**Web-basierte Anwendungen 2:**

**Verteilte Systeme**

Fachhochschule Köln Campus Gummersbach

Fakultät für Informatik und Ingenieurwissenschaften

**Phase 2 - Projekt "Social Ticker"**

**Dokumentation**

**Autoren**

Dario Vizzaccaro 11085033

Benedikt Wurth 11084022

**Inhaltsverzeichnis**

Social Ticker 4

Einleitung 4

Projektidee 4

Projektspezifisches XML Schema 5

Userprofiles: 5

Events 7

Eventcontent 8

Kommunikationsabläufe und Interaktion 10

Synchrone Datenübertragung 10

Asynchrone Datenübertragung 10

Ressourcen und die Semantik der HTTP-Operationen 11

Ressourcen: 11

HTTP-Operationen (Verben): 11

User: 11

Admin (Ersteller eines Events): 12

RESTful Webservice 13

Grundlagen zur Implementierung der Operationen 13

Implementierung der GET-Operationen 13

Implementierung der POST-Operationen 13

Implementierung der PUT-Operationen 14

Implementierung der DELETE-Operationen 14

PathParams 14

QueryParams 15

Clientseitige Operationen 16

Clientseitige GET-Abfrage 16

Clientseitige POST-Methode 16

Konzeption + XMPP Server einrichten 17

Leafs (Topics) 17

Publisher 17

Subscriber 17

Zu übertragene Daten 17

XMPP Server 17

XMPP - Client 18

Erstellen von Nodes 18

Abonnieren von Nodes 18

Nachrichten veröffentlichen & Übertragung von Nutzdaten (Payload) 18

Service Discovery 18

Nachrichten empfangen 19

Client – Entwicklung 20

Entscheidung für JavaFX 20

Graphical User Interface (GUI) 20

Abschluss 21

Fehlende Features 21

Quellen 22

# Social Ticker

## Einleitung

Im Rahmen der Veranstaltung Webbasierte Anwendungen 2 sollte ein Projekt über mehrere Monate verwirklicht werden. Dieses Projekt befasst sich mit hauptsächlich mit der Kommunikation zwischen Server und Client und wie Daten in Form von XML-Dateien transportiert werden. Zusätzlich sollte der Transport auf verschiedene Weisen - also synchron bzw. asynchron - implementiert werden. Um das notwendige Wissen zu bekommen, fanden neben dem Projekt Vorlesungen statt, die auf das Projekt vorbereiten sollten und zeitnah die bevorstehenden Hindernisse aufklären sollten.

## Projektidee

Nach dem ersten Termin entstanden zahlreiche Ideen. Die erste Idee war ein Notizzettel, der für mehrere Personen geteilt werden kann und es somit ermöglicht z.B. seine Einkaufsliste oder gemeinsame anstehende Termin für mehrere Personen zugänglich zu machen, oder eine Applikation zum verwalten von seinem geliehenen Geld. Es kann öfters mal vorkommen dass man Geld an seine Freunde verleiht und um da den überblick zu behalten wäre es interessant eine App zu haben. Da es den ersten Ideen aber am Funktionsumfang und an Innovation mangelte, entschlossen wir uns für die Idee einen Sportticker zu entwickeln, welcher es ermöglicht Nutzern einen eigenen Ticker anzulegen für ein Event seiner Wahl und somit seine Freunde immer auf den aktuellen Stand zu halten. Wenn jemand sich also zum Beispiel ein Handballspiel anguckt und möchte dass auch andere ihm beim zugucken folgen, kann er einfach dieses Ereignis anlegen und selber Kommentare zu dem Spiel veröffentlichen, damit jeder weiß was gerade auf dem Spielfeld passiert.

Als Grundfunktion wird das Erstellen von Events wichtig sein. Ein Nutzer kann für eine beliebige Sportart einen Liveticker ( Event ) erstellen. Er kommentiert dann dieses Event und aktualisiert den Spielstand und die Ereignisse. Nutzer können dem Event dann beitreten, seine Beiträge verfolgen und zu diesem Kommentare verfassen bzw. diskutieren. Der Ticker kann in Echtzeit bewertet werden, damit weitere Nutzer sehen können, ob der Ticker es wert ist, verfolgt zu werden.

Der Reiz an dem Erstellen eines Tickers ist, dass ein Nutzer seine eigenen Fans bekommen kann, die zusammen mit ihm Sportereignisse miterleben und Meinungen austauschen. Der Ersteller eines Tickers ist dabei der "Leiter". Er schreibt die wichtigen Ereignisse und seine Beiträge werden kommentiert bzw. von anderen Nutzern diskutiert.

Die App bietet eine übersichtliche Listung der Ereignisse und unterstützt die Kommunikation zwischen den Usern. Es ist zusätzlich möglich mehrere Ticker gleichzeitig in einem Fenster zu verfolgen. Wenn man z.B. Fan von zwei Teams ist, die an verschiedenen Spielen teilnehmen, will man trotzdem beide gleichzeitig erleben. Auch kann man so mehrere Ticker zum gleichen Spiel folgen und bekommt so mehr Infos und eine vielschichtige Sicht auf Geschehnisse.

