Logbuch - WBA2 - Phase 2

Web-basierte Anwendungen 2: Verteilte Systeme

Verfasst von **Simon Klinge** (11082448) zu dem Projekt: “**Fridgemanager**”.

Anmerkung:
Dieses Logbuch wurde erstellt um Gedankengänge und Probleme bei der Entwicklung unseres RESTful Webservice, der asynchronen Kommunikation und eines Beispielclienten für unser System: “Fridgemanager”, festzuhalten. Speziell auch Implementierungsdetails. (die so keinen Platz in der Dokumentation fanden. Einige Entwürfe wurden am Ende komplett ersetzt: z.B unser anfängliches Ressourcendesign.)
Dabei wurde nur bedingt Rücksicht auf Rechtschreibung und Satzbau gegeben. Ich bitte dies zu entschuldigen. Es ging primär darum, die eigenen Gedanken zu ordnen und schnell aufzuschreiben. Das ist auch der Grund warum das Logbuch in digitaler Form vorliegt: Schnelligkeit und die Möglichkeit unmittelbar Screenshots aus dem Workflow einzubinden. Ich denke meine Handschrift wäre unangenehmer zu lesen.
Viel Spaß beim lesen bzw. überfliegen.

14.05.13

Gedanken um die serverseitige Datenspeicherung

Neben den XSD-Schematas die wir bereits für die Kommunikation unseres REST-Services erstellt haben (profile.xsd, profiles.xsd, fridge.xsd, usw.), spiele ich zurzeit mit dem Gedanken eine weitere XSD für die persistente und serverseitige Speicherung zu erstellen.
Quasi eine Master-XSD die alle Daten in sich trägt.

Vermutlich könnte man auch unsere bisherigen Schematas als “Datenbank” verwenden. Diese Methode zieht allerdings unzählige Redundanzen mit sich, da etliche Querverweise und Referenzen zwischen den Schematas bestehen (z.B. beinhaltet Profile.xsd eine Liste mit kürzlich konsumierten Produkten. Für jede Zeile steht der Name und Verbrauchsdatum des Produkts. Allerdings beziehen sich diese Daten auf die Produktinstanz aus Product.xsd. Wenn wir also diese Schematas als “Datenbank” in betracht ziehen, das wäre einfach schwachsinnig und redundant.)

Die beste Lösung, für die serverseitge Datenverwaltung, ist wohl eine relationale Datenbank ala SQL. Dies würde jedoch den Aufwand nochmals immens erhöhen, und ist im Rahmen der Veranstaltung nicht vorgesehen.

So denke ich im Augenblick das ich mit meiner “Master-XSD” ganz gut liege. Ich taufe sie “fridgemanagerstorage.xsd”.

Ich werde vorerst nur die Profiledaten implementieren, um unsere Applikation erstmal an den beiden Ressourcen http://fridgemanager.de/profiles und http://fridgemanager.de/profiles/{id} mittels Jersey und Grizzly zu testen und ein Gefühl für eine REST-Implementierung zu bekommen.
Natürlich sind die Listen: kürzlich-verbrauchte-, und kürzlich-gekaufte-Produkte in der GET-Repräsentation für /profiles/{id} *noch* nicht realisierbar.

15.05.13

MyMarshaller.java

Bei einem HTTP-GET Request auf .../profiles/?name=Bernd muss nun serverseitig aus der fridgemanagerstorage.xml das entsprechende Java-Objekt erstellt werden - UNMARSHALL (XML → Java-Objekt). Daraus werden die nötigen Informationen entnommen und in ein Java-Objekt der JAXB-Klasse “Profile” gespeichert. Dieses Objekt muss nun in eine xml gewandelt werden - MARSHALL (Java-Objekt → XML). Dabei bin ich mir noch unsicher ob ich nur einen String mit dem Inhalt der XML an den Client schicke, oder eine konkrete Datei mit Endung .xml. [...]
Da diese Marshall-Operationen an vielen Stellen verwendet werden muss, - (Besonders das Marhalling. Schon jetzt mit nur einer Ressource und nur der GET-Operation und zwei Ausgabeformaten: "application/xml" und "text/html", wird sie zweimal benötigt.) - erstelle ich eine Klasse MyMarshaller die mir den Zugriff etwas erleichtert.

Namenskonvention für Service Klassen

Grobe Recherchen zu Jersey haben ergeben, dass ich für jede Ressource (oder auch mehrere) eine Klasse angebe und den Pfad über den Klassenamen mit `@Path ("/profiles")` angebe.

Ich halte es für sinnvoll eine Namenskonvention einzugehen, also Klassen die Gruppe von Ressourcen repräsentieren, werde ich in Zukunft mit KlasseService.java bezeichnen. Für die Ressource /profiles/ also ProfilesService.java. Zumal die Klasse Profiles.java schon durch die JAXB-Klasse besetzt. (Zwar in einem anderen Paket - wodurch die Nutzung gleichnamiger Klassen durch vorangegangener Paketbezeichnung wäre - den Code nur aufplustert, und schwerer verständlich macht.

update(01.06.13): Es scheint geläufiger zu sein diese Art von Klassen mit ...Resource.java zu bezeichnen. Service-Klassen umbenannt...

18.05.13

Neue Gedanken um die serverseitige Datenspeicherung, (un)marhallen nicht so häufig nötig wie gedacht

Ein aufschlussreiches Gespräch mit einem Betreuer hat ergeben, das ich zum einem nicht “per Hand” (un)marshallen muss, da mir Jersey die Arbeit abnimmt und aus meinen JAXB-Object eine entsprechende .xml erzeugt. Ich dachte ich müsse immer einen String zurückgeben bzw. akzeptieren. Stattdessen kann ich für den Rückgabewert - einer HTTP-Methode - einfach eine JAXB-Instanz zurückgeben und Jersey wandelt dieses Object entsprechend des @Produces -Param um.

Meine “Master.xsd” hat sich als schlechte Lösung “als Datenbank” herauskristallisiert. Der Hauptgrund dafür: Server muss jedesmal (z.B. bei einer GET-Anfrage) die gesamten Daten unseres System in den Hauptspeicher laden, obwohl er immer nur einen kleinen Teil benötigt. → Performenceverflust. Im schlimmsten Fall Hauptspeicher ausgelastet.

