

WBA Phase 2

Die Gruppe

Pascal Kottmann

Malte Odenthal

Meilenstein 1

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.

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 gewollt sind um die Daten die transportiert werden müssen möglichst gering zu halten.

Mögliche **Filterkriterien** wären hier:

- Längen- und Breitengrad
- Radius um einen Ort herum (z.B: 30 km Gummersbach)
- GeoCachetyp (Traditional, Multi, Mikro)

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 eine App zu erstellen, welche alle geforderten Kommunikationsverfahren abdeckt

Meilenstein 2

In Phase 2 war es erforderlich, unsere Projektvorstellungen in XML und XML-Schema umzusetzen. Letztendlich sind für unser Projekt drei XML und drei dazu gehörige Schemata entstanden, welche wir im Folgenden anhand ihrer Funktionen auflisten werden.

XML

- CacheList.xml
- User.xml
- UserInfo.xml

XML-Schema

- CacheList.xsd
- User.xsd
- UserInfo.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, inklusive Logs über Funde und Kommentaren zu diesem Cache. Diese Cachelist soll über das Internet an ein Smartphone-Endgerät gesendet werden, welches diese Daten weiterverarbeitet. 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 bis jetzt noch nicht implementiert haben ist das Einbinden von Bildern, welche die Umgebung der Caches zeigen.

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*s und *restrictions* sich in allen Dokumenten sehr ähnlich. Sinnvoll wäre es hier z.B. eine Datei mit den mehrfach vorkommenden Elementen zu haben, welche dann über *namespaces* eingebunden wird.

UserInfo

Diese Dokumente befassen sich ausschließlich mit der Verbindung von Benutzern und Caches. Hier werden für einen gewählten Benutzer alle erstellten und gefundenen Caches inklusive statistischer Informationen und Benutzerdaten aufgelistet. Es bildet somit das Bindeglied zwischen den vorangegangenen Dokumenten. Die Informationen könnten z.B. für die Öffentlichkeit einsehbar sein.

Unser Server soll später die Daten in einer SQLite-Datenbank abspeichern um einen schnellen und einfachen Zugriff zu gewährleisten. Somit wäre auch das Filtern einzelner Caches einfach zu realisieren.

Beschreibung

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 (meist PC - Server).

Die URL sollte "umgangssprachlich" nicht mehr als 255 Zeichen besitzen (Kompatibilität mit älteren Browsern)

Meilenstein 3: RESTFul-Webservice mit Grizzly und JAXB

Projektaufbau:

- ▲  de.odenthma.geocache.CacheClasses
 - ▷  BundeslandEnum.java
 - ▷  CacheListType.java
 - ▷  CacheType.java
 - ▷  CacheTypeEnum.java
 - ▷  InformationenType.java
 - ▷  KommentareType.java
 - ▷  KommentarType.java
 - ▷  LogsType.java
 - ▷  LogType.java
 - ▷  ObjectFactory.java
 - ▷  YesNoEnum.java
- ▲  de.odenthma.geocache.grizly
 - ▷  CopyOfGeoCatchingUserService.java
 - ▷  GeoCatchingCacheService.java
 - ▷  GeoCatchingClient.java
 - ▷  GeoCatchingServer.java
- ▲  de.odenthma.geocache.UserInformation.Classes
 - ▷  AccountType.java
 - ▷  AdressType.java
 - ▷  NameType.java
 - ▷  ObjectFactory.java
 - ▷  OptionalType.java
 - ▷  OrtsListeType.java
 - ▷  OrtsType.java
 - ▷  UserInformationType.java
 - ▷  UserListType.java
 - ▷  UserType.java
 - ▷  YesNoEnum.java
- ▲  de.odenthma.geocache.xml
 -  Bundesländer
 -  CacheList.xml
 -  CacheList.xsd
 -  CacheList1.xml
 -  CacheListNew.xml
 -  daten.txt
 -  testList.xml
 -  User.xml
 -  User.xsd
 -  User1.xml
 -  UserInformation.xml
 -  UserInformation.xsd
 -  UserInformation1.xml
 -  UserList.xml
 -  UserList.xsd
 -  UserListNew.xml

Beschreibung des Aubauss:

CacheClasses

Hier finden sich alle aus dem CachList.xsd-Schema generierten Klassen, welche für den Service genutzt und angesprochen werden.

Grizzly

Alle Services, welche für das Funktionieren eines RESTFul-Webservices benötigt werden

UserInformation

Hier finden sich alle aus dem UserInformation.xsd-Schema generierten Klassen, welche für den Service genutzt und angesprochen werden. Leider lässt sich nur mittels Service die id: b0001 löschen. Bei der id: b0002 bekommt der Service eine Menge Fehler.

XML

In dieses Package befinden sich alle XML- und XSD-Dokumente, welche für Phase 2 benötigt werden.

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?id=c0001.

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

Cachelist REST

Ressource	URI	Methode
Cacheliste abrufen	/cachelist	GET
Cacheliste via ID filtern	/cachelist/filter?id={id}	GET
Testcacheliste abrufen	/new	GET
Testcache erstellen	/new?id={id}	POST
Testcache löschen	/delete?id={id}	DELETE

Cachelist abrufen

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

Cachelist via ID filtern

Mithilfe der Methode GET auf die URI /Cachelist/filter?id={id} erhält man dank des Querryparams nur die Cachelist Inhalte unter einer bestimmten ID

Testcache abrufen

Mithilfe der Methode GET auf die URI /new/ wird die gesamte Testcachelist vom Server abgerufen

Testcache erstellen (prototypisch)

Mithilfe der Methode POST kann man über die URI /new?id={id} (Querryparams) einen neuen Testcache unter einer angegebenen ID erstellen (momentan noch über kopieren/vervielfältigen)

Testcache löschen

Über /delete?id={id}(Querryparams) lässt sich mit der Methode DELETE ein bestimmter Datensatz unter einer bestimmten ID löschen

Userlist REST

Ressource	URI	Methode
Userliste abrufen	/userlist	GET
Userliste via ID filtern	/userlist/filter?id={id}	GET
Testuserliste abrufen	/new	GET
Testuser erstellen	/new?id={id}	POST
Testuser löschen	/delete?id={id}	DELETE

Userlist abrufen

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

Userlist via ID filtern

Mithilfe der Methode GET auf die URI /Userlist/filter?id={id} erhält man dank des Querryparams nur die Userlist Inhalte unter einer bestimmten ID

Testuser abrufen

Mithilfe der Methode GET auf die URI /new/ wird die gesamte Testuserlist vom Server abgerufen

Testuser erstellen (prototypisch)

Mithilfe der Methode POST kann man über die URI /new?id={id} (Querryparams) einen neuen Testuser unter einer angegebenen ID erstellen (momentan noch über kopieren/vervielfältigen)

Testuser löschen

Über /delete?id={id} (Querryparams) lässt sich mit der Methode DELETE ein bestimmter Datensatz unter einer bestimmten ID löschen

Alle Dokumente für Phase 2 finden Sie unter der URL:

https://github.com/Odenthma/WBA2_P2