# Projektspezifisches XML Schema

Um die Informationen zu verwalten benötigt man XML-Dateien. Extensible Markup Language kurz XML ist eine Auszeichnungssprache, die es ermöglicht Informationen bzw Text zu strukturieren. Eine XML Datei wird durch passendes XML Schema (XSD) beschrieben. In der XSD werden die Typen des XML bestimmt und durch passende Restriktionen begrenzt. Um also später Dateien zwischen Server und Client zu verschicken, bietet XML die richtige Basis.[[1]](#footnote-1)

Alle Daten sollten in drei XML-Dateien ausgelagert. Eine Datei enthält alle Daten der Nutzer. Hier werden die wichtigen Daten wie Username, Vorname, Nachname, Geburtsdatum, Anzahl erstellte Events und Anzahl geposteter Beiträge gespeichert. Die zweite Datei ist für die Events. Hier werden alle Grund-Informationen eines Events gespeichert. Eventname, Eventdatum, Eventadmin, Eventtyp, Eventdauer und Bewertung werden hier für alle Events abgelegt. Auch hier vergibt das System zusätzlich noch eine EventID zur eindeutigen Zuweisung. Die letzte Datei ist für den kompletten Content, also den Beiträgen und Kommentaren zuständig. Man hätte den Content und das Event zwar in ein XML-Dokument zusammenfassen können. Jedoch verändert sich der Content ständig und schnell, wobei die Event-Informationen statisch sind und nach anlegen nicht mehr verändert werden. Es macht somit Sinn den ständig verändernden Teil der Events auf eine extra Datei auszulagern. Dadurch, dass der Content ebenfalls die EventID bekommt, ist es kein Problem dem Event den zugehörigen Content zuzuordnen.

## Userprofiles:

Ein Nutzer des Dienstes muss sich durch ein Profil eindeutig identifizieren, falls er Events oder Kommentare veröffentlichen will. Der Nutzer identifiziert sich durch seinen Usernamen. Zusätzlich legt das System automatisch eine UserID für den einfacheren Umgang mit den Usern an. Die UserID wird später benötigt, um die Events und Beiträge eindeutig den Nutzern zuzuordnen.

Der User kann dann noch weitere Angaben machen, um von seinen Freunden erkannt zu werden. Vor-, Nachnamen und Geburtstag sind optionale Angaben, falls der User diese nicht veröffentlichen will. Ein User kann Freunde hinzufügen, um auf dem aktuellsten Stand zu bleiben, was seine Freunde so machen.Die Anzahl erstellter Events und geposteter Beiträge werden automatisch im Profil hinterlegt. Hier kann jeder User sehen, wie aktiv man selbst oder andere im Social Ticker sind.

Um alle Nutzer in einer Datei abzuspeichern werden alle User in einer „Userlist“ abgespeichert. Die Userlist besteht dann aus allen Usern, die am Social Ticker teilnehmen. Wichtig ist die Angabe, dass in dem Element mehrere Elemente auftauchen können hierfür ist die minOccurs und maxOccurs Angabe. Diese gibt an, wie oft ein Element mindestens auftauchen muss oder maximal auftauchen kann.[[2]](#footnote-2)

*<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">*

*<xs:element name="userlist">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element ref="user" maxOccurs="unbounded" minOccurs="0" />*

*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*

**simple-types:**

Simple-Types sind dafür da die einzelnen Elemente zu typisieren und zu benennen. Wichtig ist es Angaben zu machen von welchem Typ das Element sein soll. Typische Angaben sind „xs:string“, „xs:integer“, „xs:date“ oder „xs:time“. Aus den oben genannten Angaben entstehen folgende simple-types.

* Username (String)
* Vorname (String)
* Nachname (String)
* Geburtsdatum (Date)
* erstellte Events (Decimal)
* gepostete Beträge (Decimal)

Beispiel:

*<!-- Username -->*

*<xs:element name="username“ type="xs:string"/>*

**complexe-types:**

Complexe-Types sind dafür da mehrere Simple-Types in einem Element zusammenzufassen. Da User aus mehreren Informationen bestehen und man nicht alle Informationen als einen String abspeichert, werden diese auf mehrere Elemente aufgeteilt. Nach den oben genannten Vorgaben enstehen also folgende complexe-types.

* User (Username, Vorname, Nachname, Geburtsdatum, Geschlecht, erstellte Events, gepostete Beträge, Folger, Favoriten)\*
* Folger (Username)\*
* Favoriten (Username)\*

Beispiel:

*<!--Vollstaendige Userdaten -->*

*<xs:element name="user">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element ref="username"/>*

*<xs:element ref="vorname"/>*

*<xs:element ref="name"/>*

*<xs:element ref="gender"/>*

*<xs:element ref="geburtsdatum"/>*

*<xs:element ref="land"/>*

*<xs:element ref="stadt"/>*

*<xs:element ref="anzEvents"/>*

*<xs:element ref="beitraege"/>*

*<xs:element ref="favoriten"/>*

*<xs:element ref="folger"/>*

*</xs:sequence>*

*<!-- Bei Registrierung automatisch generierte einzigartige ID -->*

*<xs:attribute type="xs:positiveInteger" name="userID"/>*

*</xs:complexType>*

*</xs:element>*

Wie man an dem Beispiel sieht, ist die UserID als „attribute“ initialisiert, dies macht Sinn, da die ID generisch ist und sie nur zu Verwaltungszwecken genutzt wird.

**Restriktionen:**

Restriktionen werden benötigt, um die Elemente in einem passenden Rahmen angeben zu können. Es macht z.B. keinen Sinn einen Usernamen anzulegen, der 100 Buchstaben lang ist, deswegen werden die Elemente in ihrer Größe begrenzt. Restriktionen können auch dazu dienen, ein Element in der Auswahlmöglichkeit zu begrenzen. Das Geschlecht kann nämlich nur „männlich“ oder „weiblich“ sein als macht es hier Sinn, dies auch als Restriktion anzugeben

* Username min.3, max.15 Zeichen
* Vorname min.2, max. 20 Zeichen
* Nachname min.2, max. 20 Zeichen
* Geschlecht ist „Male“ oder „Female“
* Geburtsdatum zwischen zwischen 1900 und 2010
* Land ist unter „Germany“,“Italy“, „France“ und weiteren ausgewählten Ländern

Beispiel:

