Master Programming

Übung messenger-model

Diese Übung adressiert den Umgang mit Technologien zum Erstellen eines Datenmodells (kurz „Model“) für eine Web 2.0 basierte Facebook-artige Applikation. Das Datenmodell besteht aus einem relationalen Datenbankschema, persistenten Entitäten und Relationen, sowie REST-Services welche mittels Service-Methoden und Marshaling Zugriff auf diese Entitäten über Prozessgrenzen hinweg gewähren. In einem nachfolgenden Teilprojekt werden diese Service-Methoden dann für die Erstellung der Web 2.0 Applikation genutzt. Details zu den verwendeten Technologien finden sich in diesem Quellen:

- http://en.wikipedia.org/wiki/Bean_Validation
- http://en.wikipedia.org/wiki/Java_persistence_api
- http://en.wikipedia.org/wiki/Java_Architecture_for_XML_Binding
- http://en.wikipedia.org/wiki/Java_API_for_RESTful_Web_Services
- http://en.wikipedia.org/wiki/Basic_access_authentication
- http://en.wikipedia.org/wiki/EclipseLink
- http://en.wikipedia.org/wiki/Project_Jersey

1 Persistentes Modell

Ziel dieses Teilprojekts ist das Aufsetzen der Datenbank, sowie die Entwicklung von Entitätsklassen für den späteren Zugriff auf erstere, inklusive Bean-Validation, JPA-, JAX-RS und JAX-B Annotationen.

