

Dokumentation

GeoCatching

Malte Odenthal
Pascal Kottmann
Sommersemester 2013

Inhalt

GeoCatching - Das Projekt.....	4
Die Idee.....	4
Konzept.....	4
Kommunikation	5
Synchron.....	5
Asynchron.....	5
Ziel	5
Meilenstein 1: XML-Schemata.....	6
XML.....	6
XML-Schema.....	6
CacheList – XML.....	6
CacheList – Schema	7
User	7
Meilenstein 2.....	8
RESTful.....	8
Cachelist REST.....	8
Userlist REST	9
Meilenstein 3: RESTFul-Webservice mit Grizzly und JAXB	10
Projektaufbau:.....	10
Beschreibung des Aubaus:	11
Projektvorgehen	13
Meilenstein 4 XMPP	14
Meilenstein 5 + 6 XMPP-Cliententwicklung und GUI	16
Funktionen:	16
Startbildschirm	17
User Erstellen	18
Abonnement anlegen.....	19

Nach dem Login	20
Caches anzeigen	20
Cache anlegen	22
Fazit	24
Probleme	24
Mögliche Erweiterungen	24

GeoCatching - Das Projekt

Die Idee

Entwicklung einer App für Geocaches, welche von einem zentralen Server abgerufen werden. Die Caches liegen in Form von Geodaten vor und sollen später auf einer Karte angezeigt werden. Der Nutzer dieser App soll die Möglichkeit haben, die Caches über bestimmte Kriterien (Ort, Typ, etc.) zu filtern. Die Geocaches sollen in einer XML-Datei gespeichert werden. Bei Abruf dieser Dateien, wird vor der eigentlichen Datenübertragung gefiltert, sodass der Client nur die Informationen erhält, welche er selber benötigt. Dies spart Traffic auf beiden Seiten.

Konzept

Bevor eine XML-Liste von Geocaches vom Server bezogen wird, wird diese anhand bestimmter Kriterien gefiltert, sodass wirklich nur die Caches heruntergeladen werden, welche auch wirklich vom User gewollt sind um die Daten die transportiert werden müssen möglichst gering zu halten.

Filterkriterien sind:

- Umkreissuche
- Cachetyp (Traditional, Multi, Mikro)
- Schwierigkeitsgrad
- Terrain
- Dauer
- Travelbug

Erhält nun der Benutzer eine Liste von Geocaches, soll er auswählen können ob er zu einem Cache mehr Informationen haben möchte und ihn auf der Karte anzeigen lassen. Als Karte haben wir uns für GoogleMaps entschieden, da dies ein einfacher Weg ist um auf dem Android-Phone eine funktionierende Karte mit Navigation zu erhalten.

Geocaches sollen über eine spezielle Eingabemaske in der App erstellt werden können. Hier ist noch zu entscheiden wie es mit Caches aus mehreren Wegpunkten funktionieren könnte. Ein oder mehrere erstellte Caches sollen dann nach dem Abschluss des Erstellvorgangs als gesammelte Liste an den Server geschickt werden. Auch hier bietet sich das Format XML an. Möglich ist aber auch der Gedanke, dass ein einzelner Cache direkt an einen RESTful Webservice übergeben wird.

Neben diesen beiden großen Punkten haben wir uns überlegt, dass wir eine Straßennavigation bis hin zu einem Parkplatz, welcher günstig für den Cache ist, anzubieten. Dies wäre aber vorerst Zusatz und ist vom Grundkonzept her nicht vorgesehen.

Kommunikation

Synchron

- Geocaches als XML herunterladen, die nach bestimmten Kriterien gefiltert ausgegeben werden.
- Die Nutzer sollen Caches erstellen und hochladen können und Kommentare zu bestehenden Objekten veröffentlichen können

Asynchron

- Wenn ein neuer Cache auf den Server hochgeladen wurde wird diese Information entsprechend der Abonnements an die Abonnennten weitergeleitet (Kriterien sind Beispielsweise: Typ oder Ort des Caches).

Ziel

Ziel unseres Projekts ist ein Programm zu erstellen, welches alle geforderten Kommunikationsverfahren abdeckt.

Meilenstein 1: XML-Schemata

Als erstes war es erforderlich, unsere Projektvorstellungen in XML und XML-Schema umzusetzen. Letztendlich sind für unser Projekt zwei XML und zwei dazu gehörige Schemata entstanden, welche wir im Folgenden anhand ihrer Funktionen auflisten werden.

XML

- CacheList.xml
- User.xml

XML-Schema

- CacheList.xsd
- User.xsd

CacheList – XML

Kommen wir nun zu dem Kernteil unseres Projekts. Den Caches. Hierfür haben wir eine validierbare Cachelist erstellt, welche alle für den Benutzer gewünschten Geocaches enthält. Die Liste enthält einzelne Cache-Elemente zu diesem Cache. Diese Cachelist soll bereits gefiltert über das Internet an den Client gesendet werden. Diese kompakte Informationsstruktur bietet den Vorteil der einfachen Handhabung. Alle relevanten Informationen sind in einer einzigen Datei gespeichert und können so komplett übertragen werden. Die Endgeräte erhalten jeweils nur die Liste, welche sie auch benötigen. Leider hat die kompakte Struktur den Nachteil, dass das Dokument sehr verschachtelt ist. Wichtige Elemente, die vom Aufbau her immer gleich sind, haben wir als Attribute definiert.