*<!-- Username -->*

*<xs:element name="username">*

*<xs:simpleType>*

*<xs:restriction base="xs:string">*

*<xs:minLength value="3"/>*

*<xs:maxLength value="15"/>*

*</xs:restriction>*

*</xs:simpleType>*

*</xs:element>*

<!-- Geschlecht -->

<xs:element name=*"gender"*>

<xs:simpleType>

<xs:restriction base=*"xs:string"*>

<xs:enumeration value=*"Male"*/>

<xs:enumeration value=*"Female"*/>

</xs:restriction>

</xs:simpleType>

</xs:element>

## Events

Die Events sind das Herzstück unseres Dienstes. Es können hier Events (wie zB ein Fußball- oder Handballspiel) erstellt und verwaltet werden. Ein Event benötigt natürlich eine eindeutige Kennung. Der Name wäre hierbei nicht in jedem Fall eindeutig. Deswegen wird eine ID zur einfacheren Verwaltung erstellt. Eventname gibt an um welches Thema der Ticker überhaupt geht. Der Eventadmin ist sofort der User, welcher das Event erstellt hat. Auch interessant für den User ist es, um welchen Typ es von Event handelt. Also ob es ein Football-, Faustball- oder doch Golfspiel ist. Ein Event kann von anderen Usern bewertet werden. Die Wertung wird abgespeichert und automatisch verrechnet.

**simple-types:**

* Eventname (String)
* Eventdatum (Date)
* Eventadmin (Username)
* Eventtyp (String)
* Eventstart (time)
* Eventende (time)
* Bewertung (positiveInteger)

**complexe-types:**

* Eventlist(Event\*)
* Event (EventID, Eventname, UserID, Eventtyp, Eventstart, Eventende, Eventdauer, Bewertung)

**Restriktionen:**

Damit der Eventname nicht zu lang wird, ist es sinnvoll ihn auf maximal 100 Zeichen zu begrenzen und, um Spam oder unsinnige Events zu verhindern, sollten diese mindestens 5 Zeichen lang sein. Zuerst war es so gedacht, dass man einen Eventtyp nur aus einer Liste von Events auswählen durfte. Doch es wurde nicht umgesetzt, da der Nutzer sonst zu beschränkt bei der Erstellung gewesen wäre, da es auch ausgefallene Sportarten gibt, die auch ihre Anhänger besitzen. Die Eventbewertung der Events sollte auf einer Skala von 1 bis 10 sein, dies bietet genügend Abstufungen, um die Qualität gut zu differenzieren.

* Eventname min. 5, max. 100 Zeichen
* Eventbeschreibung optional min. 3, max. 250 Zeichen
* Eventtyp min. 3, max. 100 Zeichen
* Eventbewertung optional <= 10 & >0

## Eventcontent

Um alle Beiträge(vom Admin) und Kommentare(von Usern) abzuspeichern, wird der Eventcontent benötigt .Zentral bei den Beiträgen des Events ist das "Soziale". Hierbei sind die Beiträge des Admins der Dreh- und Angelpunkt des Events. Andere User können dann zusätzlich noch unter den Beiträgen des Admins Kommentare setzen und diskutieren, ob sie die Geschehnisse genau so interpretieren oder einen anderen Standpunkt besitzen. Wichtig dabei ist auch, dass dem Admin eine Möglichkeit geboten wird, den aktuellen Spielstand zu veröffentlichen.

**simple-types:**

* TickerBeitrag (String)
* KommentarText (String)
* Heim (String)
* Gast (String)

**complexe-types:**

* EventContentList(Eventcontent\*)
* EventContent(aktuellerStand, eventID ,TickerBeitrag\*)
* TickerBeitrag (Text,Kommentar\*)
* Kommentar(Username,KommentarText)
* aktuellerStand(Heim,Gast)

**Restriktionen:**

Die Restriktionen beim Eventcontet sind nicht zu eng gewählt, da es jedem selbst überlassen wird, wie viel oder wie wenig man schreibt. Beiträge und Kommentare sollten aber schon eine mindestlänge von 3 Zeichen besitzen, damit es nicht zu unnötigen Spam kommt.