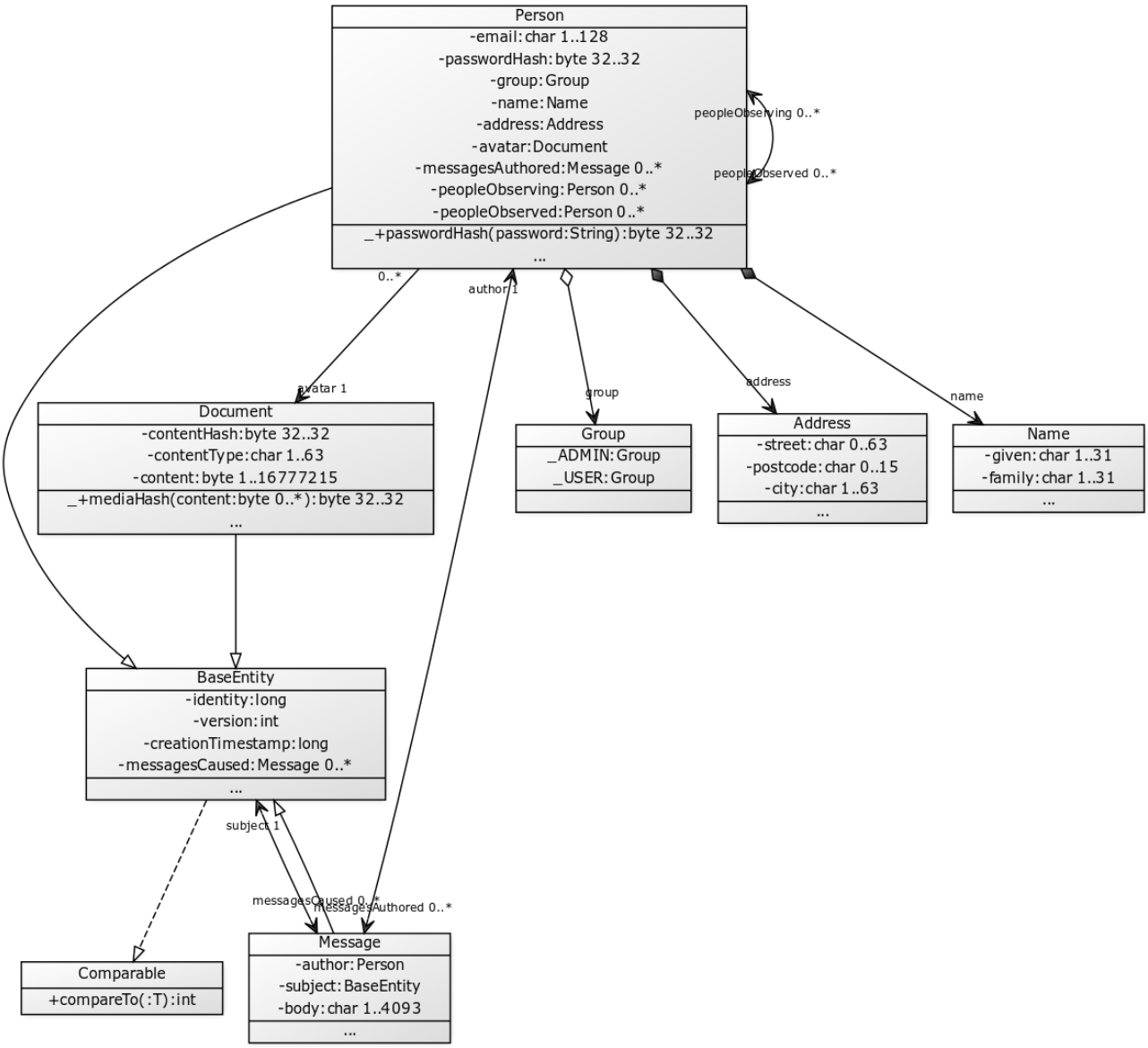
1.1 Aufsetzen des Projekts

Erzeugt ein neues Projekt **messenger-model**. Erzeugt in dessen „src“-Ordner die Pakete „`de.sb.messenger.persistence`“, „`de.sb.messenger.rest`“, sowie den Ordner „META-INF“. Übertragt aus dem entsprechenden Ordner auf dem „share“-Laufwerk (`//share/lehrende/Baumeister/Master Programming/code/messenger-model/META-INF`) alle Dateien nach META-INF.

Folgt den Anweisungen der dortigen Datei **`project-setup.txt`** um Euer Projekt und die Datenbank korrekt aufzusetzen (Compiler-Version, User-Libraries, persistence.xml, Einspielen von DDL&DML, usw.).

1.2 Entwicklung der Entitäten

Erzeugt für folgendes UML-Klassendiagramm eine passende Implementierung in Java:



Siehe http://en.wikipedia.org/wiki/Class_diagram und http://yuml.me/ für die in diesen UML-Klassendiagrammen verwendete Symbolik. Die mit „...“ gekennzeichneten Stellen stehen dabei für die typischen JavaBeans Getter- und Setter-Methoden, Konstruktoren, usw. Beachtet dass alle Relationsfelder als read-only interpretiert und daher nur mittels Konstruktorargument gesetzt werden sollen. Definitert zudem die künstlichen Getter-Methoden `Person#getSubjectReference()` und `Message#getAuthorReference()` um die Identität der jeweiligen Relationspartner abzufragen – wir benötigen diese später für das Marshaling.

Beachtet dass die Kardinalität (Mannigfaltigkeit) eines Typs 1..1 ist falls nicht explizit angegeben. Die typische Modellierung für multiple Zeichen ist `String`, die für multiple Bytes dagegen `byte[]`. **`java.lang.Comparable`** ist ein bereits bestehendes Interface, während **`Group`** am besten als statische innere enum in Person modelliert wird. **`BaseEntity`** legt für Entitäten eine natürliche Reihenfolge aufgrund ihrer Identität fest, i.e. diese ist für den gesamten

Klassenbaum eineindeutig.

Die Felder `passwordHash` und `contentHash` sollen SHA-256 Hashes des Dokument- bzw. Passwort-Inhalts (UTF-8 codiert) sein. Beachtet dass ein solcher Hash-Wert bei gut streuender Hash-Funktion aufgrund der großen Bit-Länge und des damit zusammenhängenden immensen Werteraums (2^{256} ~ 10^{77} mögliche Werte) als quasi-eineindeutige ID für einen Dokument-Inhalt bzw. ein Passwort anzusehen ist, da die Chance dass zwei unterschiedliche Inhalte den gleichen Hash-Wert erzeugen nahezu 0 ist. Damit kann ein solcher Hash die gleiche Rolle eines natürlichen Schlüssels übernehmen wie „alias“ bei Person. File-Sharing arbeitet letztlich genau nach diesem Prinzip, unabhängig davon ob P2P oder Cloud basiert, oder welche Hash-Funktion im einzelnen verwendet wird.

2 JPA-Mapping & JUnit-Regressionstest

Ziel dieses Teilprojekts ist die JAX-Validation und JPA-Annotation der Entitäten, sowie die Erstellung eines JUnit-Regressionstest zum Testen von Validierung und Datenbankanbindung.