Hierzu fällt z.B:

```
<location lat="50.98817" lon="7.83573"></location>
```

Hier sind die Attribute lat und lon definiert worden, welche die Koordinaten eines Punktes auf der Karte angeben. Oder auch:

```
<Owner b_id="b0001">Malte Odenthal</Owner>
```

Diese Vorgehensweise erstreckt sich über alle XML-Dokumente, da es uns das Erstellen der Schemata vereinfacht hat, denn alle Attribute besitzen *restrictions*.

Was wir leider nicht implementiert haben ist das Einbinden von Bildern, die die Umgebung des Caches Zeigen sollten, Kommentaren und Logs zu den einzelnen Caches, dies war zeitlich und aufwandsbedingt leider nicht möglich.

CacheList – Schema

Das Schema der CacheList wurde von uns so gestaltet, dass alle komplexen Elemente als einzelne *complexType* umgesetzt wurde, welche *named* und nicht *anonym* sind, um später eine einfachere Handhabung mit den generierten Klassen zu haben. Aus dem erstellten Schema lässt sich die geplante CacheList gut generieren und das Schema kann so verwendet werden. Bei dem generieren einer XML-Datei aus dem Schema trat ein Problem auf. Die Generierung bestimmter Attribute, welche mit *pattern* und als *base token* definiert sind, erscheint als generierter Inhalt nur „token“ und nicht das gewünschte Ergebnis. Umgehen konnten wir dieses Problem bisher noch nicht. Hier der Code:

```
<xsd:simpleType name="cldRestriction">
  <xsd:restriction base="xsd:token">
    <xsd:pattern value="[c][0-9]{4}"/> <!-- nur c+4stellige zahl -->
  </xsd:restriction>
</xsd:simpleType>
```

Das Validieren der XML-Datei gegen das Schema funktioniert einwandfrei. Viele *restrictions* sind durch die großen REGEX-Ausdrücke sehr komplex.

User

Die XML-Datei User repräsentiert die Daten eines frisch registrierten User, welche vom Endgerät an den Server verschickt werden, damit dieser die Daten für lange Zeit speichern kann. Hier haben wir uns überlegt, dass es bei der Registrierung durchaus optionale Elemente geben kann, welche wir nachträglich als *optionals* hinzugefügt haben.

```
<optional>
  <news wanted="yes"/>
  <benachrichtigung wanted="yes"/>
  <ortsListe>
    <ort postal="51645" umkreis="10km" lat="33.12"
      lon="123.12">Gummersbach-Vollmerhausen</ort>
    <ort postal="33501" umkreis="30km" lat="73.12"
      lon="193.62">Teutoburger Wald</ort>
  </ortsListe>
</optional>
```

News und Benachrichtigung sind optional und sind Fragmente der asynchronen Kommunikation. Besonders interessant ist hier die Ortsliste, welche die Orte enthält wo der Benutzer informiert werden soll, wenn dort ein neuer Cache erstellt wurde.

Das Schema-Dokument war sehr einfach, da sehr viele *complexType* und *restrictions* sich in allen Dokumenten sehr ähneln. Sinnvoll wäre es hier z.B. eine Datei mit den mehrfach vorkommenden Elementen zu haben, welche dann über *namespaces* eingebunden wird.

Meilenstein 2 : Ressourcen und Semantik der HTTP-Operationen

RESTful

Beim "Representational state transfer"(REST) handelt es sich um einen Architekturstil, bei dem eine Ressource über eine URI Identifiziert werden kann.

Die Rest Anfrage wird über eine der HTTP-Methoden ausgeführt:

GET ~> Read

POST ~> Create

PUT ~> Update

DELETE -> Delete

<http://www.w3.org/Protocols/HTTP/Methods.html>

Eine Rest Anfrage an einen Server führt eine der oben genannten Methoden aus und ermöglicht die Kommunikation zwischen 2 Geräten (Client - Server).

Die URL sollte "umgangssprachlich" nicht mehr als 255 Zeichen besitzen um die Kompatibilität mit älteren Browsern zu gewährleisten

Cachelist REST

Ressource	URI	Methode
Komplette Cacheliste abrufen	/cachelist	GET
Cacheliste via Filter filtern	/cachelist/new/filter/{filter}	GET
Cacheliste abrufen	/new	GET
Cachelist erstellen	/new	POST
Cache löschen	/delete/{id}	DELETE
Originalcache löschen	/id}:{name}	DELETE

Cachelist abrufen

Mithilfe der Methode GET auf die URI /cachelist wird die gesamte Cachelist vom Server abgerufen. Cachelist liefert die Originalliste, welche als „Backup“ dient. Deswegen werden alle weiteren Operationen über den Pfad /cachelist/new abgewickelt.

Cachelist filtern

Mithilfe der Methode GET auf die URI /Cachelist/new/filter/{filter} werden nur jene Caches zurückgegeben, welche den vorher definierten und übergebenen Filterkriterien entsprechen. Hierfür wird eine eigens dafür geschriebene Filterklasse „FilterCaches.java“ aufgerufen, welche den Filterstring verarbeitet und die Caches filtert. Dies ist ein aufwändiger Weg die Caches zu filtern, ermöglicht es uns aber eine beliebige Anzahl an Filtern anzuwenden ohne für jedes Kriterium eine

eigene Methode zu schreiben. Ein möglicher Filterpfad wäre z.B:
`http://localhost/cachelist/new/filter/CACHETYPE$TERRAIN!MULTI_CACHE$3.5`

Hierbei entspricht der vordere Teil des Filterstrings den Filterkriterien, also: CACHETYPE und TERRAIN. Der hintere Teil die Werte nach denen gefiltert werden soll.