* TickerBeitrag min. 3 Zeichen
* KommentarText min. 3 Zeichen

# Kommunikationsabläufe und Interaktion

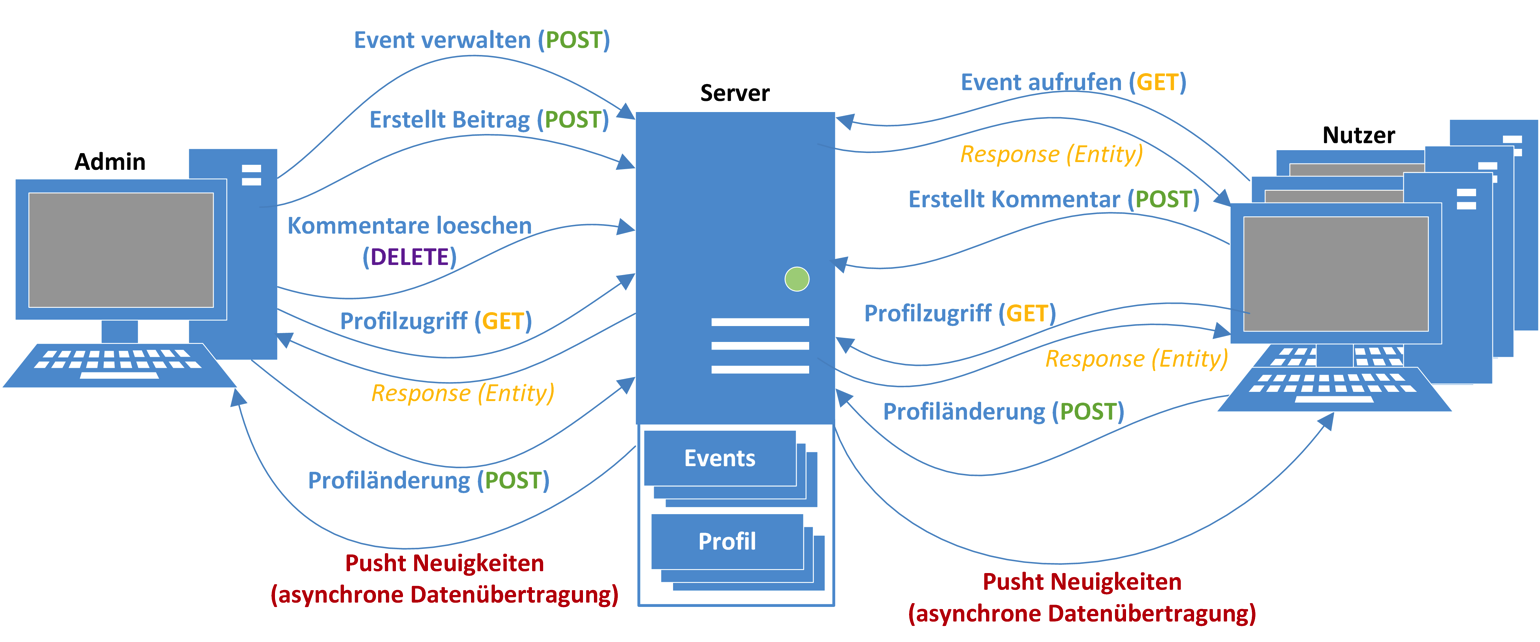
Im Grunde genommen ist erst einmal jeder Nutzer gleich und ihm stehen die gleichen Funktionen zur Verfügung. Jeder Nutzer kann auf sein eigenes Profil zugreifen und dieses auch verändern. Zusätzlich kann er auch andere Profile aufrufen und sich Freunde anschauen. Die Interaktion mit dem Server unterscheidet sich erst wenn man ein Event erstellt bzw. ein Event aufruft. Als Administrator (Ersteller eines Events) hat man einen erweiterten Funktionsumfang. Als allererstes sollte man für sein Event Informationen anlegen ("Event verwalten"). Wenn man alles soweit angelegt hat, kann man Beiträge verfassen und diese dann veröffentlichen. Sie werden chronologisch aufgelistet. Andere Nutzer, die das Event aufrufen können dann die erstellten Beiträge kommentieren. Als Admin ist es möglich Kommentare anderer Nutzer zu löschen. Als Nutzer (Aufrufer eines Events) kann man sich eine Liste aller Events anzeigen lassen. Dort wählt man dann ein Event auf, welches einen interessiert. In einem Event angekommen, sieht man alle aktuellen Beiträge und Kommentare. Man kann sich nun entscheiden, ob man selbst ein Kommentar verfassen will oder ob man erst einmal den Ticker verfolgt, bis man seine eigene Meinung veröffentlicht.

## Synchrone Datenübertragung

­Synchrone Datenübertragung funktioniert nach dem Request-Response-Prinzip. Der Client fragt Daten mit einem Request an und der Server antwortet mit einem Response. Diese Datenübertragung ist synchron, da nur Datenübertragen werden, die auch angefragt worden. Das Problem bei synchroner Übertragung ist, dass der Client nach jedem Request warten muss. Es wäre also besser wenn der Server einfach von sich aus Daten schickt, die der Client benötigt.

## Asynchrone Datenübertragung

­Asynchrone Datenübertragung ist das Gegenstück zur Synchronen Datenübertragung. Es funktioniert nach dem Publish-Subscribe-Prinzip. Der Client „subscribet“ Informationen, die er beabsichtigt automatisch zu bekommen und der Server veröffentlicht neue Informationen sobald sie geändert worden. Als einfaches Beispiel ist RSS zu sehen. Der Leser abonniert eine Quelle und bekommt automatisch neue Nachrichten zugestellt. [[3]](#footnote-3)



# Ressourcen und die Semantik der HTTP-Operationen

## Ressourcen:

Ressourcen sind ein sehr generisches und abstraktes Konzept. Eine Ressource ist identifizierbar und besitzt mehrere Repräsentationen. Sie stellt ein Objekt der Außenwelt dar und fasst dieses zum Beispiel in Textformat zusammen.

Aus unseren XML-Dateien kristallisieren sich somit drei wichtige Ressourcen heraus. Zum einen gibt es den User als Ressource, der aufgerufen und verändert werden kann. Dann gibt es die Ressource Event, die alle wichtigen Informationen eines Events enthält. Und zum Schluss gehört zu jedem Event der dazugehörige Content, welcher somit die dritte Ressource ist.[[4]](#footnote-4)

## HTTP-Operationen (Verben) [[5]](#footnote-5):

GETGET ist die grundlegendste und wichtigste Operation. Sie dient dazu Informationen welche durch eine URI (Universal Ressource Identifier) identifiziert werden abzuholen. Fast jeder Befehl im WWW ist ein GET, da so erst die Informationen ausgelesen werden können.

PUTMit PUT wird eine bestehende Ressource aktualisiert oder auch neu angelegt. PUT wirkt sich direkt auf die Ressource aus, aber muss nicht unbedingt sie komplett verändern. Es können auch nur Teile die entsprechenden Teile aktualisiert werden. PUT ist das genau Gegenstück zu GET .

POSTPOST ist ähnlich wie PUT. In erster Linie ist POST dazu eine neue Ressource unter einer vom Server bestimmten URI anzulegen. POST kann aber auch anderen Zwecken verwendet werden, bei denen die restlichen Operationen nicht passen. Der Unterschied zwischen PUT und POST ist, dass bei einem POST der Client nicht die URI der Ressource angibt.

DELETEDELETE ist zum Löschen einer Ressource zuständig. Es kommt nicht unbedingt zu einem wirklichen Löschen der Entität aber für den Client sind die Daten aber logisch gelöscht.