2.1 Bean-Validation der Entitäten

Verwendet Annotationen um folgende Integritätsbedingungen zu definieren:

- Felder (außer *:1 Relationsfelder) die im Datenbankschema nicht null werden können, deren Typ diesen Wert aber zulässt, dürfen in validen Entitäten auch nicht null sein → **`@NotNull`**.
- Felder welche eine minimale und/oder maximal Kardinalität vorsehen dürfen diese in validen Entitäten nicht unter- bzw. überschreiten → **`@Size`**.
- In validen Entitäten dürfen Stückzahlen nicht negativ sein → **`@Min`**.
- Medien-Typen müssen aus zwei durch „/“ getrennten Teilen bestehen. Der vordere Teil darf ausschließlich aus kleingeschriebenen Zeichen (a-z) bestehen (mindestens einem), der hintere Teil dagegen aus kleingeschriebenen Zeichen (a-z), „.“, „+“ und „-“ (ebenfalls mindestens einem) → **`@Pattern`**.
- Email-Adressen müssen „@“, sowie davor und danach mindestens ein weiteres Zeichen beinhalten → **`@Pattern`**.
- Die Komposite einer validen Person (Name, Address, Contract) müssen ebenfalls valide sein → **`@Valid`**.

2.2 JPA-Mapping der Entitäten

Nutzt Annotationen aus Paket **`javax.persistence`** um die Klassen und Felder auf das Datenbank-Schema „broker“ abzubilden.

Annotiert dazu jede Entität mit **`@Entity`**, und jedes Komposit (Name, Address, Contact) mit **`@Embeddable`**. Annotiert zudem für jede Entität mittels **`@Table`** sowohl Schema als auch Tabellenname: Das Schema um die Möglichkeit zum Multi-Schema Betrieb zu erhalten, den Tabellennamen um zu verhindern dass JPA-Implementierungen wie EclipseLink oder TopLink diese in Großbuchstaben vom Klassennamen ableiten (war früher mit Oracle/DB und IBM DB/2 der Normalfall), nur um dann „praktischerweise“ unter Linux oder Mac die Tabellen nicht mehr zu finden …

Bildet die Vererbungshierarchie der Entitäten ab indem ihr **`BaseEntity`** mit **`@Inheritance`** nebst Inheritance-Type **`JOINED`**, sowie **`@DiscriminatorColumn`** zur Definition des Diskriminator-Feldes annotiert. Markiert zudem das Feld „identity“ mit **`@Id`** und **`@GeneratedValue`** als Primärschlüssel für den Klassenbaum. Die Identitäten der abgeleiteten Entitäten werden dagegen mittels **`@PrimaryKeyJoinColumn`** an der jeweiligen Subklasse annotiert. Dies bildet den Klassenbaum mittels einer „eine Tabelle pro konkreter oder abstrakter Klasse“-Strategie ab, in dem jede Entität eine automatisch generierte und über den gesamten Klassenbaum gesehen eindeutige Identität aufweist. Diese Art der Modellierung ist beim Einfügen und Ändern der Daten etwas langsamer als die Alternativen, dafür jedoch redundanzfrei, leicht pflegbar, und mächtiger weil sie polymorphe Queries gegen die Basis-Klasse(n) erlaubt.

Bildet alle *:1 und 1:1 Relationsfelder mittels **`@ManyToOne`** und **`@OneToOne`** ab. Setzt bei

`Person#avatar` zudem das fetch-Attribut auf `LAZY` (für Lazy-Initialisierung) sowie **`@JoinColumn`** zur Definition des zu verwendenden Fremdschlüssels. Bildet 1:* sowie *:* Relationen mittels **`@OneToMany`** und **`@ManyToMany`** ab, wobei diese typischerweise noch mittels mappedBy-Attribut einen Verweis auf ihr Gegenstück enthalten. Definiert die ***:* Relationen** zwischen Personen so dass die „people observed"-Seite die Relation definiert, während die „people observing"-Seite ihre Spiegelung darstellt.

Annotiert alle „normalen" Felder außer Identitäten und Komposite mittels **`@Column`**. Die **Komposite** Name, Address und Contact werden mit **`@Embedded`** annotiert, während das **Aggregat** Group neben **`@Column`** mit **`@Enumerated`** zu annotieren ist.