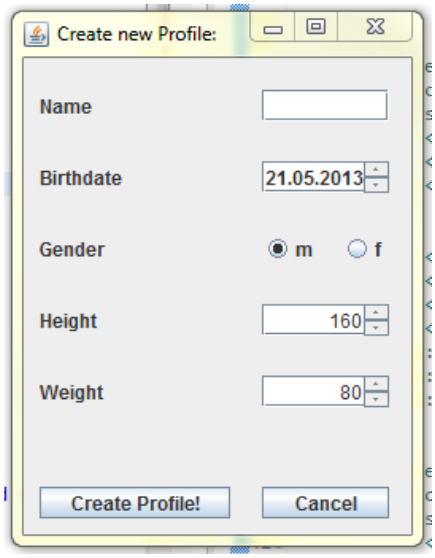
Ich denke ich werde simple Textdateien erstellen. So das für jedes Profil, jeden Kühlschrank etc. eine eigene Datei existiert. Diese werde ich dann gemäß unserer Ressourcen-Struktur ordnen. (z.B. liegt die Datei 123.txt im Ordner: data/fridges/1/profiles/123.txt und speichert die Daten eines Profils. An der numerischen Bezeichnung erkennt man auch direkt die ID)

20.05.13

Kurzer Swingausflug

Neben der Arbeit, unsere Ressourcen REST-fähig zu machen (Was wesentlich komplizierter wurde als anfänglich gedacht, besonders weil unsere Ressourcen bzw. deren Repräsentationen aufwändig berechnet werden müssen. Z.B. Liste kürzlich konsumierter Produkte in der Ressource “.../profiles/{id}” → so müssen alle Produkt-Insanzen im Kühlschrank durchschaut werden nach Besitzer verglichen und Datum abgecheckt werden.), implementierte ich schonmal Teile für die Client-Applikation:

Unsere Ressource “.../profiles/{id}” gehorcht nun auf GET und PUT. Zum anlegen eines neuen Profils nun folgendes Formular (in einfachen Swing programmiert):

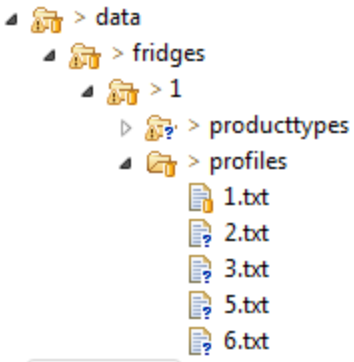


:
Will man nun diese Daten mit PUT an die Ressource /fridges/{fridgeid}/profiles{profileid} übermitteln, stellt sich die Frage woher die ID´s genommen werden. Für den Anwender dieser Client-Applikation sollten die ID´s irrelevant sein. Es sollte nicht in der Verantwortung des Anwenders liegen (korrekte) ID´s auszuwählen.
Die fridgeid sollte im diesem Anwendungsfall - neues Profil anlegen - bekannt sein, da der Nutzer erst einmal einen Kühlschrank auswählen wird, eine Übersicht aller Profile zu diesem Kühlschrank bekommt, und erst dann! macht es Sinn ein neues Profil anlegen zu können.
Für die profileid muss sichergestellt werden das es diese id noch keinem anderen Profil zugewiesen wurde. Es müssen also alle Profile zu dem Küschlschrank(fridgeid) durchgegangen werden und eine freie profileid gewählt werden.
(update: Das anlegen eines Profils ist nicht wirklich charakterisierend für unser System und wird daher in unserer Beispielapplikation nicht umgesetzt sein. Die Implementierung für das POST hingegen schon)

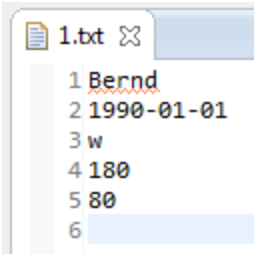
24.05.13

Textdateien - nein danke

Die Ressourcen `fridge/{id}/profiles/{id}` und `fridge/{id}/profiles` reagieren auf GET und PUT-Methode, NUN aber werden die Daten hierzu aus Textdateien ausgelesen. Die Ordnerstuktur sieht etwa folgendermaßen aus:



Eine Textdatei hat den Inhalt:



Der Umgang mit Textdateien(bzw. Dateien mit zeilenweisen ASCII-Einträgen) hat sich aber als extrem Aufwendig erwiesen. Das liegt vor allem an der Umwandlung und Zusammenstellung von .txt → .xml, bzw. .txt → JAXB-Instanz (Da Jersey den Schritt JAXB-Instanz → .xml abnimmt)
So waren allein für die GET-Methoden auf `fridge/{fridgeid}/profiles/{id}` und `fridge/{id}/profiles` folgender code notwendig:

In der Methode readProfiles werden sämtliche Dateien durchgegangen und die Namen herrausgepickt.

.
.
.

```
public static Profile readProfileByID(int fridgeid, int id) throws ...{
                reader = new BufferedReader(new
FileReader("data/fridges/"+fridgeid+"/profiles/"+id+".txt"));
                Profile p = new Profile();
                p.setName(reader.readLine());

                p.setBirthdate(DatatypeFactory.newInstance().newXMLGregorianCalendar(reader.readLine()))
                ;
                p.setGender(reader.readLine());
                p.setHeight(Float.parseFloat(reader.readLine()));
                p.setWeight(Float.parseFloat(reader.readLine()));
                return p;
        }

        public static Profiles readProfiles(int fridgeid) throws ... {
                Profiles profiles = new Profiles();
                String[] list = getFilelist("data/fridges/"+fridgeid+"/profiles");
                int[] profileIDs = new int[list.length];
                for(int i=0; i<list.length; i++) {
                        profileIDs[i] = Integer.parseInt(list[i].substring(0,
list[i].lastIndexOf('.')));
                        Profile p = readProfileByID(fridgeid, profileIDs[i]);
                        Profiles.Profile p1 = new Profiles.Profile();
                        Profiles.Profile.Name name = new Profiles.Profile.Name();
                        name.setValue(p.getName());
                        name.setHref(TestServer.url+"/fridges/"+fridgeid+"/profiles/"+profileIDs[i]);
                        p1.setName(name);
                        profiles.getProfile().add(p1);
                }
```

```
                return profiles;
        }

        public static String[] getFilelist(String path) {
                File dir = new File(path);
                return dir.list();
        }
```
.
.
.