Leider ist es uns nicht gelungen, Caches anhand einer Position und einem Umkreis richtig über den Filter zu filtern. Die Logik funktioniert einwandfrei und kann getestet werden. Dafür wurde eine eigene Klasse CalculatorLatLon.java geschrieben, welche die Entfernung zwischen zwei Koordinaten korrekt berechnet. Leider schlägt der Filter für die Distanzberechnung bei uns nicht an. Grund: Wissen wir nicht.

Cachelist abrufen

Mithilfe der Methode GET auf die URI `/cachelist/new/` wird die gesamte Cachelist vom Server abgerufen

Cache erstellen

Mithilfe der Methode POST kann man über die URI `/cachelist/new/` einen neuen Cache erstellt. Hier erfordert der Service die Übergabe eines einzelnen Caches, welche anschließend an die gesamte Liste angehängt wird.

Cache löschen

Über `/new/delete/{id}` lässt sich mit der Methode DELETE ein bestimmter Datensatz unter einer bestimmten ID löschen.

Userlist REST

Ressource	URI	Methode
Userliste abrufen	<code>/user</code>	GET
Neue Userliste	<code>/user/new</code>	GET
User anhand name und passwort filtern	<code>/user/{password}</code>	GET
User erstellen	<code>/user/new</code>	POST

Userlist abrufen

Mithilfe der Methode GET auf die URI `/userlist/` wird die gesamte Userlist vom Server abgerufen

Neue Userliste

Mithilfe der Methode GET auf die URI `/user/new` erhält man eine neue Userliste, welche dann bearbeitet werden kann

User Filtern (Accountcheck)

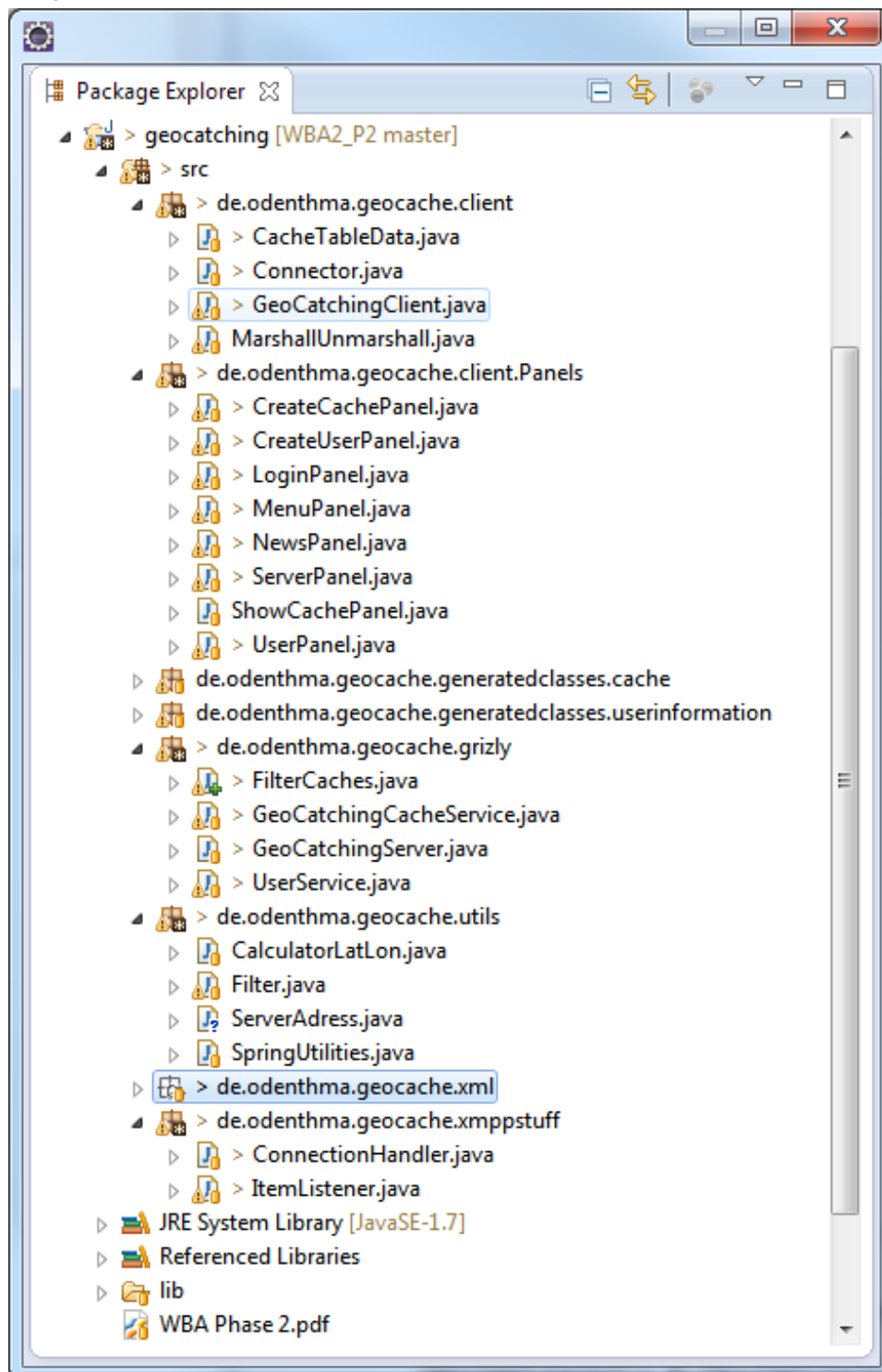
Mithilfe der Methode GET auf die URI `/user/{password}` erhält man einen User, welcher den Kriterien entspricht. Dies dient um zu prüfen ob der Account registriert wurde und ob somit der Login gestattet wird.