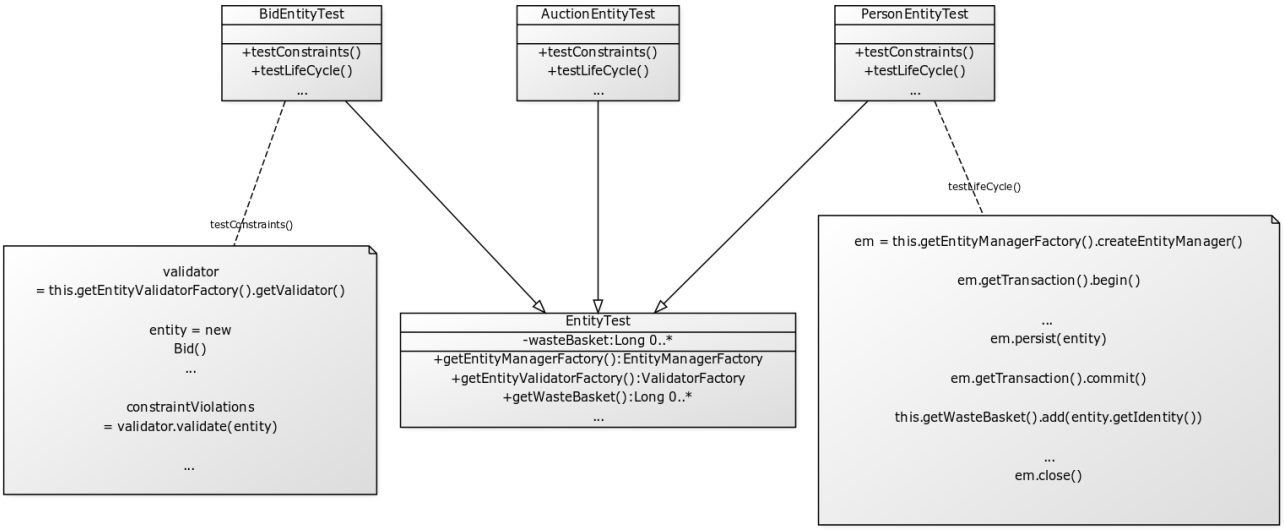
Damit LAZY-Initialisierung funktioniert muss zudem in jedem Laufzeit-Profil (Container-Applikation, JUnit-Tests) unter VM-Argumente der Eintrag „**`-javaagent:/<path>/eclipselink.jar`**" eingetragen sein, wobei <path> mit dem auf Eurem System gültigen Pfad zu ersetzen ist. Wenn dies vergessen wird, dann setzt EclipseLink von Lazy-Fetch auf Eager-Fetch zurück.

2.3 JUnit Regression-Test

Erstellt ein neues Projekt „broker-test" mit denselben Paketen und Ordnern wie in „broker-model". Kopiert aus den entsprechenden Ordnern auf dem „share"-Laufwerk die Dateien in die Pakete bzw. „META-INF", und folgt den Anweisungen der dortigen `project-setup.txt` Datei um das Projekt korrekt aufzusetzen (Projektreferenz, JUnit-Library, User-Libraries, usw.).

Die Test-Methoden sollen so entworfen sein dass sie auf einem definierten (z.B. leeren) Datenbestand basieren, und diesen nach Test-Ausführung stets wieder herstellen. Dazu ist es notwendig dass sich die Test-Klassen die Identitäten persistierter Entitäten merken, und diese nach Ausführung jeder Test-Methode verlässlich löschen, unabhängig davon wie der Test ausging, und ohne bereits vorher bestehende Entitäten zu löschen.

Verwendet daher die vorhandene Klasse **`EntityTest`** nach folgendem Muster:



`EntityTest` garantiert (mittels der JUnit-Annotationen `@BeforeClass` und `@AfterClass`) dass vor jedem Testlauf in einer Subklasse eine **`EntityManagerFactory`** und eine **`ValidatorFactory`** erzeugt wird, welche nach dem Testlauf wieder geschlossen werden. Ein Testlauf kann dabei vom Ausführen einer einzelnen Test-Methode bis zum Ausführen aller Testmethoden eines Test-Projekts reichen. Zudem wird (mittels der JUnit-Annotation `@After`) garantiert dass Entitäten mit im Papierkorb hinterlegten Identitäten nach Ausführung einer Testmethode gelöscht werden.

Erzeugt daher pro Entität eine Test-Klasse als Subklasse von `EntityTest`, und definiert dort mindestens die beiden im Klassenschema erwähnten Test-Methoden. Diese müssen als solche mittels der JUnit-Annotation **`@Test`** markiert werden. Beachtet dabei dass JUnit Test-Methoden öffentlich sichtbar, sowie Parameter- und Resultat-los sein müssen.