Für einen außenstehenden nur mühsam zu verstehen. Und dabei fehlen noch die Angaben zu den kürzlich konsumierten/gekauften Produkten.
Es handelt sich also noch um sehr einfache Strukturen. Bei komplexeren Datenstukturen z.B. producttype1.xml, in der z.B. eine Liste von Zutaten angeben wird. So muss erstmal entschieden werden wie diese Liste in der Textdatei erkannt wird: -ein Anfang und ende makieren? -jede zutat mit einen Präfix versehen? Und das noch alles Programmieren?!

Ich werde mich von handgeschriebenen Textdateien wieder distanzieren. Ich mache mir zurzeit viel zu viele Gedanken, die in Richtung Datenbankentheorie gehen. Und die Zeit rennt.

Hier nochmal die drei Vor-/Nachteile von Textdateien gegenüber XML-Dateien, bezogen auf die “Datenbank”.

Textdateien:

- + Performanter/weniger Speicherbedarf
- - hoher Programmieraufwand. Von Textdateien zur REST-Schnittstelle bzw. “Portierung” von einzelnen Zeilen aus .txt´s zu JAXB-Object´s, extrem aufwendig
- - für Außenstehende Undurchsichtig, da Struktur der Datei vom Programmierer festgelegt (kein entsprechendes Schemata wie bei XML)

XML-Dateien:

- - mehr Speicherbedarf (Tags etc.) zusätzliche Schematas.
- + Portierung von xml → JAXB-Instanz erfolgt intuitiver. Lese- Schreibzugriff durch JAXB-Klassen und Marshalling wesentlich einfacher!
- + Datenstruktur der .xml für Außenstehende sehr gut nachvollziehbar.
- + xml´s können anhand der Schemata validiert werden.

desweiteren gäbe es die Möglichkeit java-Objekte statt xml zu speichern. Dabei

- + entfällt das marshalling → kleiner Performancegewinn
- - Daten nicht mehr “per-Hand” modifizierbar, nurnoch über Programmcode

Ich werde den Drang nach Performancegewinn in den Hintergrund schieben und entscheide mich für die XML-Variante.

Zu bedenken ist: eventuell müssen neue .xsd´s und enstprechende JAXB-Klassen angelegt werden, die nur den Sinn haben die Daten serverseitig zu verwalten.

25.05.13

Problemkind: producttype.

Zur Erinnerung: Die Ressource: fridge/{id}/producttpes/{id} gibt einen ProduktTyp aus einem Kühlschrank, dieser beinhaltet zum einen allgemeine Informationen zum Produkt. Name, Beschreibung, Zutaten, Nährwerte, etc, zum anderen den aktuellen Bestand und Mindestbestand.
Serverseitig sollten diese beiden Informationen getrennt sein. Die Daten zum Bestand (stockData) sind für jeden Kühlschrank individuell, die Informationen zu einem Produkt hingegen sind über die Kühlschränke hinweg gültig. Zur Speicherung muss also eine Trennung von Produktinformationen und Produktbestand vorgesehen werden.
Ich erstelle 2 neue xsd´s:

- productinformationLOCAL.xsd und
- producttypeLOCAL.xsd

das LOCAL soll Kenntlich machen das die Schemata zur lokalen Verwaltung von Daten eingesetzt werden und wenig mit der REST-Schnittstelle zu tun haben. Der Schlüssel mit denen die beiden Schemata verknüpft werden ist das name-tag. (Erstelle ich mit diesem Vorhaben schon soetwas wie eine XML-Datenbank?)

…
Sehr nervigen Fehler gelöst. Hat Stunden gekostet. Beim unmarshallen wird auf die ObjectFactory zurückgegriffen, diese war nicht richtig generiert, da sie beim erstellen einer JAXB-Klasse aus einer XSD überschrieben wurde und nur die Konstruktoren aus einer XSD enthielt. Ergänze nun die ObjectFactory manuell...
Immerhin ein besseres Verständnis der ObjectFactory gewonnen.

zu kompliziert. Jedes Profil, jeder Produkttyp eigene xml-file?

Während ich hier fröhlich am coden bin und die GET-Methoden für .../producttypes/{id} und .../producttypes implementiere, muss ich doch kurz inne halten und bemerke zu meinem erstaunen, dass der Code durch meine .xml-Variante nicht viel weniger wird - im Vergleich zu der Variante mit Textdateien. (Es geht immernoch um die Datenverwaltung)

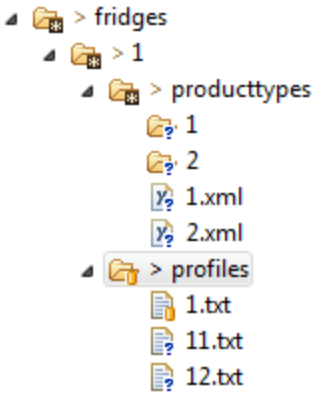
```java
@GET
@Produces({ MediaType.APPLICATION_XML, MediaType.TEXT_XML })
public ProductTypes getProducttypes(@PathParam("fridgeid") int fridgeid) {

    // Benötigte Daten beziehen
    String[] list = Data.getFilelist("data/fridges/"+ fridgeid +"/producttypes");
    int[] producttypesIDs = new int[list.length];
    int i = 0;
    for(String s : list) {
        if(s.matches("(.*).xml")) { // Nur XML keine Verzeichnisse
            System.out.println(s);
            producttypesIDs[i++] = Integer.parseInt(s.substring(0, s.lastIndexOf('.'))); // id für href-Attr speichern
        }
    }

    // Nun die Liste ProductTypes erstellen
    ProductTypes pts = new ProductTypes();

    ProductTypes.Producttype producttype = new ProductTypes.Producttype();
    for(int j=0; j<i; j++){
        ProductTypeLOCAL ptL = (ProductTypeLOCAL) MyMarshaller.unmarshall("data/fridges/"+ fridgeid + "/producttypes/"+ producttype
        producttype.setName(value);
        producttype.setHref(value);
        producttype.setStock(value);
        pts.getProducttype().add(producttype);
    }
```

