher gibt, welcher die Informationen an die umliegenden Publisher und Subscriber verteilt.

#### 4.1.2 Streaming:

Eine der Grundideen hinter XMPP nennt sich Streaming: weitestens beschreibt dies eine Verbindung zwischen Client und Server welche die folgenden drei XML-Stanzas, oder auch Nachrichtenprimitive, beliebig oft austauschen kann:

1. <message/>: Beschreibt die Kernmethode von XMPP. Sie dient dem Erhalt beziehungsweise dem Versenden von Nachrichten.
2. <presence/>: Hierbei handelt es sich beinahe schon um ein Publish und Subscribe Methode. Es geht darum, dass ein Nutzer, welcher Benachrichtigungen über den Status eines anderen Nutzers erhält, sofern er diesen abonniert hat. Wie bereits erwähnt lassen sich einige Parallelen zu dem zu entwickelnden System erkennen.
3. <iq/>: Das Info/Query Stanza stellt im Groben ein Request/Response Schema dar. Somit wird der Nachrichtenaustausch durch Anfrage und

Antwort realisiert. Zudem weisen alle drei Stanzas die gleiche Attribute, to, from, id, type,auf. <to/>: Definiert den Empfänger anhand seiner JID. <from/>: Definiert den Absender durch Angabe seiner JID. <id/>: Identifiziert jedes Stanza eindeutig. <type/>: Gibt nähere Informationen bezüglich des Typen des Stanzas.

#### 4.1.3 Fat-Ping versus Light-Ping [5]

In Zusammenhang des Systems, und vorallem bezüglich des Payloads, wurde sich auch mit dem Thema Fat-Ping und Light-Ping auseinandergesetzt. Bei beiden Verfahren handelt es sich um den Erhalt von Informationen, sie unterscheiden sich lediglich in der Übertragung des Payloads. Im Grunde scheint der Unterschied zwischen Fat und Light einfach. Mittels Light-Ping wird lediglich eine Benachrichtigung über Veränderung des abonnierten Leafs versendet, wohingegen Fat-Ping den gesamten Inhalt des Leafs mit überträgt. Dabei hat der Client bei einem Light-Ping die Möglichkeit selber zu entscheiden, ob und wann er den eigentlichen geänderten Content erhalten möchte. Mittels Fat-Ping hingegen, wird dem Client zeitgleich mit Erhalt der Benachrichtigung der geänderte Content übermittelt.

Um beide Methoden auf dem System anzuwenden wurde darüber diskutiert, welchen Payload welche Methode wie übermitteln könnte.

Anhand eines Szenarios lässt sich dies durchspielen: Jenny L. hat die Sportart Volleyball abonniert, da Sie seit vielen Jahren in einem Verein spielt nun jedoch nach Gummersbach gezogen ist und hier nach Veranstaltungen Ausschau hält. Sie abonniert nun die Sportart Volleyball und wartet gespannt auf Neuigkeiten. Heiner B., der Trainer des Gummersbacher Volleyball-Verein nutzt seit einiger Zeit die Anwendung um seine Interessenten immer auf dem neusten Stand zu halten. Da Heiner B. leider in der nächsten Woche das Training absagen muss, publiziert er dies mit Hilfe der Anwendung. Nun zu den verschiedenen Möglichkeiten der Benachrichtigung über die Veränderung der Termine in der nächsten Woche. Zum Einen könnte Jenny L. beim Start des Systems eine Nachricht erhalten, dass Veränderungen in der Sportart Volleyball getätigt wurden. Diese sähe dann beispielsweise so aus: "Die Veranstaltung >Volleyball-Training wöchentlich< wurde geändert". Somit würde Jenny L. lediglich Bescheid wissen, dass sich etwas geändert hat und könnte, sofern Sie in der Woche die Veranstaltung besuchen wollte, die Veranstaltung aufrufen und nachsehen, was sich geändert hat.

Zum Anderen wäre eine Alternative eine Benachrichtigung der genauen Veränderung jedoch ohne Inhalt zu erhalten: "Das Datum der Veranstaltung >Volleyball-Training wöchentlich< wurde geändert". Dies wäre besonders dann von Vorteil, dass der Nutzer erkennen kann, ob für ihn Interessante Informationen verändert wurden. So wäre eine Änderung der Beschreibung der Veranstaltung wohl weniger wichtig als die Veränderung der Uhrzeit.

Eine dritte Alternative wäre die direkte Übermittlung des Inhalts der Veränderung. "In der Veranstaltung >Volleyball-Training wöchentlich< wurde das Datum von 14.07.2013 auf 21.07.2013 geändert". Hiermit hätte Jenny L. direkt

die Möglichkeit genau zu sehen was verändert wurde.

Im Zusammenhang des Systems hat man sich dazu entschieden, so gut es geht, den light-ping umzusetzen. Somit werden lediglich Veränderungen gepublished, welche sich der Client im Anschluss holt. Wie erwähnt besteht hier der Vorteil, dass der Client oder in dem Fall der Nutzer selber entscheiden kann ob er den aktualisierten Content erhalten möchte. Jedoch wurde das Light-Ping Verfahren ein wenig erweitert, sodass zusätzlich übermittelt wird, um welche Änderung es sich bei welchem XML-Element handelt.

#### 4.1.4 Leafs

Im Wesentlichen wird zwischen zwei Arten von Knoten entschieden: Leaf-Nodes und Collection-Nodes. Da die zu verwendende Smack-API jedoch lediglich Leaf-Nodes unterstützt bzw. Collection-Nodes nur eingeschränkt unterstützt werden, wird sich nur auf Leaf-Nodes konzentriert. Die Elemente eines Leaf-Nodes sind veröffentlicht beziehungsweise werden veröffentlicht. Es handelt sich somit um einen Knoten, dessen Items Inhalte enthalten. Da ein Leaf-Node kein Container Format ist, ist es nicht möglich, dass ein Leaf weitere Knoten oder gar Sammlungen enthält (collection-nodes).

Um nun die erstellten Leaf-Nodes zu erläutern zu können, ist Eingangs wichtig zu erwähnen, dass ein wesentlicher Unterschied der Leaf-Nodes von Veranstaltern zu den Leaf-Nodes der Interessenten (Teilnehmer) besteht. Demnach werden im Folgenden die Leaf-Nodes für die jeweilige Zielgruppe beschrieben und erläutert.

Die Interessenten sind jene, welche Informationen bezüglich Veranstaltungen einer bestimmten Sportart erhalten wollen. An dieser Stelle geht man davon aus, dass sich die gewünschten Informationen lediglich auf eine Sportart und deren Veranstaltungen beschränkt. Im Weiteren Verlauf wäre eine Abonnierung von Veranstaltern sicher denkbar, um konkrete Informationen aller Veranstaltungen eines Veranstalters zu erhalten. Jedoch wurde sich an dieser Stelle auf das Wesentliche, die Veranstaltungen, konzentriert. Somit kann man zwischen zwei großen Gruppen von Leaf-Nodes unterscheiden. Zum Einen zwischen allen Veranstaltungslisten (Listen-Ressource) aller Sportarten und zum Anderen zwischen allen konkreten Veranstaltungen.