Die Methode `testConstraints()` soll nun pro Entität alle annotierten Integritätsbedingungen methodisch überprüfen. Dazu sollen die Entitäten zuerst Konstruiert, mit legalen Feldwerten befüllt, und dann positiv validiert werden. Daneben soll die Entität validiert werden nachdem jedes Feld mit (wenn passend) grenzwertig illegalen sowie grenzwertig legalen Werten befüllt wurde. Überprüft nach jeder Validierung mittels **`org.junit.Assert.assertEquals()`** ob die beim Validieren erzeugte ConstraintViolation-Menge eine Größe besitzt die der Erwartungshaltung beim Test entspricht.

Die Methode `testLifeCycle()` soll pro Entität das JPA-Mapping überprüfen, i.e. Entitäten erzeugen, verändern und löschen und die Erwartungshaltung dabei mittels Aufruf von statischen Methoden von **`org.junit.Assert`** verifizieren. Besorgt Euch dort zuallererst von der Superklasse eine `EntityManagerFactory`, und erzeugt damit einen neuen `EntityManager` für den Testablauf. Dieser ist am Schluss stets zu schließen, egal ob die Testmethode normal oder mittels Exception verlassen wird.

Verwendet danach das JPA EntityManager-API: Startet für jeden Test-Abschnitt eine Transaktion mittels „`entityManager.getTransaction.begin()`“, welche am Ende des Abschnitts unabhängig vom Testverlauf mittels `commit()` oder `rollback()` zu beenden ist. Beachtet dass die Identitäten neuer Entitäten bei **EclipseLink** erst nach erfolgreichem Persistieren UND Commit ihren endgültigen Wert annehmen – ihr könnt sie also auch erst danach beim Papierkorb zum automatisierten Löschen anmelden. Erzeugt neue Entitäten mittels Konstruktion, fügt diese mittels „`entityManager#persist(entity)`“ in die Datenbank

ein. Verwendet für Identity-Queries „`entityManager#find()`“ und „`entityManager#getReference()`“.

Fügt zu einer Entität relational assoziierte Entitäten in die Datenbank ein (Registrierung zum Löschen nicht vergessen), und prüft ob diese nach „`entityManager#refresh(entity)`“ in den „mappedBy“-Relationsmengen der Entitäten sichtbar werden. Überprüft zudem ob persist/flush/commit fehlende Relationen sowie die Verletzung von Integritätsbedingungen erkennen welche in der Methode `testConstraints()` mangels relational assoziierter Entitäten nicht überprüft werden konnten.

Der **Start einer Testmethode** erfolgt in **Eclipse** durch Markieren der Test-Methode und Nachfolgende Auswahl von *“run As->JUnit Test“* im Kontext-Menü. Durch Auswahl von *„debug As->JUnit Test“* wird die Testmethode im Debug-Modus gestartet, wodurch an Unterbrechungspunkten angehalten und Debugging betrieben werden kann. Wenn ihr dabei das Zurücksetzen des Zugriffs-Modus bei `Person#avatar` von LAZY auf EAGER vermeiden wollt, setzt in jedem generierten Laufzeit-Profil für die Testfälle unter VM-Argumente den Eintrag „**`-javaagent:/<path>/eclipselink.jar`**“, wobei <path> mit dem auf Eurem System gültigen Pfad zu ersetzen ist.

Nach dem gleichen Prinzip ist der Start von multiplen Testmethoden (Methoden markieren), aller Testmethoden einer bzw. mehrerer Klassen (Klassen markieren), aller Testmethoden aller Klassen eines oder mehrerer Pakete (Pakete markieren), oder aller Testmethoden aller Klassen aller Pakete eines Projekts (Projekt markieren) möglich.

3 Entwicklung der REST-Services

Ziel dieses Teilprojekts ist die **JAX-B** Annotation der Entitäten und deren Komposite für Marshaling von/nach **XML** und/oder **JSON**, sowie die Erstellung von **REST-Services** welche diese Entitäten nutzen.

3.1 JAX-B Annotation der Entitäten

Verwendet **JAX-B Annotationen** um die Entitäten und deren Felder von/nach JSON und XML marshalen zu können. Dabei ist zu beachten dass diese Annotationen einen statischen Informationsumfang festlegen, i.e. das was dort als sichtbare Information festgelegt ist, wird später für Instanz der betreffenden Entität übertragen, und zwar für jedwede Service-Methode. Daher sollte der hier festgelegte Informationsumfang minimal sein, und primär die Entität selbst betreffen.