Und plötzlich wird mir klar das ich von einem Extremum zum anderen gesprungen bin. Zuerst hatte ich den Gedanken ALLE Daten in eine xml zu speichern - was sich aus genannten Gründe als weniger geeignet herausstellte.
Nun aber bilde ich jede Ressource als eigene XML ab. D.h. jedes Profil, jede Produktinstanz und jeder Produkttyp bekommt eine eigene XML. (Produktinstanzen als Unterordner zu Producttypes. Das Unterverzeichnis trägt den Namen der id)



Das ist nun auch wieder nicht im Sinne der Performence - da zu viele Dateiaufrufe notwenig sind, auf Dateien die mitunter gerade mal 5 Zeilen beinhalten.
Ich brauche also ein gesunden Mittelweg. Die Profile können für jeden Kühlschrank (einzeln!) in eine XML gepackt werden - die die Liste enthält, ebenso wie die Produkttypen und die Produktinstazen.
Es ergeben sich folgende xsd´s:

- producttypesLOCAL.xsd (hieß zuvor producttypeLOCAL.xsd (ohne ‘s’))
- profilesLOCAL.xsd
- productsLOCAL.xsd

Zu beachten: In den ...LOCAL.xsd´s müssen nun auch die id´s festgehalten werden. Diese waren vorher am Dateinamen erkenntlich (s.o.), da nun nur eine xml für producttypes und profiles verwendet wird, entfällt diese Ordnerhierarchie.

Erkenntnissgewinn

Niemals die id aus der URI unmittelbar als Listenindex verwenden. Warum? Nun, um genauer zu beschreiben was ich meine:

```
@GET
@Path("/{id}")
@Produces({ MediaType.APPLICATION_XML, MediaType.TEXT_XML })
public ProductType getProducttypeByID(@PathParam("id") int id, @PathParam("fridgeid") int fridgeid) thr

    ProductTypesLOCAL ptL = (ProductTypesLOCAL) MyMarshaller.
            unmarshall("data/fridges/"+ fridgeid + "/producttypesLOCAL.xml");

    ProductInformationLOCAL piL = (ProductInformationLOCAL) MyMarshaller.
            unmarshall("data/productinformation/" + ptL.getProductType().get(id).getName() + ".xml");

    return createProductType(ptL.getProductType().get(id), piL);
}
```

Hier benutzte ich die id aus der Ressource: .../producttypes/{id}. Die producttypes sind in der producttypesLOCAL.xml als Liste gespeichert. Die obige Lösung hat aber ein gehörigen Haken.

- Die id´s müssen lückenlos, inkrementel verlaufen, d.h. wenn ein Producttype gelöscht wird müssen alle anderen “nachrücken”, andernfalls würde der Zugriff nicht funktionieren
- Das wiederum hat zur Folge das die URI auf einen producttypen, nicht beständig ist, was gegen ein Designprinzip von Ressourcen spricht. Zitat: “Cool URI´s don’t change” - Tim Berners-Lee

Die “richtige” Lösung: Die Liste aus producttypesLOCAL.xml muss durchgegangen werden nach dem id-Wert verglichen werden. Dies ist auch der Grund weshalb das id-Attribut in der producttypesLOCAL.xml überhaupt eine Existenzberechtigung hat.

Erwähnenswerte Unterschiede zwischen productsLOCAL.xsd und products.xsd bzw. product.xsd

(Zur Erinnerung: In unserem System bezeichnet product eine “Instanz” von producttype.)
Rechte/Besiter Owner:
In der REST-Schnittstelle wird dem Client der Besitzer anhand eines Strings und einer URI mitgeteilt. Der String ist der Name des Besitzer bzw. des Profils und die URI gibt die Ressource zu dieser an. So kann der Client leicht auf weitere Informationen zu dieser Person zurückgreifen:

```
<xs:element name="owner">
        <xs:complexType>
        <xs:simpleContent>
        <xs:extension base="xs:string">
                <xs:attribute name="href" type="xs:anyURI" use="required"></xs:attribute>
        </xs:extension>
```

```
        </xs:simpleContent>
        </xs:complexType>
</xs:element>
```

In der productsLOCAL - unserer “Datenbank” - ist das jedoch nicht notwendig. Es genügt an dieser Stelle den Schlüssel - also die id zum entsprechenden Profil anzugeben:

```
<xs:element name="ownerID" type="xs:int"/>
```

27.05.13

producttypesLOCAL - stock-wert überflüssig?

Wenn ein Produkt verbraucht wird (HTTP-Delete) oder hinzugefügt (HTTP-Post/Put) ändert sich der aktuelle Bestand - klar, dieser Wert kann durch die producttypeninstanzen (products) also über Liste der in productsLOCAL.xml enthaltenden Informationen berechnet werden, bzw. muss berechnet werden.
Macht es Sinn den stock-wert in producttypesLOCAL.xml zu speichern, wenn er doch aus productsLOCAL.xml abgeleitet werden kann? - Es wäre eine kleine Redundanz
Spart es Rechenleistung wenn beim Hinzugefügen/Entfernen einer Produktinstanz nur der Wert aus producttypesLOCAL.xml geändert werden muss und nicht jedesmal die Liste aus productsLOCAL.xml gezählt werden muss? - möglicherweise
Für <minStock> und <maxStock> stellt sich diese Frage glücklicherweise nicht.

GET - geschafft

GET-Methoden für die Ressourcen:

- fridge/{id}/profiles
- fridge/{id}/profiles/{id}
- fridge/{id}/producttypes
- fridge/{id}/producttypes/{id}
- fridge/{id}/producttypes/{id}/products
- fridge/{id}/producttypes/{id}/products/{id}
- fridge/{id}/notifications
- fridge/{id}/notifications/{id}

erfolgreich implementiert.
Die Informationen werden ausunserer “XML-Datenbank” mit 4 xml Dataien bzw. 4 Entities (für jeden Kühlschrank) entnommen:

- profilesLOCAL.xml
- producttypesLOCAL.xml
- productsLOCAL.xml
- notificationsLOCAL.xml

und

- productinformation/PRODUKTNAME.xml (Kühlschrankübergreifend)

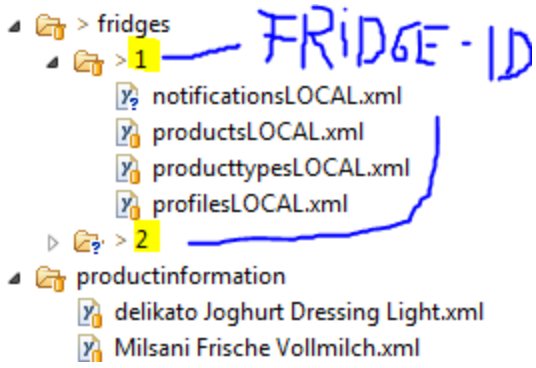


Abb: Datenstruktur auf dem Server.