Die Veranstaltungsliste ist eine Liste, welche alle Veranstaltungen beinhaltet. Interessiert sich ein Teilnehmer für alle Veranstaltungen einer Sportart, bietet sich ihm die Möglichkeit, eine Veranstaltungsliste (einer konkreten Sportart) zu abonnieren. Dadurch erhält er sofort eine Benachrichtigung, wenn eine neue Veranstaltung (zu dieser bestimmten Sportart) hinzugefügt wurde.

So wird dem Benutzer lediglich eine Mitteilung zugesandt, wenn eine neue Veranstaltung (zu der konkreten Sportart) erstellt wurde. Nicht aber wenn eine Veranstaltung, die womöglich für ihn uninteressant ist, geändert wurde. Dies hat den Vorteil, dass der Interessent lediglich über Neuigkeiten informiert wird, welche ihn betreffen und interessieren.

Um die Funktion weiter zu differenzieren ist es als zweites möglich eine konkrete Veranstaltung zu abonnieren. Es ist somit ermöglicht, dass ein Interessent

lediglich Veränderungen oder Löschungen bezüglich der von ihm abonnierten Veranstaltung erhält, nicht aber über alle Veranstaltungen dieser Sportart.

Dies bietet den Vorteil, dass der Interessent unmittelbar bei Veränderungen der Veranstaltung über diese informiert und benachrichtigt wird. Sollte sich also kurzfristig z.B. Zeit oder Ort ändern, werden alle Abonnenten der Veranstaltung rechtzeitig informiert.

Hier ist noch zu beachten, dass zeitgleich bei der Erstellung einer Veranstaltung beziehungsweise einer Veranstaltungsliste ein Leaf-Node für jede neue Veranstaltung definiert werden muss. Somit wird es bei der Implementierung erst möglich eine konkrete Veranstaltung beziehungsweise Veranstaltungsliste zu abonnieren. Denkbar wäre, im Hinblick auf eine weiterführende Arbeit, die Implementierung von Abonnements von beispielsweise Veranstaltern.

Um nun die Sicht des Veranstalters zu erläutern wird im Folgenden beschrieben, welche Leaf-Nodes für diesen festgelegt werden und damit auch, welche Topics er abonnieren kann.

Ein Veranstalter bekommt zusätzlich die Möglichkeit, Gebäude und Equipment zu abonnieren. Dies wäre für eine Art Reservierung von Vorteil. Zudem können die Hallen optimal genutzt werden, ohne viele Pausen zu haben.

Veranstalter haben eventuell Präferenzen zu bestimmten Gebäuden, sodass Benachrichtigungen über Veränderungen, beispielsweise die Belegung zu einer bestimmten Uhrzeit, erwünscht sind. Somit stellt auch jedes Gebäude einen einen Leaf-Node dar. Denkbar wäre auch ein Abonnement der Gebäudeliste. Da hierdurch allerdings zum Einen lediglich eine Benachrichtigung gesendet werden würde, wenn Gebäude gelöscht oder hinzugefügt wurden. Da dies eher seltener passiert, ist die Implementierung an der Stelle für die zu entwickelnde Anwendung wenig sinnvoll. Zum Anderen würde der Veranstalter wie bereits weiter oben beschrieben über alle Veränderungen mehrerer Gebäude informiert werden; somit möglicherweise auch über Gebäude welche ihn nicht betreffen oder interessieren. Daher wurde sich auf das Abonnieren von einem konkreten Gebäude beschränkt.

Als weiterer Leaf-Node wird Equipment bestimmt. So ist es dem Veranstalter möglich ein konkretes Equipment (beispielsweise Tennisschläger) zu abonnieren und benachrichtigt zu werden sobald Änderungen diesbezüglich vorgenommen worden sind. Vorteilhaft ist dies, da ein Veranstalter an dieser Stelle frühzeitig informiert wird, sobald ein Equipment für welches er sich interessiert verändert und damit zur Verfügung steht.

#### 4.1.5 Publisher versus Subscriber

Wie bereits zu erahnen ist, handelt es lediglich bei einem Veranstalter um einen Publisher. Dieser hat die Möglichkeit neue Veranstaltungen zu erstellen oder von ihm erstellte Veranstaltungen zu ändern.

Ein Interessent hingegen hat lediglich die Möglichkeit zu abonnieren und somit Informationen zu erhalten. Bereits zu Beginn wurde die Idee der Interessenten welche auch publishen können verworfen, da es sich bei dem zu entwickelnden System um eine zuverlässige Quelle der Informationserhaltung handeln soll. So

macht es nur Sinn Informationen direkt von dem zuständigen Veranstalter zu erhalten und nicht etwa von anderen Teilnehmern. Zudem ist der Veranstalter gleichzeitig auch Subscriber. Wie bereits erläutert stehen ihm in der Rolle als Veranstalter jedoch andere Topics zum subscriben zur Verfügung.

## 4.2 Realisierung des XMPP-Server Systems

Für die Umsetzung eines XMPP-Service wurde der Openfire-Server in Kombination mit der Java Smack API verwendet. Mit dem XMPP-Service soll die asynchrone Kommunikation bei Verwendung des Publish-Subscribe Paradigmas umgesetzt werden.

Zunächst wurde ein XMPP Manager geschrieben, welcher die Verbindung aufbaut und das Publishen und Subscriben ermöglicht. Die Smack-API bietet die Möglichkeit auf den Openfire-XMPP Server zuzugreifen. Mittels einem sogenannten Pub-Sub-Managers wird das Publish-Subscribe Paradigma umgesetzt.

### 4.2.1 Benutzerrollen und Leaf-Nodes

Der Interessent einer Veranstaltung oder einer Sportart, kann sich über die Änderungen der gewünschten Sportarten und Veranstaltungen benachrichtigen lassen. Der Interessent gibt dazu eine Subscription auf dem Node auf, zu dem er regelmäßig über Aktualisierungen benachrichtigt werden möchte. Der Veranstalter hat die Rolle des Publishers. Er veröffentlicht die Änderungen an seinen Veranstaltungen. Damit bekommt jeder Subscriber der zu ändernden Veranstaltung eine Benachrichtigung. Jeder Subscriber einer Sportart, bekommt eine Benachrichtigung, wenn innerhalb dieser Sportart eine Veranstaltung hinzugefügt oder gelöscht wird. Es ergeben sich demnach zwei Rollen. Der Einfachheit halber wurden dazu, zunächst lediglich zwei Benutzer auf dem Openfire Server eingerichtet; einen Interessenten und einen Veranstalter. Der XMPP-Service wurde so geschrieben, dass es natürlich auch mehrere Veranstalter und Interessenten geben kann, dafür sollte dann der richtige Benutzername und das Passwort bei der Methode ‘login(username, password)’ verwendet werden. Des Weiteren befinden sich im XMPP-Manager Methoden, welche im Gesamtsystem später nicht umgesetzt worden sind, jedoch für die Entwicklung und Weiterentwicklung des Systems durchaus nützlich sein können. Damit der XMPP-Server mit dem Publish-Subscribe Paradigma umgesetzt werden kann, müssen wie bereits in Kapitel 4.1.4 auf Seite 42 beschrieben, Leaf-Nodes definiert werden.<sup>14</sup> Da diese, gerade bei der Entwicklung oftmals störend anzulegen und wieder zu entfernen sind, wurden zwei Java Klassen geschrieben, welche alle vorhandenen Leafs löschen (deleteLeafScript), bzw. für jede Sportart und jede Veranstaltung einen LeafNode erzeugen.<sup>15</sup> Bereits hier wird auf den REST-Server zugegriffen, indem zuvor dann doch ein REST-Client geschrieben worden ist, welcher die Anfragen an den Server schickt. Bei dem Erstellen der Leaf-Nodes mittels der Java Klasse