User erstellen

Mithilfe der Methode POST kann man über die URI `/user/new` einen neuen User anlegen.

Meilenstein 3: RESTful-Webservice mit Grizzly und JAXB

Projektaufbau:



Beschreibung des Aufbaus:

de.odenthma.geocache.client

CacheTableData.java

Ist eine Hilfsklasse, welche es uns ermöglicht eine Tabelle mit spezifischen Informationen der Caches zu füllen. Mit dieser Klasse kann zwischen Cachevorschau und voller Cachedarstellung unterschieden werden

Connector.java

Bearbeitet die Anfragen des Clients an den Server und verschickt diese dann. Antworten des Servers werden ebenso verarbeitet und dem Client mit dem richtigen Datentyp übergeben.

GeoCatchingClient.java

Dies ist der Einstiegspunkt des Clients mit GUI. Diese Klasse muss auf der Clientseite gestartet werden und stellt das gesamte Interface und die Kommunikation der erforderlichen Klassen bereit. Das gesamte Userinterface wurde mittels Cardlayout realisiert. Dazu mehr im dazugehörigen Meilenstein.

MarshallUnmarshall.java

Formt Datenströme richtig für die aufrufende Anwendung um.

de.odenthma.geocache.client.panels

CreateCachePanel.java

Stellt die Oberfläche für das Erstellen eines Caches bereit.

CreateUserPanel.java

Stellt die Oberfläche zum Erstellen eines Users bereit.

LoginPanel.java

Stellt die Oberfläche für Login auf einem Server mit Username und Passwort (bisher noch unverschlüsselt) bereit.

MenuPanel.java

Stellt die Oberfläche für das Menü nach dem Einloggen bereit.

NewsPanel.java

Stellt die Oberfläche für das Anzeigen von News, welche über den XMPP-Server geschickt wurden, bereit.

ServerPanel.java

Stellt die Oberfläche für den Connect zum Server bereit.

ShowCachePanel.java

Hier werden alle Caches angezeigt. Caches lassen sich in einer Detailtabelle anzeigen und können gefiltert werden.

UserPanel.java

Beinhaltet alle Cards, welche vor dem Login sichtbar sind.

de.odenthma.geocache.generatedclasses.cache

Hier sind alle aus einem XSD-Schema generierten Klassen zu finden, welche zu den Caches gehören.

de.odenthma.geocache.generatedclasses.unserinformation

Hier sind alle aus einem XSD-Schema generierten Klassen zu finden, welche zu den Usern gehören.

de.odenthma.geocache.grizly

FilterCaches.java

Stellt Routinen zum Auseinandernehmen des Filters und anschließendem Filtern der Caches bereit.

GeoCachtchingCacheService.java

Stellt Recourcen für Anfragen auf Caches an den RESTful-Webserver bereit.

GeoCatchingServer.java

RESTful-Webserver, welcher gestartet werden muss.

UserService.java

Stellt Recourcen für Anfragen auf Benutzer an den RESTful-Webserver bereit.

de.odenthma.geocache.utils

CalculatorLatLon.java

Stellt Routinen zur Berechnung der Entfernung zwischen zwei Koordinaten im dezimalen Latitude-Longitude-Format bereit.

Filter.java

Beinhaltet Konstanten, welche als Filterkriterium verwendet werden können.

ServerAdress.java

Hier kann die Serveradresse geändert werden.

SprintUtilities.java

Nicht verwendet. Eventuell später.

de.odenthma.geocache.xml

Enthält alle XML, bzw. XSD-Files, die für das Projekt verwendet werden. Alle Dateien, die mit New enden, werden letztendlich von den Services benutzt.

de.odenthma.geocache.xmppstuff

ConnectionHandler.java

Verwaltet alle Anfragen an den XMPP-Server. Hier können User registriert werden. Es wird sich hier drüber eingeloggt und Nodes erstellt. Enthält noch weitere Funktionen, welche später noch erläutert werden.

ItemListener.java

Listener, welcher an Nodes gehängt werden muss. Dieser wird dann gefeuert, sobald eine Nachricht an einen Knoten gehängt wird.

Projektvorgehen

Aufgrund der Tatsache, dass wir alle KomplexTypes in unserem CacheList.xsd-Schema nicht anonym deklariert haben, haben wir nach dem generieren der Klassen eine Menge verschiedener Klassen vorgefunden, was das Arbeiten mit diesen deutlich erschwert hat. Ein großes Problem trat bei Elementen auf, welche sich mit einer unbestimmten Menge wiederholen. Hierfür musste beachtet werden, dass unter dem eigentlichen Rootelement in den Klassen eine Liste generiert wurde.

Die XML-Listen, welche für die Verarbeitung genutzt werden, sollen vorher über den Path kopiert werden. Als Beispiel führt die URI: localhost:4434/cachelist eine Kopieroperation des entsprechenden Service für das Dokument aus. Mit dieser Kopie können nun alle weiteren Operationen ausgeführt werden, wie z.B. POST, GET und Delete. Beispielsweise: localhost:4434/cachelist/new zum Anlegen eines neuen Caches.

Das Erstellen einer lokalen Kopie ermöglicht es auf die ursprünglichen Listen zuzugreifen, falls man einen Fehler bei den Operationen gemacht hat.