Es fehlen noch folgende Ressourcen:

- fridge ???
- fridge/{id}
- In fridge/{id}/profiles/{id}: Angaben zu den kürzlich verbrauchen- und konsumierten Produkten

Da uns die Zeit wegrennt werde ich diese vorerst auslassen.

Als nächster Schritt stehen die POST und DELETE-Methoden an.

29.05.13

Gedanken zu POST und PUT:

Ein guter Beitrag im Forum:
"Great both can be used, so which one should I use in my RESTful design:

You do not need to support both PUT and POST.

Which is used is left up to you. But just remember to use the right one depending on what object you are referencing in the request.

Some considerations:

- *Do you name your URL objects you create explicitly, or let the server decide? If you name them then use PUT. If you let the server decide then use POST.*
- *PUT is idempotent, so if you PUT an object twice, it has no effect. This is a nice property, so I would use PUT when possible.*
- *You can update or create a resource with PUT with the same object URL*
- *With POST you can have 2 requests coming in at the same time making modifications to a URL, and they may update different parts of the object."*

Unser Ressourcendesign schreit förmlich nach POST-Methoden! Zum einen haben wir zu allen Ressourcen auch übergeordnete Listen auf denen POST-Requests hervorragend anwendbar sind, zum anderen haben wir uns für Identifikationsnummern entschieden, die vom Server zugewiesen werden sollten.
Somit hat sich auch die Frage vom 20.05 geklärt, - woher die Client-Applikation weiß welche id er bei dem erstellen eines Profils verwenden soll - was bei einem PUT ja nötig wäre.
(Zum updaten einer Ressource würde die Implementierung eines PUT´s dennoch Sinn machen. Das updaten von Ressourcen werde ich - aus Zeitgründen - vorerst überspringen, da es in unserem Fall nicht so schwer wiegt wie GET, POST oder DELETE)

Implementierung: POST

- @Consumes({ MediaType.APPLICATION_XML}) statt @Produces..
- Als Parameter ein JAXB-Object angeben, das in Form des @Consumes vom Clienten angegeben wird
- unmarshalling am Anfang und marhalling am Ende
- response-code zurückgeben

In unserem Fall sieht das Vorgehen für einen POST immer ähnlich aus

1. “Datenbank” nach einer noch nicht vergebenen id durchsuchen (unmarshall)
2. jaxbObject entsprechend anlegen und Attribute aus dem übermittelten jaxbObject (Consumiert als xml)
3. jaxbObject als xml in die Datenbank speichern (marshall)

```java
@POST
@Consumes({ MediaType.APPLICATION_XML})
public Response addProfile(@PathParam("fridgeid") int fridgeid, Profile p) throws JAXBExc
    ProfilesLOCAL psL = (ProfilesLOCAL) MyMarshaller.
            unmarshall("data/fridges/"+ fridgeid + "/profilesLOCAL.xml");

    // Nach einer freien Profile-id suchen
    int freeID = -1; boolean found;
    for(int i=1; i<=50 && freeID==-1; i++){
        found = true;
        for(jaxbClasses.ProfilesLOCAL.Profile profile : psL.getProfile()){
            if(profile.getId() == i) { // id belegt?
                found = false;
                break;
            }
        }
        if(found) // id noch frei?
            freeID = i; // übernehmen
    }

    // Neues Profil anlegen
    jaxbClasses.ProfilesLOCAL.Profile profile = new jaxbClasses.ProfilesLOCAL.Profile();
    profile.setId(freeID);
    profile.setName(p.getName());
    profile.setBirthdate(p.getBirthdate());
    profile.setGender(p.getGender());
    profile.setHeight(p.getHeight());
    profile.setWeight(p.getWeight());

    psL.getProfile().add(profile);

    // Daten auf Platte speichern
    MyMarshaller.marshall(psL, "data/fridges/"+ fridgeid + "/profilesLOCAL.xml");

    // Neu erstellte URI in Repsone angeben:
    return Response.created(new URI("fridges/"+fridgeid+"/profiles/"+freeID)).build();
}
```

Es werden nicht alle Attribute die als xml übermittelt werden berücksichtigt, was ja durchaus legitim ist. Z.b wird bei einem POST auf fridge/producttype/{id}/products, der Name des Produkts ignoriert da er sich aus der URL (producttypeid) ermitteln lässt. (Ohnehin wird in einer productsLOCAL.xml nur die id gespeichert.)

31.05.13

POST-geschafft

POST-Methoden auf die Ressourcen:

- fridges/{id}/profiles
- fridges/{id}/producttypes
- fridges/{id}/producttypes/{id}/products
- fridges/{id}/notifications

erfolgreich implementiert und getestet.

Nächste Aufgabe: DELETE-Methoden
Mir stellen sich folgende Fragen:

- Was passiert wenn ein Profil gelöscht wird, auf das andere Ressourcen verweisen? Z.b. das Profil noch Rechte auf eine Produktinstanz hat
    - Lsg: Im Idealfall werden alle Produkte/Notifications durchgegangen und der Besitzer gelöscht oder einem allgemeinen Profil zugewiesen.
      Man könnte auch vorraussetzen das alle Produkte und Nachrichten die zu ihm

gehören erst gelöscht sein müssen bzw. beim Löschen des Profils diese gelöscht werden.
Ich werde diesen Fall vorerst ignorieren

- Was passiert mit den Produktinstanzen wenn ihr Producttyp gelöscht wird?
    - Lsg: Es sollten alle Productinstanzen gelöscht weden. (kaskadierendes löschen) oder: Vorraussetzen das es keine Instanzen zu diesem Produkttyp gibt. (bzw. stock = 0)

XMPP

Welche Leafs(Topics)?