## User:

Die Funktionen eines Users sind die Standard Funktionen des „Social Tickers“. Ein Nutzer muss zum einen auf alle bereits bestehenden Nutzer zugreifen, sowie sich selbst bearbeiten können. Im Mittelpunkt der Applikation steht das Verwalten und Einsehen aller Events. Ein Nutzer kann sich alle bereits bestehenden Events anzeigen lassen, und auch bestimmten Events beitreten. Als Nutzer besitzt man in der Applikation dann die Fähigkeit, Beiträge vom Admin zu Kommentieren.

Um auf einzelne Ressourcen zuzugreifen, werden IDs benötigt, welche einen Nutzer eindeutig identifizieren. Diese werden als PathParams benutzt. Was PathParams sind, wird in einem späteren Abschnitt erklärt. Aus diesen Überlegungen folgen deswegen diese Operationen:

|  |  |
| --- | --- |
| **Operationen** | **Beschreibung** |
| **GET**/users | Gibt alle User zurück |
| **GET**/users/$userID | Gibt einen bestimmten User zurück |
| **PUT**/users/$userID | Verändert seine eigenen User-Informationen |
| **GET**/events | Gibt alle Events zurück |
| **GET**/events/$eventID | Gibt ein bestimmtes Event zurück |
| **GET**/events/$eventID/eventcontent | Gibt den gesamten Content eines Events zurück |
| **POST**/events/$eventID/eventcontent/$tBID | Erstellt einen Kommentar zu einem Beitrag |

## Admin (Ersteller eines Events):

Ein Admin hat andere Rechte bzw. andere Funktionen als ein normaler Nutzer, der auf einen Ticker zugreift. Er hat die volle Kontrolle, kann Spielstände aktualisieren, Beiträge löschen und überhaupt den Ticker erst mit wichtigen Informationen - den Beiträgen - füllen. Dem Admin stehen zusätzlich die Funktionen der User zur Verfügung.

|  |  |
| --- | --- |
| **Operationen** | **Beschreibung** |
| **POST**/events | Admin erstellt ein neues Event |
| **PUT**/events/$eventID | Aktualisieren eines bereits erstellten Events |
| **POST**/events/$eventID/eventcontent | Aktualisieren des Contents eines Events |
| **DELETE**/events/$eventID | Löschen der bestimmten Ressource |
| **DELETE**/events/$eventID/eventcontent | Löschen des zugehörigen Contents |

# RESTful Webservice

## Grundlagen zur Implementierung der Operationen

Um auf die XML-Daten zuzugreifen wird Marshalling und Unmarshalling verwendet. Damit man auf die Funktionen zugreifen kann, wird zuerst das JAXBContext benötigt. In der Instance wird die zugehörige Klasse mit Angegeben. Unmarshalling dient dazu aus einer angelegten XML-Daten auszulesen. Danach können Daten angehängt, verändert oder gelöscht werden und anschließend durchs Marshalling wieder abgespeichert werden.

*JAXBContext jc= JAXBContext.newInstance(Eventlist.****class****);*

*Unmarshaller um = jc.createUnmarshaller();*

*Eventlist events = (Eventlist) um.unmarshal(****new*** *File("XML/Eventlist.xml"));*

*Marshaller marshaller = jc.createMarshaller();*

*marshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, Boolean.TRUE);*

Die letzte Zeile ist optional, sie ist aber sinnvoll, wenn man nicht möchte, dass die Datei unformatiert abgespeichert wird. Lässt man die Zeile weg, wird beim marshallen die komplette Datei in einer Zeile wieder abgespeichert.

## Implementierung der GET-Operationen

Wie bereits erwähnt, dient GET dazu Ressourcen abzufragen. Um es in REST zu implementieren, muss man angeben, welchen Pfad die Entität hat, und was sie produziert. Damit der Server weiß, dass es sich um den MIME Typ XML handelt, ist er sinnvoll MediaType.APPLICATION\_XML zu verwenden.

*@GET*

*@Path( "/events" )*

*@Produces( MediaType.APPLICATION\_XML )*

Zurückgegeben werden muss dann nur noch eine Nachricht, dass alles ok ist.

***return*** *Response.status(200).entity(events).build() ;*

## Implementierung der POST-Operationen

Etwas schwieriger gestaltet sich die Implementierung der POST-Operation, da zum einen eine vom Server generierte ID erstellt werden muss und zum anderen eine neue Ressource angelegt werden muss. Eine weitere Änderung ist, dass beim POST der MIME Typ nicht mehr Produziert sondern Aufgenommen wird.

*@POST*

*@Path( "/events" )*

@Consumes *( MediaType.APPLICATION\_XML )*

Durch eine Schleife wird abgefragt, welche die bisher größte vorhandene ID ist und somit wird die neue ID um eins größer sein. An die Funktion selbst wird nur die zu erstellende Entität übergeben, der Server macht dann alles von selbst. Um die neue Entität zu erstellen, wird sie dann an alle bisherigen angehängt. Der Server muss zum Schluss nur noch die neue URI zurückgeben.

*List<Event> eventliste = events.getEvent();*

*BigInteger id = BigInteger.ZERO ; // IDs sind BigInteger*

***for****(Event ev : eventliste ){ // ev dient zum Durchlauf*

***if****(ev.getEventID().compareTo(id)==1){ // Ergebnis ist 1 wenn größer*

*id = ev.getEventID(); } }*

*event.setEventID(id.add(BigInteger.ONE));*

*eventliste.add(event);*

*marshaller.marshal(events,* ***new*** *File("XML/Eventlist.xml"));*

***return*** *URI.create*

*(„*[*http://localhost:4434/events/*](http://localhost:4434/events/)*„+event.getEventID().toString());*

## Implementierung der PUT-Operationen

Analog zur Implementierung der POST-Operation wird PUT ähnlich implementiert. Eine Änderung dabei ist, dass der Path mit einem PathParam erweitert werden muss. Was genau das ist, wird aber im Abschnitt „PathParams und QueryParams“ diskutiert. Die einzigen Änderung sind in der for-Schleife, da diese jetzt keine ID überprüft sondern die zugehörige Entität sucht, es muss also für die Suche die zur Entität gehörige ID übergeben werden und es wird keine neue ID mehr erstellt sondern nur noch die alte Entität mit der neuen ersetzt.