---

<sup>14</sup>Aus zeitliche Gründen wurden zunächst nur Veranstaltungen und Sportarten als Leafs umgesetzt.

<sup>15</sup>Diese Datein sind in dem xmpService Package zu finden.

‘createLeafScript’ werden jedoch keine gelöschten Veranstaltungen beachtet, da dieses Script zur Initialisierung, also zum ersten Start des XMPP-Servers dienen soll, damit die Leaf-Nodes richtig hinzugefügt werden und dies nicht einzeln geschehen muss. Dabei wird dann natürlich vorausgesetzt, dass es noch keine gelöschten Veranstaltungen im System gibt.

Payload Der Payload für das umgesetzte System stellte einige Schwierigkeiten dar. Er wird nun verwendet, um das aktualisierte oder gelöschte Element zu identifizieren.

Sollte ein Veranstalter beispielsweise eine Veranstaltung hinzufügen, so würde sich dies auf die Sportartenliste, in der die Veranstaltung ausgetragen werden sollte, auswirken. Der Payload beinhaltet nun die Identifikation der

- Sportgruppe in der sich die Sportart der Veranstaltung befindet,
- Sportart, in der sich die Veranstaltung befindet,
- Veranstaltung die hinzugefügt wurde.

Zudem enthält der Payload einen Präfix, welcher Auskunft darüber gibt, welche Art der Änderung getätigt worden ist. So wird im Payload lediglich die Identifikation übermittelt, sodass der Client sich daraus die veränderte Ressourcenliste ableiten und aktualisieren kann.

Ein Problem, welches leider bisher noch nicht komplett gelöst werden konnte, stellt die Abfrage nach dem reinen übermittelten Payload auf der Subscriber-Seite dar. Leider konnten die WBA-Mentoren bei diesem Problem auch nicht helfen, sodass nun auf eine alternative Möglichkeit zurückgegriffen wurden. Der Payload wird nun in das Wurzelement des ‘eigentlichen’ Payload geschrieben, da auf ein Item die Methode ‘getElementType’ (liefert Wurzelement) anwendbar ist. Somit wurde über den Wurzelementnamen auf den Payload zugegriffen, sodass dieser dann verarbeitet werden kann. Eine andere Alternative wäre es, das Item in einen String zu konvertieren und mit ‘getItem.get([index])’ auslesen zu lassen und den zurückgegebenen String, der neben dem Payload auch noch Informationen zum Datum des Publishings und die Item-ID enthält, zu zerschneiden. Da diese Methode jedoch verglichen zu der eben genannten Lösung deutlich aufwendiger ist und eventuell nicht immer richtig funktioniert, wurde auf diesen Lösungsweg verzichtet.

Nachdem also ein Item mit Payload gepublisiert wurde, entnimmt der ItemListener des Subscribers automatisch das Wurzelement des Payloads, und damit den wirklich gewünschten Payload, und verarbeitet diesen. Es wird das Präfix ausgewertet um erkennen zu können, ob es sich um eine POST-, PUT- oder DELETE-Operation handelt (für genauere Informationen siehe JavaDoc). Anschließend wird die ID der Sportgruppe, Sportart und Veranstaltung ermittelt. Daraufhin wird eine Methode ausgeführt, welche über den REST-Server die Änderung holt und die Informationen darstellt. Aus diesem Verfahren könnte nun abgeleitet werden, dass es sich um einen Light-Ping handelt; Der Payload des Items meldet, dass Änderungen vorliegen, der Client holt sich diese Änderungen. Andererseits jedoch, wird im Payload schon erkenntlich gemacht, worum

es sich handelt indem das Präfix dafür Auskunft gibt, dies ist ein Merkmal für einen Fat-Ping. Da jedoch nicht die gesamte Änderung des betreffenden Elements, sondern lediglich nur übermittelt wird, um welche Änderung es sich bei welchem XML-Element handelt, wird dieses Verfahren eher dem Light-Ping Verfahren entsprechen.

#### 4.2.2 Probleme bei der Umsetzung des XMPP Servers

Die Verwendung der Smack-API brachte leider einige Probleme mit sich. Teilweise scheinen beinhaltete Methoden nicht funktionsfähig, oder nicht fertig umgesetzt. Aus diesem Grunde, konnte das Programm nicht vollständig funktionsfähig programmiert werden. Im Folgenden werden die Probleme erläutert und Lösungsvorschläge vorgestellt.

Zugestellte gepublisierte Items sollen nicht persistent gespeichert werden – sie werden es aber. Wenn sie zugestellt worden sind, werden sie trotzdem weiterhin auf dem Server gespeichert, d.h. mit jedem Login erhält der Benutzer immer die letzte Benachrichtigung, auch, wenn er diese bereits als er zuletzt online war, erhalten hat. Wenn in der XMPP-Konfiguration als Lösung, die persistente Speicherung ausgeschaltet wird, scheint es auf den ersten Blick, dieses Problem zu lösen. Doch das tut es leider nicht; alle Benachrichtigungen verfallen, wenn ein Benutzer nicht während der Benachrichtigung online ist. Kommt er dann offline, bekommt er keinerlei Informationen, dass sich seine Abonnements verändert haben<sup>16</sup>. Doch es besteht ein weiteres Problem, was die Situation verschärft: Es kann lediglich das letzte gepublisierte Item auf dem Server gespeichert werden. Es sei folgendes Szenario gegeben:

Wird eine Veranstaltung/Sportart abonniert, bekommt der Nutzer, bei persistenter Speicherung der Items, trotzdem immer die letzte Benachrichtigung des abonnierten Leafs, auch, wenn ihm diese Benachrichtigung im Moment gar nicht interessiert (weil dieser ja gerade erst die Veranstaltung/Sportart abonniert hat). Der Benutzer abonniert nun fünf Veranstaltungen und loggt sich aus. Angenommen es werden nun drei Veranstaltungen, zu denen er ein Abonnement abgeschlossen hat, verändert, so bekommt dieser beim nächsten Login lediglich die zeitlich letzte gepublisierte Benachrichtigung (anstelle der eigentlichen drei Benachrichtigungen, da drei Änderungen stattgefunden haben). Für dieses Problem gibt es leider keine Lösung; wenn die persistente Speicherung abgeschaltet wird, so erscheint immerhin am Anfang keine veraltete Meldung beim ersten mal Abonnieren, jedoch auch keine Benachrichtigung über die letzte Änderung nach dem Abonnieren und ausloggen. So entsteht in jedem Fall ein Problem, welches womöglich mit der aktuellen Smack-API nicht lösbar ist.