Die asynchrone Kommunikation in unserem System beschränkt sich auf das Empfangen von Benachrichtigungen(notifications), dabei werden Typen unterschieden: Typ: warning, oder Typ: notification...
Eine Möglichkeit wäre die Nachrichten nach Typ zu abonnieren und dementsprechend die Topics zu setzen. Evt. weiter unterscheiden:
Benachrichtigen wenn (Topics)

- Produkte ablaufen
- Produkte unberechtigt verbraucht werden
- Nachbestellt wurde

Wer ist Publisher, wer subscriber?

Publisher:

- Der webservice/server
- evtl. Client(App)

Sub:

- Client (App)

Welche Daten sollen übertragen werden?

Da unsere Nachrichten im Webservice eine eigene Ressource darstellen, brauchen wir nur id´s zu dieser Ressource verschicken. Besser: die URI verschicken. => light-Ping
….....

07.06.13

Neues Ressourcendesign

Unser Ressourcendesign wurde von den Betreuern kritisiert. Und zwar Subressourcen auf den Kühlschrank.
Z.B. wäre der Fall, dass ein Profil mehreren Kühlschränken angehört, damit nicht abgedeckt. Und wir würden uns so “Freiheiten nehmen”.

Ich glaube unser Fehler war es, die Ressourcen zu nah mit dem Hintergedanken einer Clientapplikation designt zu haben. Scheinbar sollten die Ressourcen in etwa so strukturiert sein

wie die zugrunde liegenden Datenbank - um möglichste Flexibilität zu gewährleisten. So fallen z.B. die Liste der kürzlich verbrauchten Produkte Produkte in der Ressource /profiles weg, zwar ist diese Information ein nice to have in Bezug auf ein Profil, kann aber durch die Clientapplikation über andere Ressourcen (/products) berechnet werden.

Wir haben folgendermaßen umstrukturiert::
/fridges
/fridges/{id}
/profiles
/profiles/{id}
/producttypes
/producttypes/{id}
/products
/products/{id}
/notifications
/notifications/{id}

(Man könnte notifications als Subressource zu einem Profil setzen, oder auch products zu einem Profil. Allerdings scheinen Subressourcen nicht gut bei den Betreuern anzukommen. Und der ach so wichtige Fall das man alle Nachrichten über alle Benutzer hinweg sich ausgeben möchte ja nicht vernachlässigt werden.)
Das setzt natürlich Voraus das wir mehr mit Query-param hantieren müssen, um die Profile Producte etc. zu einem Kühlschrank anfordern zu können
Der Kühlschrank bildet so etwas wie eine Konzeptressource, die verschiedene Ressourcen miteinander verkpnüpft.

Alle .xsd-Files werden neu geschrieben. Jede GET, POST und DELETE wird überarbeitet. Zahlreiche QueryParams implementieren.
Das wird ein schönes Wochenende! :)