Die gleiche Vorgehensweise ist auch bei den User-Klassen zu finden, d.h. Kopie erstellen und mit der Kopie weiterarbeiten.

Meilenstein 4 XMPP

Um die Asynchrone Synchronisation nutzen zu können wird auf XMPP zurückgegriffen, zur Funktionsprüfung wurde ein Openfire-Server installiert und konfiguriert, als XMPP-library wird die Smack API genutzt.

Prinzip und Aufbau der Knoten

Erstellt der User Orte, für welche er Benachrichtigungen bekommen möchte wenn ein Cache in diesem Bereich erstellt wurde, wird über den Publisher-User „publisher“ automatisch von dem CacheService ein Knoten für diesen Ort angelegt. Der Knotenname, bzw. die ID hat den diesen Aufbau: TYP:LAT:LON:UMKREIS. Beispielsweise könnte ein realer Knoten so lauten: CACHE:21.233:43.22134:20. Der Umkreis ist immer in Km anzugeben. Wir haben uns für diese Art von Knotenname entschieden, da sie einen entscheiden Vorteil bietet. Alle Informationen die einen abonnierten Ort und seinen Umkreis betreffen, finden sich im Namen des Caches wieder. So benötigt der CacheService keine weiteren Informationen um dem Ort Caches zuordnen zu können.

Funktionen der User

Einen eigenen Publisher-User zu haben, bietet den Vorteil, dass ein zentraler XMPP-User alle Knoten verwaltet und auch Messages an diese hängen kann. So ist es möglich, an alle Knoten, welche das Kriterium „innerhalb der Reichweite“ erfüllen direkt die erstellten Caches zu hängen. Die einzelnen User Abonnieren diesen Knoten lediglich und leiten die Knotenerstellung ein.

Registrierung eines Users und Login

Registriert sich ein neuer User bei dem RESTful-Webservice, so wird automatisch auch eine Registrierung bei dem XMPP-Server mit den gleichen Benutzerdaten vorgenommen. Das gleiche gilt für den Login. Möchte ein User sich auf dem System anmelden, so wird er direkt auf dem XMPP-Server angemeldet.

Probleme und Stand der Dinge

Wie oben beschrieben, erstellt der Service einen Knoten für einen Ort welchen der User anschließend abonniert. Wird nun von irgendeinem User ein Cache angelegt, so wird vom Service überprüft ob dieser den Kriterien der Knoten entspricht. Ist die Überprüfung positiv, so soll eine Payloadmessage an diesen Knoten geheftet werden. Dieser Vorgang funktioniert leider nicht. Die Gründe dafür konnten wir leider nur erahnen aber nicht eliminieren. Anscheinend verliert der Client in diesem Moment die Connection zum Server und der Vorgang schlägt fehl. Dies würde die NullPointerException erklären, welche jedes Mal geworfen wird. Da wir sehr lange versucht haben, diesen Fehler zu beseitigen und es nicht geschafft haben, sind wir zu dem Entschluss gekommen es für das Erste zu lassen. Um den Fehler präzise lokalisieren zu können, müssten wir die gesamte Klassenstruktur und die Kommunikation zwischen den Klassen überarbeiten. Sollte uns weitere Zeit zur Verfügung stehen, wäre dies der Erste Punkt an dem wir arbeiten würden.

Aufgrund dieser Probleme, konnten wir keine geeignete Struktur zum Anzeigen von Messages eines Knoten entwerfen.

Die zuvor in einem Testprojekt entwickelten Routinen funktionierten dort einwandfrei, von daher gehen wir davon aus, dass eben die Struktur und der Aufbau unseres eigentlichen Projekts falsch umgesetzt wurden.

Meilenstein 5 + 6 XMPP-Cliententwicklung und GUI

Aufgrund der vielen aufgetauchten Probleme mit Android und einer guten Kommunikation zwischen Endgerät und Server, haben wir uns gegen eine Android-App entschieden und stattdessen einen Client mit Java-Swing entwickelt. Dieser Client dient aber als Prototyp und könnte später als App umgesetzt werden. Mit Swing können wir eine einfache Client-Server-Kommunikation gewährleisten ohne Verluste im Umfang. Die Oberfläche des Clients ist sehr einfach gehalten und soll nur die Funktionalität bieten, welche für die Kommunikation geplant wurde.