***int*** *i = 0;*

***for*** *(Event ev : eventliste) {*

***if*** *(ev.getEventID().equals(id)) {*

*eventliste.set(i, event);*

*}*

*i++;*

*}*

## Implementierung der DELETE-Operationen

Die Delete-Operation stellt keine großen Herausforderungen, da auch dort nur die for-Schleife angepasst werden muss und per remove wird die fällige Entität gelöscht.

**int** i = 0;

**for** (Event ev : eventliste) {

**if** (ev.getEventID().equals(id)) {

eventliste.remove(i);

}

i++;

}

Wodran aber unbedingt gedacht werden musste, ist, dass wenn man ein Event löscht, zusätzlich auch die Entität des Contents gelöscht werden musste.

## PathParams

Mit PathParams kann man auf einen bestimmten Teil der Daten zugreifen. So ist es möglich ein bestimmtes Event bzw einen bestimmten User aufzurufen. Die beste Weise dies umzusetzen ist es Die PathParams als ID zu implementieren und somit schnell auf einzelne Daten zuzugreifen.

*@Path( "events/{eventID}" )  
...  
getOneEvent(@PathParam("eventID")* ***int*** *i)*

Die übergebene EventID wird lokal als Variable i abgespeichert. So kann man in der Funktion auf den PathParam zugreifen.

## QueryParams

Mit QueryParams lassen sich Ergebnisse filtern. Sinnvolle Filterung wäre z.B. User nach Namen oder Land zu filtern. Genau so macht es Sinn die Events nach Namen des Events zu filtern. Als Beispiel wäre ein Nutzer der alle Spiele von Bayern haben möchte und als Filter den Namen Bayern eingibt.

*getAllEvents( @QueryParam("name") String name )*

*.....*

***if****(name!=****null****){*

***for*** *(Iterator<Event> iter = eventliste.iterator(); iter.hasNext(); ) {*

*Event ev = iter.next();*

***if****(!ev.getEventname().toLowerCase().contains(name.toLowerCase())){*

*iter.remove();*

*} } }*

Falls man in den Browser nun „/events?name=Bayern“ eintippt, bekommt man alle Events, die das Wort Bayern enthalten. Alle anderen werden vorrübergehend entfernt.

# Clientseitige Operationen

Dem Client muss die Möglichkeit geboten werden auf alle Daten des Servers zuzugreifen bzw. muss er auch neue URIs anlegen können. Er soll sich zusätzlich nicht um die Namensgebung kümmern, da dies alles das System übernehmen soll. Primär sollten deswegen GET und POST implementiert werden.

## Clientseitige GET-Abfrage

Damit die Abfrage klappt, muss das Programm auf die Ressource des Servers zugreifen. Dies wird erledigt mit Hilfe von WebResource. Diese greift dann auf die Ressource zu und gibt die richtigen Datei zurück. Die URI muss mit der des Servers übereinstimmen.

*String url = "http://localhost:4434/events";*

*WebResource wrs = Client.create().resource(url);*

*Eventlist ev = wrs.accept("application/xml").get(Eventlist.****class****);*

## Clientseitige POST-Methode

Der Aufbau von POST-Abfragen ist ähnlich wie der des GETs. Der Unterschied besteht darin, dass zu aller erst ein neues Event angelegt werden muss. Die geschieht durch die automatisch generierte ObjectFactory.

*Event event =* ***new*** *ObjectFactory().createEvent();*

Das angelegte Objekt sollte danach mit Informationen gefüllt werden. Auch dies geschieht mit den automatisch von JAXB generierten Methoden „set...“.

*event.setUsername("User");*

*event.setEventname("Eventname");*

Danach muss die Ressource abgefragt und zum Schluss neu abgespeichert werden. Dies geschieht durch die Methoden „.entity(entität)“ und „.post(klasse)“. Am Ende gibt es noch eine Ausgabe, ob die POST-Methode erfolgreich war. Falls ja, gibt diese den Wert „201“ zurück.

*String url = "http://localhost:4434/events";*

*WebResource wrs = Client.create().resource(url);//erstellt neues Event*

*//erstellt Event und gibt einen Response zurück. Server übernimmt die genaue Verwaltung*

*ClientResponse cr=wrs.accept("text/html").type (MediaType.APPLICATION\_XML).entity(event).post(ClientResponse.****class****);*

*//Rückgabe des Status*

*System.out.println(cr.getStatus());*

# Konzeption + XMPP Server einrichten

XMPP dient dazu Daten asynchron zu übertragen. Es funktioniert nach dem Publish-Subscribe-Prinzip. Im Mittelpunkt steht ein XMPP Server. Dieser sendet Nachrichten an alle Nutzer welche sich für die Nachrichten eingetragen hat - also welche er bekommen möchte. Ein Nutzer muss sich vorher entscheiden welche Informationen er bekommen will.

## Leafs (Topics)

Jedes Event ist abonnier bar. Sobald man dem Event beitritt, bekommt man als erstes alle Nachrichten als GET-Operation. Wenn es dann aufgerufen ist, werden neue Benachrichtigungen per asynchrone Benachrichtigungen verschickt. Ein User kann selbst neue Kommentare schreiben oder von anderen welche zugestellt bekommen. Also sind alle Nachrichten die sobald man sich in einem Event befindet asynchron.

## Publisher

Im Grunde benommen ist jeder Publisher, wer sich in einem Event befindet. Als User kann man Kommentare erstellen und diese werden asynchron an andere gepusht. Der Admnis selbst ist natürlich auch Publisher. Er erstellt Beiträge zu seinem Event welche alle zugestellt bekommen.