Da es leider bisher keinerlei Lösung zu diesem Problem gibt, wurde die persistente Speicherung dennoch beibehalten, damit die Asynchronität umgesetzt werden kann. Der Benutzer bekommt dann leider immer noch veraltete Daten beim ersten mal subscriben. Dennoch bekommt er eine Benachrichtigung beim einloggen, falls sich ein abonniert Leaf (Veranstaltung oder Sportart) geändert

---

<sup>16</sup>Vorausgesetzt diese haben sich auch geändert.

hat – jedoch leider nur die letzte Meldung.

#### 4.2.3 Kombination: REST- und XMPP-Server

Die beiden erstellten Server sollen später miteinander harmonisieren. Das heißt, wenn z.B. eine Veranstaltung geändert wird, muss der XMPP-Server diese Änderung wahrnehmen und an alle Subscriber dieser Veranstaltung senden, bzw. diese über die Änderung benachrichtigen.

Zunächst muss für den RESTful-Webservice einen Client geschrieben werden, welche die Anfragen GET, PUT, POST und DELETE senden und verarbeiten kann. Dieser Client befindet sich, wie der REST-Server selbst im Package ‘restService’. Die Funktionalität ist sehr simpel. Der Client realisiert die eben genannten HTTP-Anfragen und liefert das Ergebnis entsprechend als Rückgabewert. Zusätzlich zu reinen Serveranfragen dient die REST-Client Klasse dazu, die Daten für den REST-Server aufzubereiten. Beispielsweise, um eine Veranstaltung zu übertragen, besteht die Möglichkeit, diese in XML einheitlicher Darstellung zu übermitteln, oder aber, ein JAXB-Object zu versenden. Im Falle einer Veranstaltung, kann so die Methode ‘buildVeranstaltung’ verwendet werden, welche die für eine Veranstaltung relevanten Informationen verarbeitet und in ein JAXB-Object wandelt.

Zusätzlich zum REST-Client muss natürlich auch ein XMPP-Client erstellt werden. Da dies jedoch schon erläutert wurde, wird hier nicht näher darauf eingegangen.

Nun, da sowohl auf den REST-, als auch auf den XMPP-Server zugegriffen werden kann und Informationen verschickt und verarbeitet werden können, muss ein Weg gefunden werden, diese beiden Server zusammen mit einem Client zu verwenden. Schließlich wird im fünften Meilenstein eine grafische Benutzeroberfläche als Client benötigt, welche für beide Server funktionsfähig sein soll. Es sei folgendes Szenario gegeben: Der Interessent I hat die Sportart Judo abonniert. Er möchte stets über hinzugefügte oder gelöschte Veranstaltungen zu dieser Sportart informiert werden. Dazu sendet er mit seinem XMPP-Client eine Anfrage, sich zu dem Leaf, der die Veranstaltung Judo repräsentiert, zu abonnieren. Veranstalter V fügt nun eine neue Veranstaltung in der Sportart Judo hinzu. Dazu bedient er sich an dem REST-Client, und verschickt eine POST-Anfrage an den REST-Server, samt Veranstaltungsinformationen zu der neu zu erstellenden Veranstaltung. Wie bekommt nun der Interessent I die Benachrichtigung, dass der Veranstalter V eine neue Veranstaltung in der besagten Sportart hinzugefügt hat? Die beiden Server haben schließlich keinerlei Verbindung zu einander, sodass die Veranstaltung zwar zunächst hinzugefügt wird, der Interessent jedoch keinerlei Informationen darüber erhält. Aus diesem Grunde wurde die Java Klasse ‘CombinedServicesInteressent’ und ‘CombinedServices-Veranstalter’ erstellt<sup>17</sup>. Diese Klassen beinhalten alle Anfragemöglichkeiten für den XMPP als auch für den REST-Server. Im Folgenden wird die Funktionsweise genauer betrachtet: Die Klasse ‘CombinedServicesInteressent’ initialisiert

---

<sup>17</sup>CombinedServiceVeranstalter

sowohl vom REST-Client als auch vom XMPP-Manager ein Objekt, sodass innerhalb dieser Klasse auf beide Services zugegriffen werden kann. Bei diesen kombinierten Clientklassen werden zwischen Veranstalter und Interessent unterschieden, da diese eine unterschiedliche Zielsetzung zum System aufweisen und somit unterschiedliche Funktionen nutzen und auch nutzen dürfen. So darf der Interessent keine Veranstaltung ändern oder löschen, der Veranstalter hingegen schon. Der Interessent ist mit allen GET-Anfragen ausgestattet, die das System soweit hergibt. Das heißt, der REST-Client wurde kaum modifiziert; außer PUT, POST und DELETE Anfragen kann der Interessent den REST-Service vollständig nutzen. Zusätzlich zu der Abfrage nach Informationen auf dem REST-Server, hat der Interessent zudem die Möglichkeit sich für einen Leaf-Node zu subscriben, unzusubscribe, sowie alle Subscriptions anzeigen zu lassen. Dies dient dazu, sowohl sich zu einer Veranstaltung und oder Sportart als Interessent eintragen zu lassen, als auch dieses Interesse wieder zu lösen. Damit der Interessent Informationen darüber bekommt, zu welcher Veranstaltung oder Sportart er bereits ein Abonnement besitzt, kann die Methode ‘showSubscriptions’ aufgerufen werden. Natürlich sind sowohl beim Veranstalter als auch beim Interessent das Initialisieren des XMPP-Servers, für das Verbinden, Einloggen, usw., und auch das Ausloggen und Beenden der Verbindung vorgesehen. Der interessante Teil der Umsetzung einer Kombination der beiden Services findet sich in der Kombinations-Klasse für den Veranstalter wieder (CombinedServicesVeranstalter). Der Veranstalter hat zusätzlich zu den GET-Anfragen, wie sie dem Interessenten zur Verfügung stehen, die Möglichkeit, neue Veranstaltungen hinzuzufügen, Veranstaltungen zu ändern oder zu löschen. Die Umsetzung, des Veranstalters als Interessent für Equipment oder Gebäude wurde aus zeitlichen Gründen nicht umgesetzt. Daher bekommt der Veranstalter im Gegensatz zum Interessenten nicht die Möglichkeit sich zu einem Leaf-Node zu subscriben. Nun das Besondere an dieser Klasse: Der Veranstalter publisiert automatisch zu einem Leaf-Node, wenn er eine Veranstaltung hinzufügt. Die Methode POST (im Programm ‘postVeranstaltung’ genannt) sorgt dafür, dass nicht nur auf den REST-Server zugegriffen wird und dort eine neue Veranstaltung erzeugt wird, sondern auch, dass der XMPP-Server benachrichtigt wird. Wie bereits im Kapitel 4.1.4 beschrieben, bekommt jede Sportart einen Leaf-Node für Veranstaltungen die zu dieser Sportart ausgetragen werden. Dementsprechend wird gleichzeitig bei der Methode ‘postVeranstaltung’, die vom Veranstalter aufgerufen wird um eine Veranstaltung hinzuzufügen, ein Publishing zu dem entsprechenden Sportarten-Leaf getätig. Zudem wird ein neuer Leaf-Node zusammengesetzt, damit auch diese einzelne neu hinzugefügte Sportart abonniert werden kann. Bei der Methode zum Aktualisieren (‘putVeranstaltung’) sowie zum Löschen (‘deleteVeranstaltung’) einer Veranstaltung wird ebenfalls ein Leaf-Node angesprochen. Bei der Aktualisierung wird der Leaf-Node der Veranstaltung angesprochen, sodass Interessenten dieser Veranstaltung eine Benachrichtigung darüber bekommen. Beim Löschen einer Veranstaltung wird der zugehörige Sportgruppen-Leaf benachrichtigt, dass diese Sporgruppe nun eine Veranstaltung weniger beinhaltet, sowie der Abonnent dieser Veranstaltung bekommt eine Benachrichtigung, dass die Veranstaltung und damit auch sein Abonnement gelöscht wird. Hinweis:

Wird eine Veranstaltung gelöscht, so wird auch der entsprechende Leaf-Node gelöscht, sodass auch trotz dem logischem Löschen, also das weitere Vorhandensein der Veranstaltung nach dem Löschkvorgang, zu dieser Veranstaltung keinerlei Benachrichtigung mehr erscheinen kann.

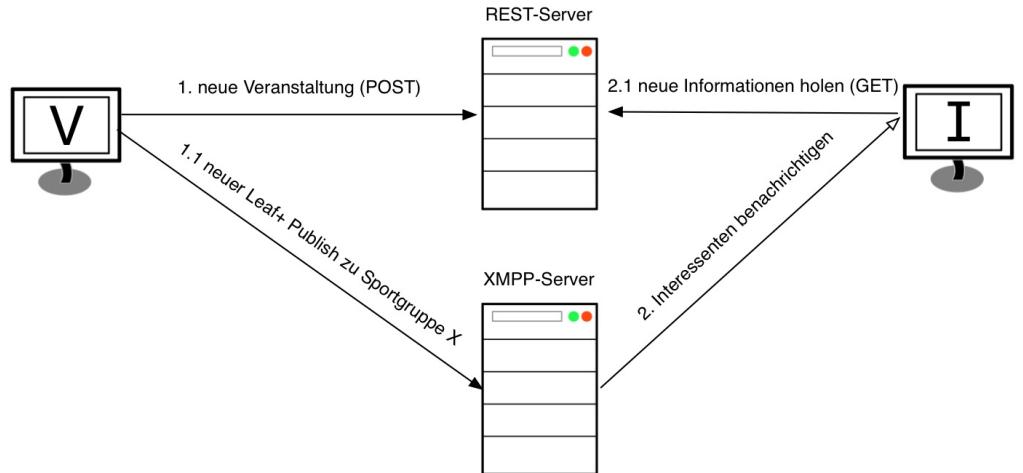


Abbildung 23:

Abbildung 23 stellt das eben erwähnte Szenario einmal bildlich dar, mit der Ausnahme, dass der Interessent I bereits zu der Sportart Judo ein Abonnement abgeschlossen hat: Der Veranstalter fügt eine neue Veranstaltung hinzu indem er gleichzeitig unbemerkt mit zwei Servern kommuniziert; Der REST-Server nimmt die neuen Informationen entgegen, die mittelst POST hinzugefügt worden sind (Schritt 1). Gleichzeitig wird an den XMPP-Server eine Nachricht an den entsprechenden Sportarten-Leaf, zu welcher die Veranstaltung hinzugefügt werden soll, gesendet (Schritt 1.1). Diese Nachricht enthält den Payload, wie er bereits beschrieben wurde. Somit weiß der XMPP-Service diesen Payload zuzuordnen und Informiert alle betreffenden Abonnenten (Schritt 2). Daraufhin empfängt der Interessent (asynchron) die Nachricht, dass seine abonnierte Sportart eine neue Veranstaltung enthält und holt sich diese Informationen dann vom REST-Server (Schritt 2.1).

#### 4.2.4 Zusammenstellung der Leaf-Nodes

Doch woher weiß der Veranstalter zu welchem Leaf-Node er publishen soll und woher weiß der Interessent, welcher Leaf-Node welche Veranstaltung repräsentiert? Die Leaf-Nodes haben, ähnlich wie der Payload eine Struktur, welche Aussagen darüber macht, um welche Veranstaltung oder Sportart es sich handelt. Es werden auch hier die ID's der XML-Elemente zur Identifikation verwendet. So kann ein Leaf-Node für die die Sportart durch die Sportgruppen-ID inklusive der Sportart-ID kenntlich gemacht werden. Zusätzlich wird noch am Ende

der zusammengesetzten ID's entweder die Zeichenkette "Sportart" oder "Veranstaltung" angehängt. In diesem Falle könnte demnach der Leaf-Node für die Sportart Judo so aussehen: "12Sportart". Der Interessent wird in der grafischen Benutzeroberfläche mit Dropdown-Feldern arbeiten, sodass dieser sich keinerlei Gedanken über die Leaf-Node Namen machen muss. Die Dropdown-Felder werden mit ihrem Index verwendet, dazu ein Beispiel: Wählt der Interessent die Sportgruppe Kampfsport im zweiten Eintrag des Dropdownfeldes wird der Index 1 gespeichert. Wählt der Interessent daraufhin bei der Sportart den dritten Eintrag, welcher für Judo steht, so wird der Index 2 gespeichert. Da der Interessent noch keine Veranstaltung zu dieser Sportart ausgewählt hat, wird davon ausgegangen, dass, sobald dieser einen Button zum Abonnieren betätigt, die Sportart abonniert werden soll. Es ergibt sich die Struktur "12Sportart" und der richtige Leaf wurde gewählt. Für genauere Informationen zu der Umsetzung der Leaf's wird hiermit auf die JavaDoc verwiesen.

### 4.3 GUI Entwicklung

Abschließend bestand die Aufgabe ein graphical user interface (GUI) zu entwickeln und zu implementieren. Dies ist mit Java Swing verwirklicht worden.