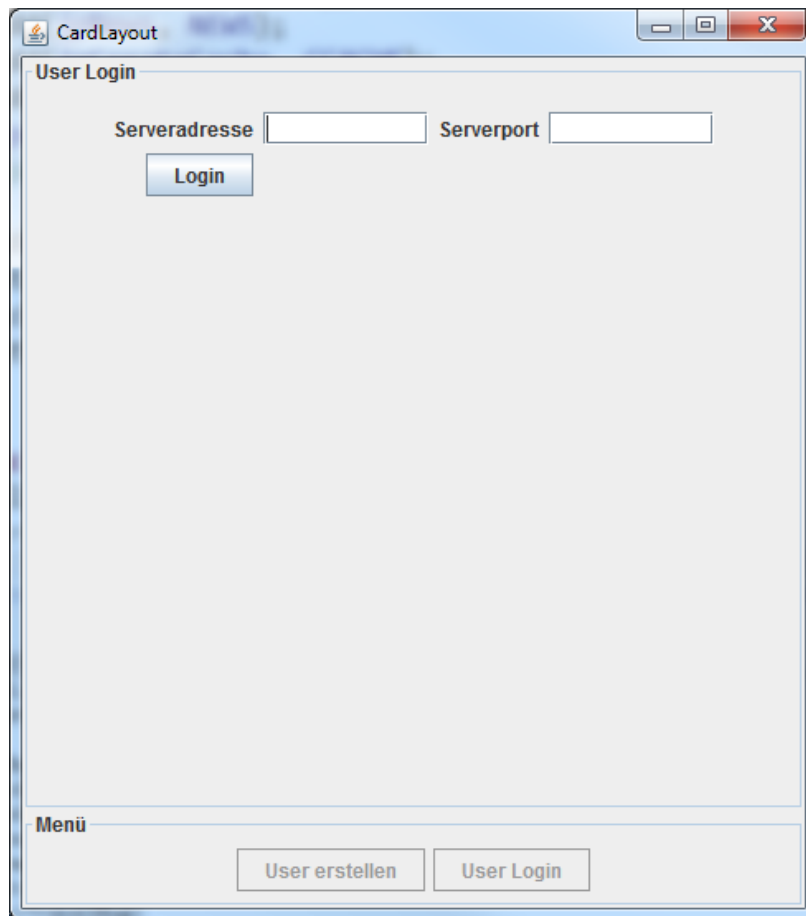
Funktionen:

- User anlegen und bei den Servern registrieren
- Userlogin auf REST und XMPP-Server
- Caches filtern und anzeigen
- Cache löschen
- Cache hinzufügen
- Ort Abonnieren

Das gesamte Layout des Client ist ein CardLayout, welches uns ermöglicht gewünschte Cards anzuzeigen. Im Folgenden werden wir die Funktion der einzelnen Cards anhand von Screenshots erklären um einfacher ein Bild von der Oberfläche verschaffen zu können.

Startbildschirm

Wenn der Client gestartet wurde, landet der Benutzer direkt auf der Startcard, welches die Funktionen Login und Benutzer erstellen bietet. Bevor der Benutzer sich einloggen oder sich registrieren kann, muss er die Adresse des Servers mit Port angeben.



The screenshot shows a window titled "CardLayout" with a standard Windows-style title bar (minimize, maximize, close buttons). The main content area is titled "User Login" and contains two input fields: "Serveradresse" and "Serverport", each followed by a text box. Below these fields is a "Login" button. At the bottom of the window, there is a "Menü" section containing two buttons: "User erstellen" and "User Login".

User Erstellen

Wird ein neuer Benutzer angelegt, müssen alle erforderlichen Informationen eingegeben werden. Nodes des XMPP-Servers sind optional, d.h. sie müssen nicht angelegt, bzw. abonniert werden.

Nachdem auf den Button Erstellen geklickt wurde, gerät eine Menge an Operationen ins Rollen. Hier gibt es zwei verschiedene Optionen. Zum einen die, in der der User einen Account anlegt und gleichzeitig Orte zum Abonnieren angibt. Zum anderen die normale Registrierung ohne den XMPP-Server in Anspruch zu nehmen. Die Tabellenzeilen, welche eine Registrierung mit XMPP betreffen, sind grau markiert.

Client/Server	Operationsbeschreibung	Klasse	Methode
Client	Erstellung UserType	CreateUserPanel	buildUserType()
Client	Erstellung der Orte	CreateUserPanel	buildOptionalType()
Client	Senden der Daten (aufruf)	CreateUserPanel	Send()
Client	Erstellen eines XML-Strings	MarshallUnmarshall	writeUser()
Client	XML-String an Server senden	Connector	createUser()
Server	Ankommende Daten verarbeiten und abspeichern	UserService	postNew()
Server	Für jeden Ort wird ein Node angelegt	UserService	postNew()

Die BenutzerID ist von dem System generiert, da mit der geplanten ID keine Einzigartigkeit garantiert werden konnte. Somit entspricht diese nicht dem geplanten Schema: c_id=c?????.

The screenshot shows a Java Swing window titled "CardLayout" containing a "User Login" form. The form is organized into several sections separated by horizontal lines:

- Informationen:** Contains two text input fields labeled "Vorname:" and "Nachname:".
- Adresse:** Contains three text input fields labeled "Straße", "PLZ:", and "Ort:".
- Account:** Contains three text input fields labeled "LoginName:", "Passwort:", and "Email:".
- Menü:** Contains two buttons labeled "Abonieren" and "Erstellen".

At the bottom of the window, there is another section labeled "Menü" containing two buttons labeled "User erstellen" and "User Login".

Abonnement anlegen

Während der User Registrierung kann der User über den Button Abonnieren ein Abonnement erstellen der ihn Benachrichtigt wenn ein neuer Cache um ein angegebenes Gebiet erstellt wurde.

The screenshot shows a Windows-style application window titled 'CardLayout'. Inside, there is a 'User Login' section with several input fields. A 'Test' dialog box is open in the foreground, containing fields for 'PLZ', 'Latitude', 'Longitude', and 'Umkreis' (Radius), along with 'OK' and 'Abbrechen' buttons.

User Login Form Fields:

- Informationen:**
 - Vorname: Max
 - Nachname: Mustermann
- Adresse:**
 - Straße: Musterstraße 34
 - PLZ: 42343
 - Ort: MusterOrt
- Account:**
 - LoginName: [empty]
 - Passwort: [empty]
 - Email: [empty]
- Menü:**
 - [empty]

Test Dialog Box Fields:

- PLZ: 45234
- Latitude: 23.43123
- Longitude: 54.2342
- Umkreis: 20

Buttons:

- User erstellen
- User Login
- OK
- Abbrechen

Es können beliebig viele Orte abonniert werden. Erforderliche Informationen für ein solchen Ort sind: PLZ, bzw. Name, Latitude, Longitude und Umkreis in Kilometern um den durch Latitude und Longitude bestimmten Punkt herum. Ist alles korrekt eingegeben worden wird der Registrierungsvorgang welcher in der Tabelle „User registrieren“ beschrieben wurde eingeleitet.