Damit ein Topic zu einem Event erst erstellt wird, muss sich ein User dafür entscheiden ein Event zu erstellen. Sobald er dies getan hat, wird er automatisch als Admin festgelegt und „publisht“ die angelegte Node.

## Subscriber

Genauso wie jeder Publisher sein kann, ist auch jeder Subscriber der sich in einem Event befindet. Sobald man einem Event beitritt kriegt man alle neuen Beiträge und Kommentare asynchron. Die neuen Events werden sofort auf der neuen Liste gezeigt also ist ab dem Login jeder User ein Subscriber.

Was zuerst noch zu Fehlern führte, war dass das „Unsubscribe“ nicht implementiert war. Es ist nämlich wichtig, da sonst immer mehr EventListener zur selbe Node hinzugefügt werden und ja öfter man beitritt, desto öfter bekommt man neue Nachrichten, da die Listener jeweils eine Rückgabe haben.

## Zu übertragene Daten

Die übertragbaren Daten sind die Beiträge eines Admins und die Kommentare der User. Im Grunde genommen ist der Ticker nicht viel anders als ein Chat mit besonderer Formatierung und Featuers wie Aktueller Stand und Bewertungen. Zusätzlich ist es sinnvoll alle aktuellen bzw neuen Events automatisch zu übertragen. Sobald also ein User seinen eigenen Ticker erstellt, kriegen alle anderen User sofort diesen zur Auswahl.

## XMPP Server

Die Einrichtung des XMPP Servers fand mit Hilfe des Beispielvideos statt.[[6]](#footnote-6)

# XMPP - Client

## Erstellen von Nodes

Um eine Node zu erstellen wird wie bei allen anderen Funktionen auch der PubSubController benötigt, dieser ermöglicht erst die Operationen mit einer Node. Die ConfigureForm ist dafür da, bestimmte Einstellungen für die zu erstellende Node vorzunehmen. Danach wird nur noch eine neue Node mit dem angegebenen Namen und Einstellungen erstellt. [[7]](#footnote-7)

*PubSubManager mgr =* ***new*** *PubSubManager(verbindung.conn);*

*ConfigureForm form =* ***new*** *ConfigureForm(FormType.submit);*

*// Einstellungen für die Node*

*form.setAccessModel(AccessModel.open);*

*form.setDeliverPayloads(****true****);*

*form.setNotifyRetract(****true****);*

*form.setPersistentItems(****true****);*

*form.setPublishModel(PublishModel.open);*

*LeafNode leaf = (LeafNode) mgr.createNode(nodeName, form);*

## Abonnieren von Nodes

Das Abonnieren von Node funktioniert wie folgt. Es wird eine Instanz von PubSubManager zum verwalten benötigt. Dieser kann jedoch nur mit einer gültigen Verbindung erstellt werden.

Danach wird die Node gesucht. Zu der Node wird ein EventListener hinzugefügt, der dafür sorgt, dass alle Nachrichten richtig eingehen. Als letztes wir nur noch die Node zu einem bestehenden Account abonniert.1

*PubSubManager mgr =* ***new*** *PubSubManager(verbindung.conn);*

*LeafNode node =* ***null****;*

*node = mgr.getNode(nodeName);*

*node.addItemEventListener(****new*** *ItemEventCoordinator());*

*node.subscribe(myJid);*

## Nachrichten veröffentlichen & Übertragung von Nutzdaten (Payload)

Damit neue Nachrichten auch veröffentlicht werden, dient die publish()-Methode hier wird PayLoad übergeben damit dieser mit übertragen wird. 1

*node.publish(* ***new*** *PayloadItem<SimplePayload>(****null****,****new***

*SimplePayload("", "" ,xml)));*

## Service Discovery

Service Discovery dient dazu um alle Nodes samt ihrer Informationen ausgeben zu können. Die Methode gibt dann einfach alle Nodes zurück. Das Programm welches diese Methode aufgerufen hat, kann dann entscheiden, wie die Daten weiter genutzt werden können. 1