XML-Datenbank - Grobe Darstellung

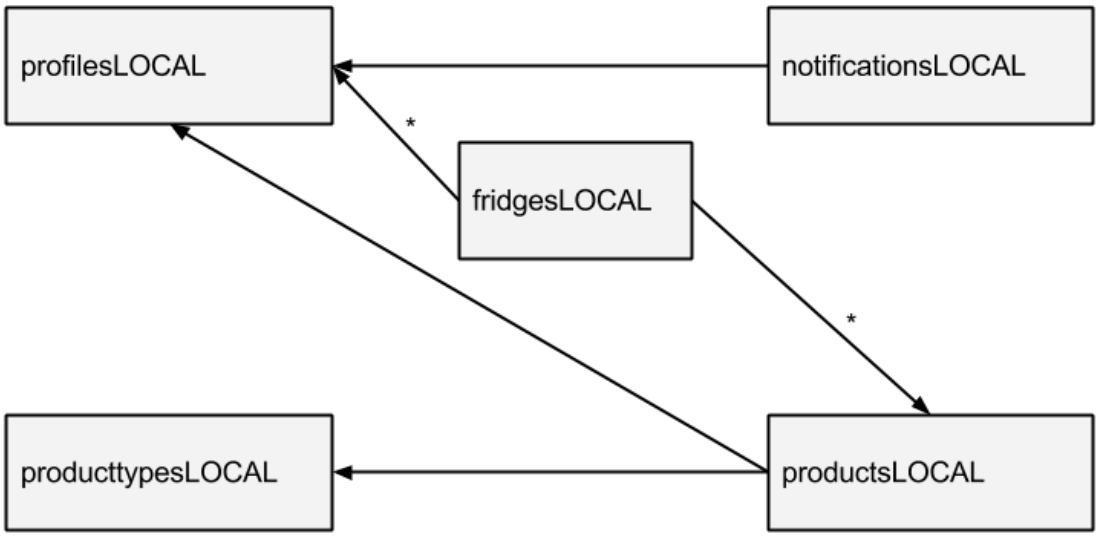


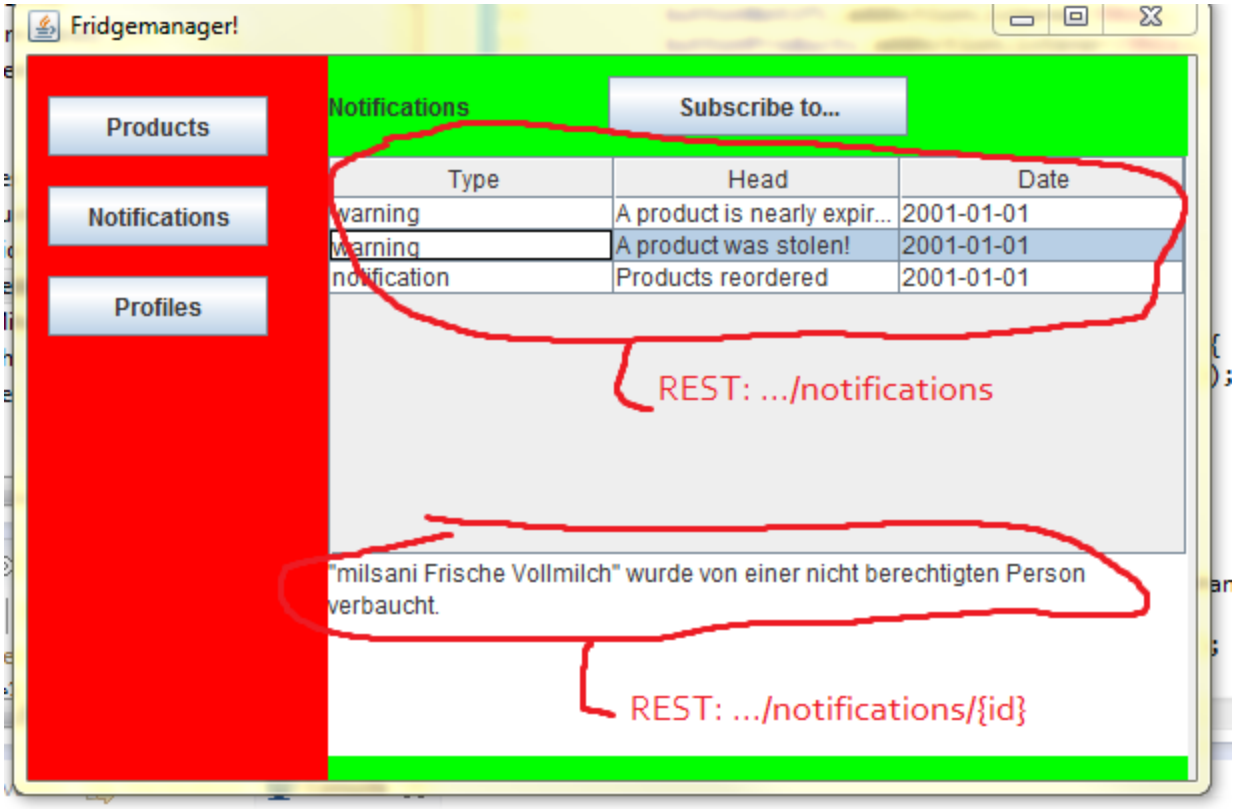
Abb: Die Pfeile verdeutlichen die Beziehungen zwischen den Entitäten. Eine Nachricht speichert eine id zu einem Profil. products speichert eine id zu producttypes und eine zu profiles. Der Kühlschrank selbst verweist auf mehrere(*) Produkte und Profile...

In den REST-Repräsentationen sind die Pfeile evtl. bidirektional! (Mit vollständigen URI´s)

TimeSimulator.java
Unsere asynchrone Kommunikation findet erst über einen längeren Zeitraum statt (z.B. wenn ein Produkt abläuft). Also haben wir eine Klasse erstellt die diesen Zeitraum simuliert (TimeSimulator) Die Klasse verwaltet ein Datum und besitzt die Methode “sleep1day” was den Thread ein kurze Zeit einschlafen lässt und das Datum um einen Tag erhöht. (Man hätte die Klasse auch gut DateSimulator.java nennen können...)

Wann published der Server eine Nachricht(Item)?
Nach jedem Tag prüft der Webservice alle Produkte auf das Haltbarkeitsdatum und sendet - vorrausgesetzt das Datum ist (fast) abgelaufen - eine Nachricht an das Profil diesen Produkts.

Swing Notifications



Man kann jetzt schon die Früchte unseres Ressourcendesign begutachten... Für die Liste der Nachrichten brauchen wir auch nur die Liste der Nachrichten anfordern, ohne den eigentlichen Inhalt für alle Nachrichten empfangen zu müssen. Wenn nun eine Nachricht ausgewählt wird, wird über den entsprechenden Hyperlink ein GET ausgeführt und man kommt an den eigentlichen Text der Nachricht.

Unsere Beispielapplikation

Wir wollen nur ein paar Kernfunktionalitäten auf einfache Weise demontrieren. Es soll deutlich werden welche Möglichkeiten es gibt. Und da unser neues Ressourcendesign nun sehr flexibel ist, (Was anfänglich nicht der fall war) ist noch viel mehr möglich.

Wir beschränken uns auf drei Oberflächen (JPanels):

- Fridges
    - “Blick in den Kühlschrank”. Je nach Profil werden einem die Kühlschränke gelistet **(queryParam auf .../fridges?profileid={id})** also alle Kühlschränke, die zu dem Profil eine Referenz tragen.
    - User kann einen Kühlschrank auswählen
    - Produkte in ausgewählten Kühlschrank werden in einer Liste nach Produkttyp und Bestand gelistet.
    - Buttons zum verbrauchen oder eintragen eines Produkts
    - Button für genaue Informationen zu einem Produkttyp
    - (wenn genug zeit) User kann neuen Produkttype anlegen

- Notifications
    - Liste von Nachrichten
    - User kann bestimmte Nachrichten-Arten abbonieren
- Profiles
    - Liste von Profilen
    - Wählt Profil aus
    - Informationen zum Profil

Jede Oberfläche entspricht eine Klasse:
- FridgePanel.java
- NotificationPanel.java und
- ProfilePanel.java

sie erben jeweils von JPanel und sind in der Klasse TabbedPanel in einem JTabbedPanel zusammengefasst, was sie gut navigierbar macht... (anders als oben in der Abbildung)

Zugriffe vom Clienten auf den Webservice, ausgelagert auf die Klasse RESTHandler.java

Es wurde ein wenig unübersichtlich in den Panel-Klassen auch die REST-Aufrufe unterzubringen - speziell weil einige Zeilen Code von Nöten waren um aus den “gegetteten” jaxb-Objekten auch die entsprechenden Daten ins entsprechende Format - je nach Swing-element, zu bringen.
Die Kommunikation mit dem Webservice findet clientseitg jetzt nur in der Klasse RESTHandler statt, was schön ist, da die visuellen GUI-Elemente nun besser von der logischen Stuktur getrennt sind

POST auf .../products Ergänzung

Wenn ein Produkt hinzugefügt wird muss die Referenz auch in dem entsprechenden Fridge vorliegen. Hatte kurz überlegt den Clienten unmittelbar nach dem POST ein PUT auf den Fridge auszuführen und ihm so die Produkt-Referenz zu übergeben... was aber totaler Schwachsinn ist; der Server muss mit dem Post auch den Kühlschrank “updaten” alles andere macht keinen Sinn.