Nach dem Login

Hat der Benutzer sich erfolgreich auf den REST- und XMPP-Server eingeloggt, hat er die Möglichkeit zwischen drei Menüpunkten zu wählen. Caches anzeigen, Feeds anzeigen und Cache erstellen. Alle drei Punkte werden wir hier kurz erläutern. Was den Login betrifft haben wir das nötigste implementiert. Es wird aber nicht überprüft ob der oder die Server zwischenzeitlich offline gegangen sind. Der Login dient lediglich zur Überprüfung ob ein Benutzer registriert ist oder nicht. Auch die Passwort Übergabe findet unverschlüsselt statt.

Caches anzeigen

Im Folgenden werden wir die Schritte bin hin zur Anzeige eines Caches im Detail anhand einer Tabelle mit Funktion, Methodenaufrufen und der Verwendeten Klassen erläutern.

Client/Server	Operationsbeschreibung	Klasse	Methode
Client	Erzeuge leere Tabellen	ShowCachePanel	createTable()
Client	Hole Cachedaten	ShowCachePanel	getCaches()
Client	Stelle Anfrage für Cacheliste an Server	Connector	getCaches()
Server	Holt Daten aus XML und gibt diese an Connector zurück	GeoCatchingCacheService	getAllNew()
Client	Erhaltenen String zu Liste umwandeln und an aufrufende Methode zurückgeben	Connector	getCaches()
Client	Für jeden Cache wird eine Zeile in der Tabelle erzeugt	ShowCachePanel	createTable()
Client	Zeige gefüllte Tabelle	ShowCachePanel	createTable()

Nun ist die Vorschautabelle mit allen verfügbaren Caches gefüllt. Der User kann nun wählen ob er einen bestimmten Cache als Ganzes angezeigt haben oder die gesamte Cacheliste Filtern möchte. Zum anzeigen aller Cacheinformationen muss auf einen angezeigten Cache in der Tabelle geklickt werden. Zum Filtern werden wiederum eine Menge Mechanismen aktiviert, welche aber für die Kommunikation zwischen Client und Server kaum Unterschiede zwischen den zuvor erklärten Mechanismen besitzen. Die gesamte Filterung wird über eine GET-Operation auf dem Server eingeleitet. Die Filterkriterien werden über Pathparams übergeben. Ebenfalls besteht die Möglichkeit, einen ausgewählten Geocache zu löschen. Vorgehensweise wie immer: Anfrage an Server mit Pathparams, Server löscht und gibt die neue Liste zurück.

Show Caches

Cacheliste

Cachename	ID	Owner	Datum
Rundblick	c0002	Kalle	Traditional Cache
b	730b52a4:13f705...	b	Multi Cache
sad	18603ae:13f70bf7...	dsf	Multi Cache
asd	65d10bf6:13f70ed...	asd	Multi Cache

Ausgewählter Cache

Attribut	Inhalt
Cache ID:	c0002
Location lat:	10.03214
Location lon:	111.23451
Name:	Rundblick
Datum:	java.util.GregorianCalendar[time=?,are...
Typ:	TRADITIONAL_CACHE
Owner:	Kalle
Parkplatz lat:	10.03214
Parkplatz lon:	111.23451
Schwierigkeit:	1.0
Terrain:	2.5
Land:	DE
Bundesland:	Nordrhein-Westfalen
Provinz:	Oberberg
Beschreibung klein:	Kleine Runde
Beschreibung gross:	Kleine Runde mit super Finale
Hinweise:	Man braucht keine Hinweise
Travelbug:	JA

Menü

Menu Delete Filter

Show Caches

Cacheliste

Cachename	ID	Owner	Datum
Rundblick	c0002	Kalle	Traditional Cache
b	730b52a4:13f705...	b	Multi Cache
sad	18603ae:13f70bf7...	dsf	Multi Cache
asd	65d10bf6:13f70ed...	asd	Multi Cache

Ausgewählter Cache

Attribut	Inhalt
Cache ID:	c0002
Location lat:	10.03214
Location lon:	111.23451
Name:	Rundblick
Datum:	java.util.GregorianCalendar[time=?,are...
Typ:	TRADITIONAL_CACHE
Owner:	Kalle
Parkplatz lat:	10.03214
Parkplatz lon:	111.23451
Schwierigkeit:	1.0
Terrain:	2.5
Land:	DE
Bundesland:	Nordrhein-Westfalen
Provinz:	Oberberg
Beschreibung klein:	Kleine Runde
Beschreibung gross:	Kleine Runde mit super Finale
Hinweise:	Man braucht keine Hinweise
Travelbug:	JA

Menü

Menu Delete Filter

Filter Caches

Distance:

Latitude:

Longitude:

Cachetype:

Schwierigkeit:

Terrain:

Dauer:

Travelbug:

OK Abbrechen

Cache anlegen

Werden Caches über die Eingabemaske angelegt, so wird für jeden Einzelnen Cache der XMPP-Server in Anspruch genommen. Wie das funktioniert und welchen Sinn das Ganze hat, werden wie in diesem Abschnitt näher erläutern. Die Eingabemaske wurde mit einer Testversion des GUI-Designers JFormDesigner erstellt. Eingegeben werden können alle Attribute, müssen aber nicht. Ein Cache lässt sich nur über seine automatisch generierte ID eindeutig identifizieren. Da wir für die Entwicklung eines geeigneten Systems ausschließlich Dummycaches verwendet haben, besteht weiterhin die Möglichkeit Dummycaches zu erstellen.