Annotiert nun die Klasse **`BaseEntity`** mittels **`@XmlAccessorType`**`(XmlAccessType.`***`NONE`***`)`, **`@XmlType`** und **`@XmlSeeAlso`**. Erstere sorgt dafür dass alle zu marshalenden Elemente explizit angegeben werden müssen, was einige Restriktionen umschifft und am flexibelsten ist. Zweitere Annotation sorgt dafür dass alle Subklassen von `BaseEntity` in Struktur-Beschreibungen (wie WSDL, WADL) mit einem Namen vertreten werden, statt direkt durch ihre Struktur. Letztere Annotation ist notwendig weil es in Java notorisch schwierig und unzuverlässig ist herauszufinden welche Subklassen für eine Klasse definiert sind; da Service-Methoden jedoch polymorphe Resultate deklarieren können, muss dem Marshaler bei Superklassen und Interface-Typen bekannt gemacht werden welche Subklassen zur Laufzeit auftreten können.

Annotiert folgende Konstrukte mittels **`@XmlElement`**:

- die **Felder** aller Entitäten und Komposite, außer sie sollen beim Marshaling nicht abgebildet werden (wie `Person#passwordHash` und `Document#content`), oder sie stellen Relationsfelder dar
- in der Klasse `Person` zudem die Felder für die Komposite Name, Adresse und Kontakt
- für die *:1 Relationen `Message#subject` und `Message#author` die Identität der relational assoziierten Entität (über getter getSubjectReference() und getAuthorReference()

Beachtet: Wird ein Feld mittels **`@XmlElement,`** **`@XmlAttribute`** oder **`@XmlID`** annotiert, dann greift der Marshaler zum Lesen & Schreiben direkt mittels Java Reflection API auf das Feld zu. Wird dagegen eine Getter-Methode annotiert, dann greift der Marshaler zum Lesen auf diese zu, und zum Schreiben auf die zugehörige Setter-Methode; existiert eine solche nicht, wird der Wert nicht gesetzt. Daher:

- für reale Properties immer Feld annotieren
- für virtuelle Properties (typischerweise read-only) immer Getter-Methode annotieren

3.2 REST-Services & JAX-RS Annotation

Es existiert bereits eine REST-Service Klasse `EntityService` im Paket `de.sb.messenger.rest`. Dieser bietet Service-Methoden für alle Arten von Entitäten, ist also polymorph:

- **`GET /entities/{identity}`**: Returns the entity matching the given identity.
- **`DELETE /entities/{identity}`**: Deletes the entity matching the given identity.
- **`GET /entities/{identity}/messagesCaused`**: Returns the messages caused by the given identity.

Erstellt zusätzlich im Package `de.sb.messenger.rest` zwei weitere Klassen **`PersonService`** und **`AuctionService`**. Erweitert diese mittels geeigneten Methoden und **JAX-RS Annotationen**. Dabei werden Entitäten stets im Request/Response-Body übertragen, IDs als Path-Parameter, und weitere Information als Header-Felder, Query-Parameter, usw. Erzeugt und speichert dafür pro Service-Klasse eine EntityManagerFactory in einer statischen Variable, und implementiert so je eine Service-Methode für folgende HTTP-Zugriffe:

- **`GET /people`**: Returns the people matching the given criteria, with missing parameters identifying omitted criteria.
- **`PUT /people`**: Creates a new person if the given template's identity is zero, or otherwise updates the corresponding person with the given *template* data. Optionally, a new password may be set using the header field “Set-Password”. Returns the affected person's identity
- **`GET /people/requester`**: Returns the authenticated requester, which is useful for login operations.
- **`GET /people/{identity}`**: Returns the person matching the given identity.
- **`GET /people/{identity}/messagesAuthored`**: Returns the messages authored by the person matching the given identity.
- **`GET /people/{identity}/peopleObserving`**: Returns the people observing the person matching the given identity.
- **`GET /people/{identity}/peopleObserved`**: Returns the people observed by the person matching the given identity.
- **`PUT /people/{identity}/peopleObserved`**: Updates the person matching the given identity to monitor the people matching the form-supplied collection of person identities. Hint: Make sure all people whose Mirror-Relations change due to this operation are evicted from the 2nd-level cache!
- **`GET /people/{identity}/avatar`**: Returns the avatar content of the person matching the given identity (plus it's content type as part of the HTTP response header). Hint: Use `@Produces(WILDCARD)` to declare production of an a priori unknown media type, and return an instance of **Result** that contains both the document's media type and it's content.