Wichtig ist zu erwähnen, dass es sich bei der GUI lediglich um eine vereinfachte visuelle Repräsentation der Anwendung ist und nicht alle Funktionen beinhaltet. An erster Stelle stand die Implementierung der gesamten Funktionalität im System und auf dem Server, sodass die Einbindung der Funktion in die GUI erst zum Schluss getätigkt wurde und sich somit auf das Minimum beschränkt worden ist. Hinzu kommt, dass weitaus mehr Methoden für die Services geschrieben wurden, als sie in der GUI Verwendung finden. Dennoch können diese Methoden frei verwendet werden bzw. bei Erweiterung des Systems zu einem späteren Zeitpunkt implementiert werden.

Nach einigen Recherchen hat man sich darauf geeinigt keinen GUI-Bilder zu verwenden sondern stattdessen die gesamte Oberfläche "per Hand" zu implementieren. Besonders in Hinblick darauf, dass bislang kaum Erfahrung mit Swing besteht und die Auseinandersetzung somit deutlich intensiver sind, wurde sich gegen einen Builder entschieden.

Auch wurde sich gegen ein Layout entschieden und somit das Null-Layout verwendet. Dies bietet dem Programmierer den Vorteil alle Komponenten exakt mittels Koordination zu positionieren. Hierbei ist drauf zu achten dass nach der Erzeugung der Komponente unbedingt eine Position (mittels beispielsweise `setBounds(x, y, width, height)`) angegeben werden muss. Wird dies nicht getan, wird die Komponente, trotz korrekter Erzeugung, nicht angezeigt. Zudem muss jede erzeugte Komponente in einem nächsten Schritt einer JPanel zugeordnet werden. Hierbei handelt es sich um sogenannte Container. Diese werden im Folgenden besonders dahingehend verwendet, dass für jeden Tab ein eigenes Panel angelegt wurde, um die spezifischen Inhalte darzustellen.

#### 4.3.1 Prototyp

Um ein möglichst reichhaltiges aber auch funktional hochwertiges, grafisches User Interface zu erhalten, wurde zu Beginn ein Prototyp angelegt, welche grob die möglichen Interaktionen skizziert. Anhand dieses "Fahrplans" wurde im Nachhinein die GUI umgesetzt. Hinzufügend ist anzumerken, dass die Idee der Tabs erst im Nachhinein entstanden ist und somit an dieser Stelle nicht aufgeführt wird.

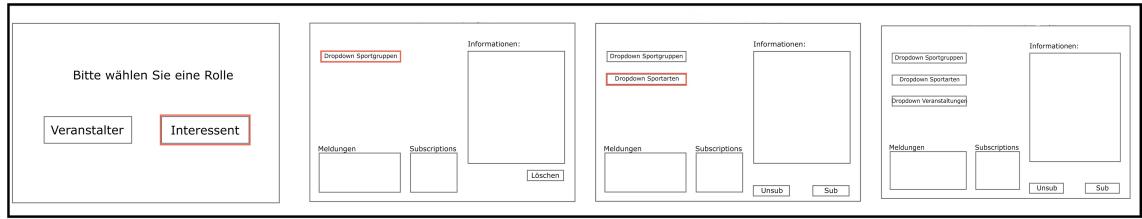


Abbildung 24: Prototyp - Rollenauswahl Interessent

Zu Beginn des Systems soll es die Möglichkeit geben, sich einzuloggen, so dass sich das spezielle Fenster für die Rolle öffnet. Dazu wird der Einfachheit halber, zunächst lediglich eine Abfrage zum Start des Systems erstellt, bei der der Benutzer zwischen Veranstalter und Interessenten wählen kann.

In Abbildung 24 wird die Rolle des Interessenten gewählt, welcher ein Fenster bekommt in welchem er mittels eines Dropdown-Feldes eine Sportgruppe wählen kann. Sobald dies erfolgt ist erscheint zum Einen ein weiteres Dropdown-Feld zur Wahl der Sportart und zum Anderen werden Informationen bezüglich der Sportgruppe in ein Textfeld (rechts) geschrieben. Diese Informationen sollen jene der unmarshalten XML-Datei sein. Auch bei der Wahl einer Sportart erscheint ein weiteres Dropdown-Feld zur Wahl von Veranstaltungen sowie der aktualisierte Text im Feld. Nun sollen die Informationen bezüglich der Sportart präsentiert werden. Zusätzlich dazu werden Subscribe und Unsubscribe Button eingeblendet welche sowohl die Abonnierung einer Sportart als auch die Abonnierung einer Veranstaltung, sofern diese im Dropdown gewählt wurde, ermöglicht. In einem weiteren Textfeld sollen alle für den Interessenten relevanten Mitteilungen stehen. Zum Einen Meldungen bezüglich des subscriben und unsubscriben zum Anderen aber auch Veränderungen an abonnierten Leafs. Das weitere Textfeld Subscriptions stellt alle abonnierten Leafs des Interessenten dar.

Die Oberfläche des Veranstalters wird, besonders in der Realisierung ein wenig aufwändiger, ganz einfach aufgrund der erweiterten Funktionalitäten, wie das Hinzufügen, Ändern oder Löschen einer Veranstaltung.

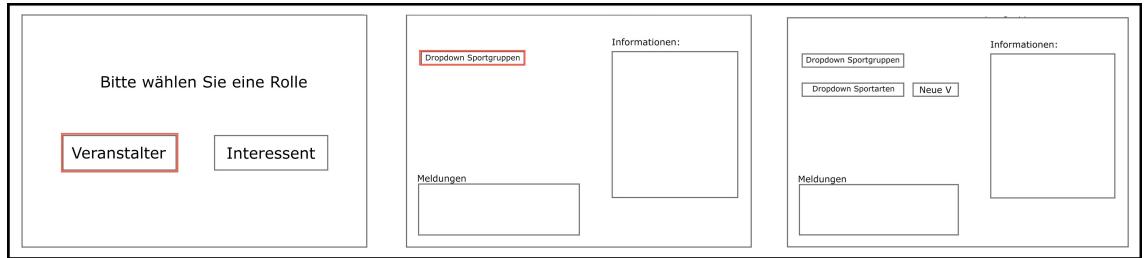


Abbildung 25: Prototyp Rollenauswahl Veranstalter

Abbildung 25 zeigt erneut den Startbildschirm des Prototypen nur, dass, an dieser Stelle der Veranstalter gewählt wird. Dieser soll zu Beginn die gleichen Interaktionsmöglichkeiten wie der Interessent erhalten. Es unterscheidet sich lediglich im Textfeld der Subscriptions welche in diesem Kontext keine Verwendung hätten. Auch hier kann der Benutzer eine Sportgruppen auswählen und erhält ein weiteres Dropdown-Feld. Zusätzlich dazu kann der Veranstalter, sobald er eine konkrete Sportart gewählt hat eine neue Veranstaltung zu dieser erstellen. Dies soll durch einen weiteren Button realisiert werden.

Sobald der Button zur Erstellung einer neuen Veranstaltung gewählt wurde, erscheint eine Art Formular, in welchem alle Angaben bezüglich einer Veranstaltung gemacht werden können. Dies kann abgebrochen oder bestätigt werden.