23.06.13

Es geht aufs Ende zu

und ich muss sagen, ich habe eine Menge gelernt. Zu beginn dieses Semesters waren mir xml, xsd, jersey, xmpp, smack und vor allem der Begriff REST(ful) noch völlig unbekannt.
Mir ist klar, das ich noch lange nicht alles gänzlich durchdrungen habe, dennoch erahne ich Möglichkeiten sich hinter diesen Systemen verbergen. Nun selbst einen kleinen Webservice bereitgestellt zu haben, auf den 2 Applikationen über dieselbe Schnittstelle zugreifen (Browser und swingapp) ist schon ein befriedigendes Ergebniss. Auch die ganzen Gedanken um die Datenspeicherung serverseitig im Kontrast zu jenen Daten die über den Websevice verschickt

werden, waren für mich sehr wertvoll - (auch wenn ich es da ein wenig übertrieben habe). Besonders in den Resource-Klasse scheint nicht alles ganz optimal gecoded zu sein. Eine Zeile die eine derartige Form annimmt:
`FridgesLOCAL.Fridge.ProductTypes.ProductType.Products pdsL = new FridgesLOCAL.Fridge.ProductTypes.ProductType.Products();` lässt ein schon ins zweifeln kommen. Man hätte wohl die JAXB-Klassen geschickter gestalten können. Oder auch derartige Abschnitte

```
// Lokale Fridgeliste nach passender id durchsuchen.
FridgesLOCAL.Fridge fL = new FridgesLOCAL.Fridge();
for (int i=0; i<fsL.getFridge().size(); i++) {
     if(fsL.getFridge().get(i).getId() == fridgeID){
         fL = fsL.getFridge().get(i); // Fridge herausnehmen
         break;
     }
}
```

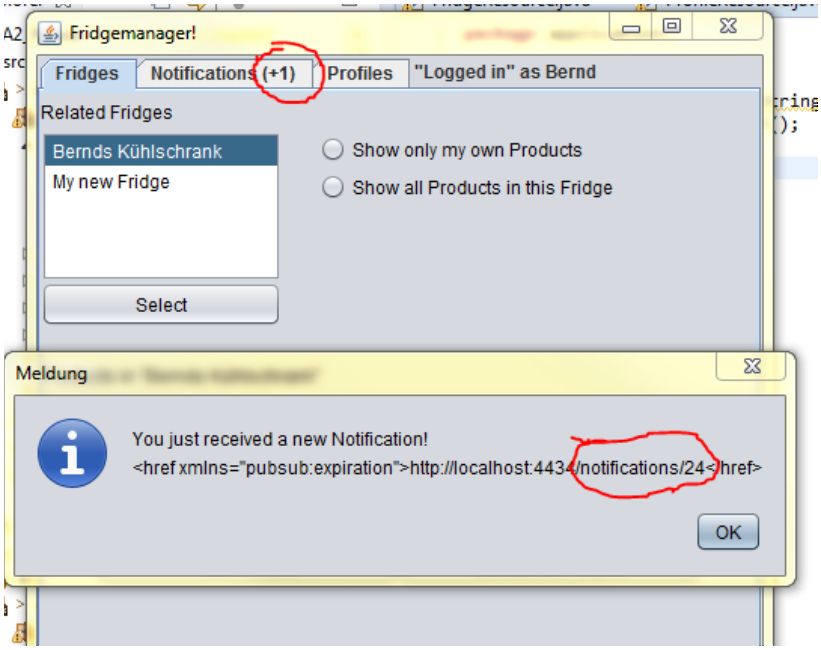
treten so in ähnlicher Form wiederholt auf. Eine Schnittstelle für alle Listen-Klassen die die Methoden getId(), size() und get() umfasst und einer Methode zum herausfischen eines Listenelements nach id, hätte hier Abhilfe leisten könne.
Aber ich wollte nicht noch mehr Zeit für die Datenbank bzw. den Zugriff auf die Datenbank (XML-Datenbank) investieren.
Von unserer Beispielapplikation möchte ich garnicht erst anfangen. Diese wurde nur sehr wenig in Bereichen wie Usability, Optik, etc abgewägt, aber ich denke das sollten wir uns für EIS aufsparen :)

Bevor ich hier noch lange reflektiere werde ich die restlichen Stunden nutzen um folgende Dinge zu implementieren:

- DELETE-Operationen, diese Fehlen noch für das neue Ressourcendesign, sollten aber relativ einfach realisierbar sein. (Oder doch nicht? Ich denke gerade an die Querverweise. So wie bei einem POST auf ein Produkt der Verweis im Kühlschrank hinzugefügt wird, sollte auch beim löschen dieses Produkts die Referenz im Kühlschrank entnommen werden. Und was passiert mit den Produkten und Kühlschränken eines Profils wenn es gelöscht wird? …)
- Beispielapplication:
    - Verbrauchen von Produkten (Wie beim konsumieren ein kleines Fenster- im Idealfall zeigt es die einzelnen Produkte mit ihrem Besitzer an
    - Löschen von Nachrichten
    - SubcribeToFenster buggt noch rum
    - Eigentlich noch millionen Dinge mehr...

Bevor ich diese letzten Punkte umsetze, möchte ich noch ganz kurz auf ein paar schon umgesetzten Ergebnisse eingehen:

Asynchrone Nachricht trifft ein: (Client hat experationNode abboniert!)



Das href-Tag ist der Payload aus dem Item.
Natürlich ist es wenig hilfreich dem Endanwender die Referenz ins Gesicht zu werfen. Es ging eher darum zu zeigen das der Client nun weiß, das eine Nachricht, und vorallem welche Nachricht, neu hinzugekommen ist (Der Client könnte diese in der Nachrichten-Listen hervorheben oder mit einem “New-Flag” versehen (ähnlich wie bei Mail-Anwendungen))

Die Implementierung für das publishen (serverseitg) einer solchen Nachricht sieht folgendermaßen aus: (aus der Klasse Notification Service)

```
// POST it
url = Server.url + "/notifications";
Server.wrs = Client.create().resource(url);
ClientResponse r = Server.wrs.type(MediaType.APPLICATION_XML).post(ClientResponse.class, n);
System.out.println(r.toString());
// Referenz publishen
Server.xmpp.pubItemInNode(XMPPData.expirationNodeID,
        "<href xmlns='pubsub:expiration'>"+r.getLocation()+"</href>");
```

direkt nach dem POST wird - und das sind Momente die einen Programmierer auf tiefste befriedigt - aus dem HTTP-Response, im Location-Header-Field, die Referenz auf die neu angelegte Ressource entnommen. (Btw. Fehlerbehandlung nach dem POST fehlt hier noch - ClientResponse auf Statuscode 201 prüfen)