*List<String> nodes=****new*** *ArrayList<String>();*

*DiscoverItems item=****this****.mgr.discoverNodes(****null****);*

*Iterator<DiscoverItems.Item> items = item.getItems();*

***while****( items.hasNext()){*

*nodes.add(items.next().getNode());*

*}*

## Nachrichten empfangen

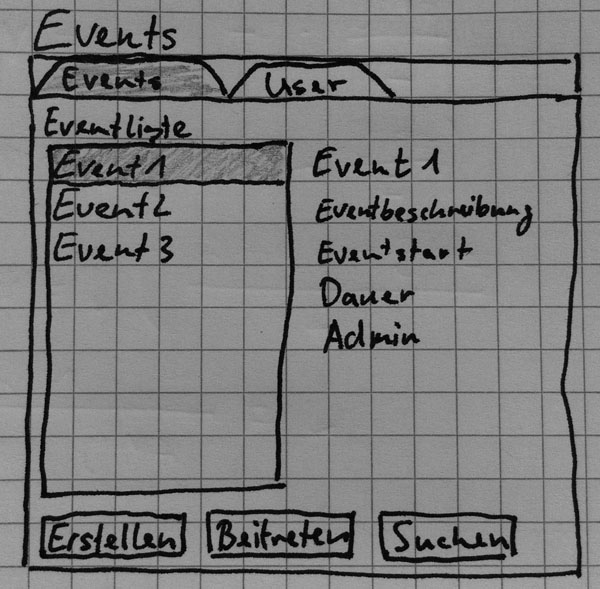
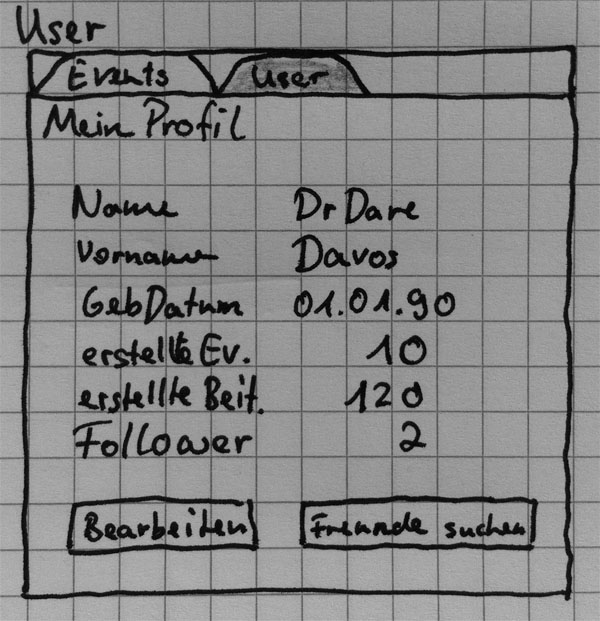
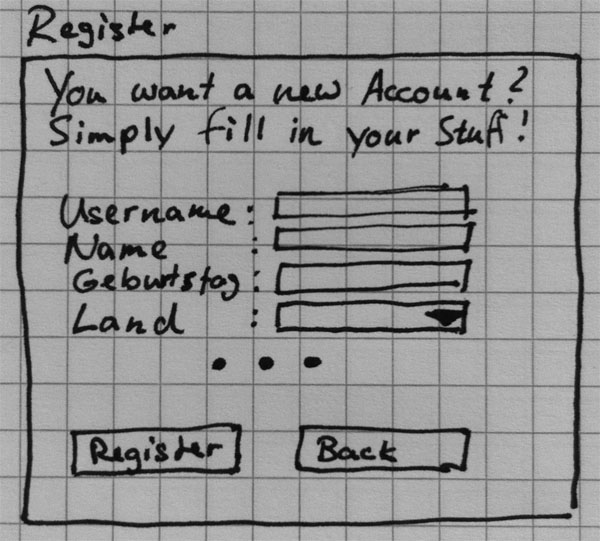
Damit der Social Ticker dann die neuen Nachrichten bekommt, ist es wichtig in der Service Discovery den Update für alle Kommentare auszulösen, damit sich im geöffneten Fenster automatisch alles aktualisiert. Hierfür greift man auf die Instance der Social Main zu und benutzt die definierte Methode.

*WindowSocialMain.getInstance().update();*

# Client – Entwicklung

## Entscheidung für JavaFX

## Graphical User Interface (GUI)



## 

# Abschluss

## Fehlende Features

Auf manche Features wurde noch verzichtet, da erst einmal alle wichtigen Funktionen implementiert werden sollten um eine Kommunikation zwischen den Clients zu gewährleisten. Manche Funktionen wurden in Ansätzen auch schon implementiert, aber noch nicht in die GUI eingebaut. Man hätte folgende Funktionen noch ergänzen können:

* Bewertung für Events
  + Man hätte dafür einfach zwei weitere Buttons benötigt, die es einem ermöglichen ein Event zu empfehlen oder negativ zu bewerten. Es wäre dann als synchrone Dateiübertragung an den Server geleitet worden, der dafür gesorgt hätte, die neue Bewertung abzuspeichern
* Suche für User
  + Im User Fenster wäre hierfür ein Textfeld gewesen welches einem die Möglichkeit bietet, angemeldete User zu suchen. Die Suche selbst wäre mithilfe von QueryParameter umgesetzt worden und als synchrone GET-Abfrage an den Server gesendet worden. Die Ausgabe, sähe dann ähnlich aus wie bei den aufgelisteten Events. Nach einem Klick auf einen Usernamen, wären dann auch weitere Informationen wie Geburtsdatum und Heimatland erschienen
* Suche für Events
  + Falls sehr viele Events erstellt worden wären, hätte man diese ebenfalls durch QueryParameter filtern können. Und als kürzere Liste ausgeben können.
* Löschen eines Events
  + Noch ist es nicht möglich Events zu löschen. Aber auch hier wäre es eine einfache DELETE-Operation gewesen. Im Server ist diese auch schon implementiert. Nur die GUI bietet dieses Feature noch nicht.
* Follower und Freunde
  + In der XML gibt es Platz um Freunde zu speichern, jedoch bietet die GUI auch hierfür noch keine Möglichkeit Freunde hinzuzufügen, damit dies umgesetzt werden könnte, müsste auch zuerst die User-Suche umgesetzt sein, deswegen ist es logisch.

# Quellen

Buch: REST und HTTP, Stefan Tilkov, dpunkt.verlag 2011

<http://www.w3schools.com/schema/schema_complex_indicators.asp>

<http://xmpp.org/extensions/xep-0248.html>

<http://www.youtube.com/watch?v=xPV0UodnTrA>

<http://www.igniterealtime.org/builds/smack/docs/latest/documentation/extensions/pubsub.html>

<http://www.igniterealtime.org/builds/smack/docs/latest/documentation/extensions/disco.html>

<http://www.torsten-horn.de/techdocs/jee-rest.htm#JaxRsHelloWorld-Grizzly>

1. Buch: REST und HTTP, Stefan Tilkov, 2011 s.84ff. [↑](#footnote-ref-1)
2. http://www.w3schools.com/schema/schema\_complex\_indicators.asp [↑](#footnote-ref-2)
3. Aus Vorlesung und Skript von Prof. Dr. Kristian Fischer [↑](#footnote-ref-3)
4. Buch: REST und HTTP, Stefan Tilkov, 2011 s.33ff. [↑](#footnote-ref-4)
5. Buch: REST und HTTP, Stefan Tilkov, 2011 s.33ff. [↑](#footnote-ref-5)
6. <http://www.youtube.com/watch?v=xPV0UodnTrA> [↑](#footnote-ref-6)
7. http://www.igniterealtime.org/builds/smack/docs/latest/documentation/extensions/pubsub.html [↑](#footnote-ref-7)