- **`PUT /people/{identity}/avatar`**: Updates the person's avatar (owner only), with the document content being passed as the HTTP request body, and the media type passed as Header-Field “Content-type”. If the given content is empty, the person's avatar is set to the default document (identity=1). Otherwise, if a document matching the media hash of the given content already exists, then this document becomes the person's avatar. Otherwise, the given content and content-type is used to create a new document, and it is registered as the person's avatar. Hint: Use `@Consumes(WILDCARD)` to declare consumption of an a priori unknown media type.
- **`PUT /messages`**: Creates a new message with the requesting person as author, using the given form fields “content” and “subjectReference”, and returning the message's identity. Hint: Make sure that the author and the (polymorphic) subject are evicted from 2nd-level cache once the message is stored, because their mirror relations have to change.
- **`GET /messages/{identity}`**: Returns the message matching the given identity.
- **`GET /messages/{identity}/author`**: Returns the author of the message matching the given identity.
- **`GET /messages/{identity}/subject`**: Returns the (polymorphic) subject of the message matching the given identity.

Sind dabei **Criteria-Queries** gefordert, so definiert als Query-Parameter ein Suchkriterium pro textuellem Entity-Feld, und zwei für numerische Felder (für >= und <= Vergleich). Definiert zudem einen JP-QL Query nach folgendem Muster (statt „=“ kann für mehr Flexibilität auch „like“ verwendet werden, ist aber deutlich teurer in der Ausführung):

```
select x from X as x where
(:lowerNumber is null or x.number >= :lowerNumber) and
(:upperNumber is null or x.number <= :upperNumber) and
(:text is null or x.text = :text) ...
```

Wird durch einen Service eine Relation modifiziert, dann sind nach Speicherung der Änderungen zudem alle Entitäten aus dem **2nd-Level Cache** zu entfernen deren „mappedBy“-Relationsmengen sich dadurch ändern; diese Spiegel-Mengen werden weder im 1st-Level Cache noch im 2nd-Level Cache automatisch verwaltet:

```
cache = entityManager.getEntityManagerFactory().getCache();
cache.evict(entity.getClass(), entity.getIdentity());
```

Beachtet: Streng genommen müssten wir dieselbe **Cache-Invalidierung** auch mit dem **1st-Level Cache** durchführen (mittels `entityManager#refresh(entity)`); da wir diesen in Web-Services nach Commit jedoch gleich wieder schließen ist es in diesem Fall einfacher & performanter darauf zu verzichten.

3.3 Sanity-Check mittels Web-Browser

Startet die vorhandenen Klasse **`de.sb.messenger.rest.ServiceTest`** in **Eclipse** indem ihr sie markiert, um im Kontext-Menü (Rechtsklick) „Run As → Java Application“ auswählt. Alternativ könnt ihr mittels „Debug As“ die Klasse im Debug-Modus starten, was dann das Anhalten und Debuggen an gesetzten Unterbrechungspunkten erlaubt.

Während der Ausführung könnt ihr mittels eines Web-Browsers in der Folge REST Service-Methoden des eingebetteten HTTP Containers über Port 8001 aufrufen, z.B. mittels **`http://localhost:8001/services/people`**:

Damit kann die Funktion GET-basierter Service-Methoden von Web-Browser aus leicht durch Eingabe ihrer URI in die Statuszeile auf Sicht geprüft werden. Beachtet dabei dass ihr in Firefox mittels „Ctrl-Shift-Q“ bzw. „F10 → Tools → Web Developer → Network“ detaillierte Informationen bezüglich HTTP Codes, Request/Response Header, Responses usw. erhaltet. Durch Editieren der Variable „`network.http.accept.default`“ in „`about:config`“ könnt ihr zudem die Präferenz des Firefox-Browsers für XML oder JSON einstellen.

4 Erweiterung der REST-Services

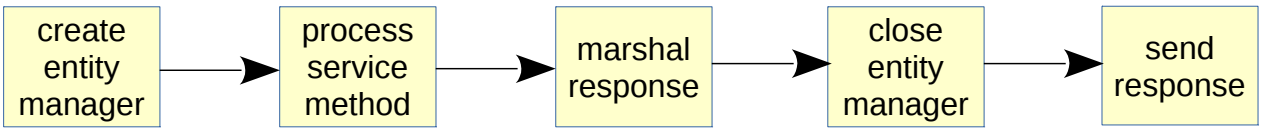
Ziel dieses Teilprojekts ist die Behebung der Unzulänglichkeiten in den bisher erstellten REST-Services: Lebenszyklus-Management, Fehlerbehandlung, HTTP Authentifizierung, und flexibleres Marshaling.