Create Cache

Name

Benutzer

Datum

Location lat location lon

Park lat Park lon

Cachetyp **MULTI_CACHE** ▼

Schwierigkeit **1.0** ▼ Terrain **1.0** ▼

Land

Bundesland

Provinz

Beschreibung klein

Beschreibung lang

Hinweise

Geschätzte Zeit **1.0** ▼ Travelbug **ja** ▼

Speichern **Menu**

JFormDesigner Evaluation

Die Vorgänge, die ab der Cacheerstellung ablaufen werden wir für einen besseren Überblick auch dieses Mal anhand einer Tabelle erläutern. Vorgänge, die aufgrund von Fehlern auskommentiert wurden, werden wir rot markieren, Vorgänge die auf dem XMPP-Server ablaufen in grau.

Client/ Server	Operations- beschreibung	Klasse	Methode
Client	Cachedaten werden verarbeitet	CreateCachePanel	createCache()
Client	Daten werden für das Versenden vorbereitet	CreateCachePanel	Send()
Client	Cachdaten werden zu sendbaren String verarbeitet	MarhsallUnmarshall	writeCache()
Client	Daten werden an Server geschickt (POST)	Connector	sendRequestAndData()
Server	Ankommende Daten werden verarbeitet	GeoCatchingCacheService	postNew()
Server	Hole alle Nodes von XMPP-Server	GeoCatchingCacheService	handleNodes()
Server	Gebe alle Nodes zurück	ConnectionHandler	getAllNodes()
Server	Hole Attribute aus Nodenamen	GeoCatchingCacheService	handleNodes()
Server	Vergleiche die Nodeattribute mit erstelltem Cache	GeoCatchingCacheService	handleNodes()
Server	Wenn Kriterien stimmen, hänge Payload an Knoten	GeoCatchingCacheService	handleNodes()
Server	Schreibe neuen Cache in XML	GeoCatchingCacheService	postNew()

Da der Vorgang: Hänge Payloadmessage an Knoten nicht umsetzbar war, haben wir keine sinnvolle Darstellung für Abonnierte Nodes gefunden.

Fazit

Die Entwicklung von GeoCatching hat uns in Bezug auf Anwendungssysteme gezeigt wie viel Aufwand hinter manchen Projekten steckt. Wir haben auch gelernt wie die Kommunikation der Programme und Anwendungen oder Webseiten im Hintergrund funktioniert. Der Umfang eines solchen Projektes hat unsere Vorstellungen bei weitem übertroffen. Das Projekt hätte von unserer Seite aus besser durchdacht werden müssen, damit die Struktur sauber gestaltet hätte werden können. Der Stück-für-Stück-Aufbau der Anwendungen hat ein Teil zur Unübersichtlichkeit beigetragen. Das Projekt hat uns gezeigt, dass es ohne genaue und strukturierte Planung schwer wird, die Struktur in allen Meilensteinen zu bewahren.

Probleme

Die Aufgetretenen Probleme werden hier tabellarisch aufgelistet

Problem	Ursache	Lösung	Verbesserungen
@GET-Methoden konnten keine einzelnen Caches zurückgeben	In Schleife Caches gefiltert	Gefilterten Cache außerhalb der Schleife speichern	Keine
Pfade der Dateien nicht dynamisch	Pfade waren absolut	Relative Pfade benutzen	Anwendungen laufen auf allen Systemen
Logs und Kommentare zu Caches	Müsste eigene Maske für erstellt werden	Erstellung von Kommentaren und Logs weglassen	Komplexität nicht weiter steigend
Befüllen JComboBox	Enums	Müssten für jede Combobox ein Array definieren	Keine doppelten Werte mehr
Payloadmessage an Knoten hängen	Vermutlich disconnect vom Server	Klassenstruktur ändern	Sollte dann funktionieren, aber nicht sicher
CacheIDs nicht wie Schema	Eindeutigkeit nicht gewährleistet	System generiert eindeutige IDs	IDs zwar kryptisch, aber eindeutig
Caches anhand Distanz von Koordinate filtern	Distanzberechnung vermutlich fehlerhaft	Neue Methode zur Berechnung	Sollte für alle Eingabedaten funktionieren

Mögliche Erweiterungen

Da das Grundgerüst der Anwendung steht und weitestgehend funktioniert, wäre die Entwicklung einer Smartphone-App möglich um jederzeit auf die Caches zuzugreifen, auch die Navigation via Googlemaps zu einem Cache wie im Konzept vorgesehen wäre mit einer Android-App möglich. Sollte man den ersten Cache gefunden haben könnte man so gleich vor Ort kommentieren(sobald es implementiert ist) oder sich einen weiteren Cache raussuchen. Auch eine Umkreis suche mit der genauen Position des Anwenders wäre dann dank des im Normalfall eingebauten GPS-empfängers sehr gut möglich. Des Weiteren wären die nicht umgesetzten Funktionen der Kommentare und Logs zu den einzelnen Caches implementierbar. Eine Userstatistik wäre ebenfalls eine denkbare Erweiterung.