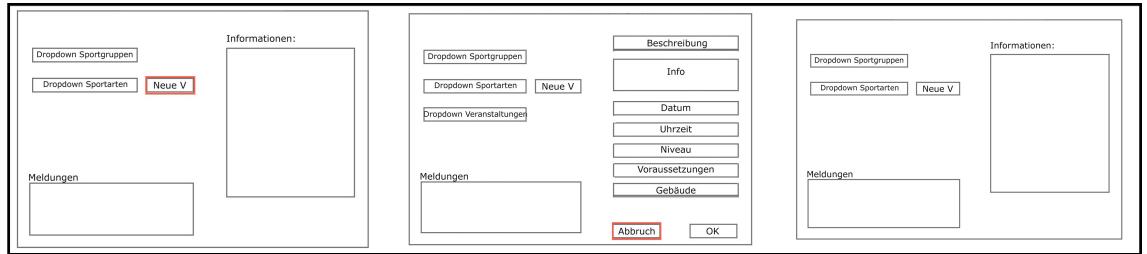


Abbildung 26: Prototyp - Erstellung einer neuen Veranstaltung

Sofern allerdings eine konkrete Veranstaltung gewählt wird, erhält der Veranstalter die Möglichkeit diese Veranstaltung zu ändern oder zu löschen. Auch dies kann wieder abgebrochen oder bestätigt werden.

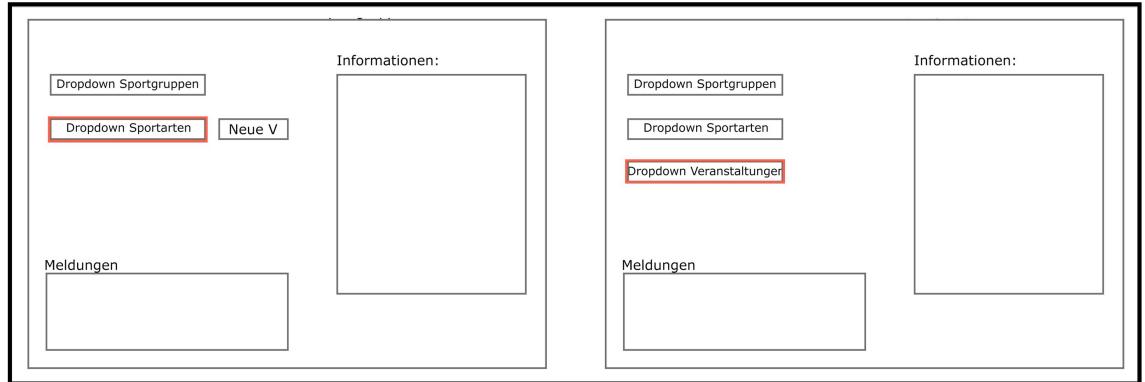


Abbildung 27: Prototyp Veranstaltung auswählen

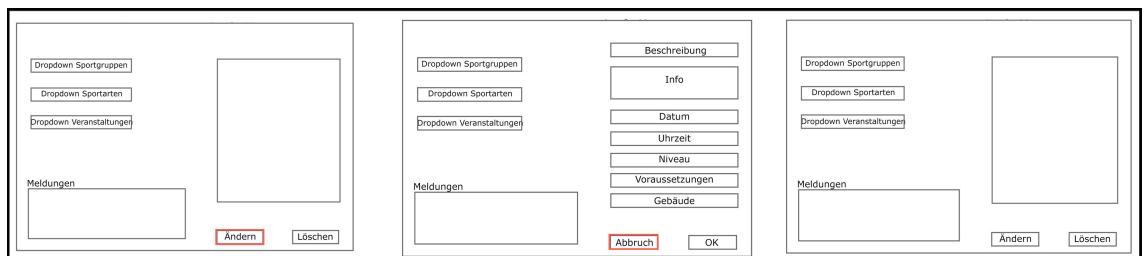


Abbildung 28: Prototyp Veranstaltung ändern (abbrechen vs. bestätigen)

#### 4.3.2 Realisierung und Umsetzung des GUI

Die grafische Benutzeroberfläche spielt, verglichen zu dem REST und XMPP-Service eine deutlich geringere Rolle. Es wurden nicht alle Funktionalitäten, welche der umgesetzte XMPP-Service unterstützt, für die GUI implementiert, da die GUI lediglich als visuelle Repräsentation und Veranschaulichung der dahinterstehenden Services dient. Dementsprechend wurde in der GUI darauf verzichtet, einen Servernamen angeben zu können sowie sich mit eigenem Benutzernamen anmelden zu können, obwohl die Funktionalität dafür gegeben ist. Demnach ist es nur möglich zwischen Veranstalter und Interessent zu wählen, nicht aber erhält der Benutzer die Möglichkeit sich einzuloggen beziehungsweise zu authentifizieren. Dies wurde der Einfachheit halber nicht angezeigt und geschieht im Hintergrund im Programmcode automatisch. Jene Auswahl wurde durch eine OptionPane verwirklicht. Somit wird zunächst zu Beginn die Rolle des Nutzers erfragt und entsprechend die grafische Oberfläche geladen. Diese unterscheidet sich, wie bereits die Abonnements zwischen Interessent und Veranstalter. Des Weiteren wurde aufgrund von Zeitmangel auf die Implementierung einiger Funktionen der Veranstalter-Oberfläche verzichtet. So ist es zum jetzigen Zeitpunkt als Veranstalter lediglich möglich, neue Veranstaltungen hin-

zuzufügen oder bestehende zu ändern oder zu löschen. Wie bereits im Abschnitt der Leaf-Nodes erwähnt wurde, wäre zusätzlich die Implementierung der Abonnements von Gebäuden und Equipment als Zusatzfunktion von Vorteil. Diese wurde allerdings, wie eingangs erwähnt, nicht implementiert. Da die eigentliche Funktion der Anwendung bei den Veranstaltungen liegt, sind Equipment und Gebäude (beziehungsweise Orte) nur nebенächlich und zeigen keine weitergehende Funktionalität, sodass auf diese zunächst ohne Eibüßen verzichtet werden konnte.

#### 4.3.3 Oberfläche des Interessenten

Der Sportinteressierte hat mittels Tab-Navigation die Möglichkeit zwischen Sportgruppen, Veranstaltern und Orten zu wählen. Über den Tab der Sportgruppen gelangt man zur eigentlichen Funktionalität des Systems. Mit Hilfe einer Dropdown-Liste (JCombo Box) kann ein Interessent zwischen allen vorhandenen Sportgruppen wählen. Diese kann jedoch, wie bereits beschrieben nicht abonniert werden - weiterhin erscheint ein weiteres Dropdown-Feld, in welchem alle Sportarten der gewählten Sportgruppe erscheinen. Gleichzeitig werden Button zum Subscriben und Unsubscriben angezeigt und ein neues Dropdown-Feld erzeugt. Dieses beinhaltet alle Veranstaltung bezüglich der gewählten Sportart. Auch diese kann man abonnieren beziehungsweise entabonnieren. Zudem wurden Logout Button realisiert, welche den Nutzer vom System ausloggen. Zudem ist der Rollenwechsel durch den Button 'Logout' realisiert. Dieser würde jedoch dem Benutzer im finalen System nicht zur Verfügung stehen, da man entweder Interessent oder Veranstalter ist. Sobald man sich für eine Veranstaltung interessiert würde man also die Rolle des Interessenten annehmen. Zur Vereinfachung des Systems wurde dieser Wechsel jedoch realisiert.<sup>18</sup>

Des Weiteren wird eine JTextArea angezeigt, welche alle relevanten Informationen enthält. Diese werden im Weiteren aus der XML-Datei gelesen. Informationen sind jene, welche im Schema beschrieben wurden. Eine weitere TextArea wurde zwecks Mitteilungen und Bestätigungen realisiert. Beispielsweise würden Veränderungen der abonnierten Leaf-Nodes an dieser Stelle angezeigt werden. Auch Benachrichtigungen über das erfolgreiche Abonnieren oder Entabonnieren werden an dieser Stelle realisiert. Diese wechseln, ebenfalls, wie die zuvor vorgestellten TextAreas, ihren Inhalt je nach Aktualisierung. Zudem wird mit Hilfe einer JList eine Aufzählung aller bisherigen Subscriptions realisiert. Somit kann ein Interessent sehen, welche Elemente er bereits abonniert hat.

Wie Eingangs bereits erwähnt wurde auf die Implementierung der Tabs Veranstalter und Orte verzichtet. Hier werden lediglich die Komponenten dargestellt, weisen jedoch keine Funktionalität auf.

#### 4.3.4 Oberfläche des Veranstalters

Auch bei der Oberfläche des Veranstalters wurde sich zunächst auf die Veranstaltungen und deren Änderung beziehungsweise Erstellung konzentriert. Erneut

---

<sup>18</sup>Jedoch funktionier der Button noch nicht wie gewünscht.

stellen die Tabs Equipment und Gebäude lediglich eine Zusatzfunktion dar und schienen somit zweitrangig. Wie bei den Interessenten baut sich die Auswahl aller Elemente beim Veranstalter durch Dropdown-Felder auf. Zu unterscheiden ist jedoch die Möglichkeit eine neue Veranstaltung hinzuzufügen. Sobald eine konkrete Sportart gewählt wurde kann eine neue Veranstaltung zu dieser Sportart erstellt werden. Dazu werden JTextFields / JTextAreas und JComboBoxes verwendet. Nun kann der Veranstalter alle notwendigen Daten wie z.B. Beschreibung, Info und Voraussetzungen (gemäß dem Schema) eintragen. Die Bestätigung sowie das Abbrechen der Erstellung wird erneut durch Buttons realisiert.

Im Weiteren Verlauf ist es möglich, dass der Nutzer, in der Rolle des Veranstalters, eine konkrete Veranstaltung, mittels des eingeblendeten Dropdown-Feldes, auswählt. Der Benutzer kann die dann gewählte Veranstaltung im Anschluss löschen oder ändern. Wichtig ist besonders, dass ein Veranstalter nur die von ihm erstellen Veranstaltungen ändern oder löschen kann. Da jedoch in der grafischen Benutzeroberfläche der Login, wie bereits erwähnt, im Hintergrund abläuft, wird hier nur zwischen einem Veranstalter und einem Interessenten unterschieden. Die dahinterstehenden Services ermöglichen natürlich unterschiedliche Benutzer. So kann und muss einer Veranstaltung einen Veranstalter zugeordnet werden, worauf jedoch für die beispielhafte Repräsentation mittels der GUI zunächst verzichtet wurde. Diese Funktionalität ließe sich jedoch ohne weitere Änderungen an den XMPP-Service zu einem späterem Zeitpunkt realisieren, da die Methoden hierfür bereits angefertigt worden sind. Entsprechend besteht für die Benutzeroberfläche momentan nur genau ein Veranstalter, welcher jegliche Veranstaltung ändern oder auch löschen darf.

#### 4.3.5 Probleme

Die GUI hat wohl deutlich mehr Aufwand und Bearbeitungszeit beinhaltet hat als zuvor vermutet. Besonders durch die Verwendung des Null-Layouts war der Aufwand der exakten Positionierung aller Elemente enorm aufwändig und zeitraubend. Des Weiteren ist die Gestaltung einer Oberfläche für mehr als einen Benutzer, wobei jeder Benutzer nochmals eigene Tabs und somit das z.B. selbe Dropdown-Feld mehrfach programmiert werden muss, unglaublich unübersichtlich in Java Swing. Wie bereits im Vorfeld erwähnt wurde auch auf einen GUI-BUILDER verzichtet. Vorallem in Anbetracht der Komplexität der GUI hätte es eventuell sinniger und besonders einfacher sein können einen GUI-BUILDER zu verwenden. Allerdings lässt sich im Gegenzug sagen, dass man sich somit tiefstmöglich mit Java Swing und deren Komponenten befasst hat.

## 5 Fazit und Ausblick

Abschließend ist anzumerken, dass der enorme Arbeitsaufwand zu Beginn eventuell unterschätzt oder falsch bedacht wurde. Dies hatte jedoch den Vorteil dass sich das ganze Projekt als eine große Herausforderung darstellte. Die Möglichkeit einem Veranstalter Subscriptions zur Verfügung zu stellen, damit dieser sich über Gebäude oder Equipment informieren und benachrichtigen lassen kann, wurde aufgrund der mangelnden Zeit vernachlässigt.

Obwohl sich zeitlich an alle Meilensteine gehalten wurde und zu den jeweilig empfohlenen Termine der dazugehörige Meilenstein erfolgreich abgeschlossen war, wurde es besonders zum Ende zeitlich sehr knapp, sodass auf einige Funktionalitäten verzichtet werden musste. Wie allerdings, wie des Öfteren an einigen Stellen in der Dokumentation vermerkt, besteht die Funktionalität größtenteils wie geplant, jedoch wurde sie lediglich aus zeitlichen Gründen nicht mehr in die GUI implementiert. Dies stellt somit auch einen Ausblick, Verbesserung und Potential zur Erweiterung dar.

## Literatur

- [1] Stefan Tilkov, REST und HTTP - Einsatz der Architektur des Web für Integrationsszenarien, dpunkt.verlag, 2. Auflage, 2011
- [2] Web-basierte Anwendungen 2: Verteilte Systeme, Foliensatz K3, Dr.
- [3] Web-basierte Anwendungen 2: Verteilte Systeme, Foliensatz, Dr. Fischer
- [4] [http://www.medieninformatik.fh-koeln.de/w/index.php/WBA2\\\_{\}So-Se13:Phase2](http://www.medieninformatik.fh-koeln.de/w/index.php/WBA2\_{\}So-Se13:Phase2)
- [5] <https://code.google.com/p/pubsubhubbub/wiki/ComparingProtocols>
- [6] Peter Saint-Andre, XMPP: The Definitive Guide Building Real-Time Applications with Jabber Technologies, O'Reilly, 1. Auflage, 2009