4.1 Lebenszyklus-Management

Der bisherige Ansatz pro Service eine `EntityManagerFactory` zu erzeugen hat aus Architektur-Sicht einige entscheidende Nachteile:

- Multiple Instanzen von **`EntityManagerFactory`** implizieren multiple **2nd-Level Caches**. Dies ist zum einen sehr ineffektiv, und zum anderen müssten wir so für eine korrekte Lösung auch noch Cache-Synchronisation betreiben!
- Jede Service-Methode erzeugt einen eigenen **`EntityManager`**, und entscheidet selbst ob/wie sie **ACID-Transaktionen** nutzt oder nicht. Dies hat ebenfalls Nachteile:
  - Der `EntityManager` ist beim Marshalen des Ergebnisses bereits wieder geschlossen, was sich fatal auswirkt sobald der Marshaler vorher ungeladene Information benötigt.
  - Entwickler übersehen gerne dass der `EntityManager` nach Gebrauch in jedem Fall wieder geschlossen werden muss, speziell auch im Fehlerfall. Nach Exceptions nicht geschlossene EntityManager führen dann regelmäßig zu schwer lokalisierbaren Resource-Leaks.
  - Transaktionen werden maximal zum Schreiben von Information verwendet, und das obwohl sie zum einen günstig erzeugbar sind, und zum anderen auch beim Lesen eine isolierende Wirkung gegenüber parallel ablaufenden Transaktionen erzeugen.

Daher ist es oft vorteilhaft den Lebenszyklus der Entity-Manager nicht in jeder Service-Methode programmatisch zu steuern, sondern an den Lebenszyklus der HTTP Anfragen zu koppeln. Ein ähnliches Entwurfsmuster wird in *Enterprise Java Beans* z.B. bei „persistence unit injection“ sowie „container managed transactions“ umgesetzt:



Dazu ist in *ServiceTest* (broker-test) und *ApplicationContainer* (broker-web) ein Lifecycle-Provider konfiguriert der folgende Aufgaben erfüllt:

- Er erzeugt eine gemeinsam nutzbare **`EntityManagerFactory`**, und vermeidet damit den Zusatzaufwand multiple 2nd-level Caches koordinieren zu müssen, wenigstens solange die Applikation nicht auf multiple Prozesse verteilt wird.
- Er erzeugt zudem zu Beginn der Bearbeitung jeder HTTP-Anfrage eine frische Instanz von **`EntityManager`**, und bindet diese an den **aktuellen Thread**. Diese Instanz kann innerhalb der Service-Methoden mittels der statischen Methode **`RestJpaLifecycleProvider#entityManager("broker")`** abgerufen werden. Die Instanz wird am Ende der Anfrage-Bearbeitung, also nach (sic!) dem Marshaling, automatisch geschlossen.
- Er startet zudem direkt nach Erzeugung des Entity-Managers eine aktive Transaktion, und rollt vor dem Schließen des Entity-Managers eine noch aktive Transaktion zurück.

Die Service-Methoden sind zur Nutzung dieses Lifecycle-Providers so umzubauen dass

sie sich zu Beginn den aktiven `EntityManager` von diesem besorgen, und diese am Ende NICHT mehr schließen. Des Weiteren müssen sie bei jedem **commit** nachfolgend sofort eine neue Transaktion starten, auch im Fehlerfall, und selbst wenn sie selbst keine weiteren Datenbankoperationen durchführen → try-finally-Block. Denn nach der Ausführung einer Service-Methode wird der Marshaler aktiv, und auch dieser profitiert von der isolierenden Wirkung einer aktiven Transaktion wenn er implizit relational assoziierte Daten nachlädt.

4.2 Fehlerbehandlung

Die bisherigen Services erzeugen stets HTTP Code 500 wenn während der Abarbeitung ein Fehler auftritt. Das ist sowohl zur Wartung der Services ungenügend, als auch für Web 2.0 Clients zu wenig Information.

Passt daher alle REST-Service-Methoden so an dass sie geeignete HTTP Return Codes generieren. Dazu gibt es im Prinzip zwei Vorgehensweisen:

1. Im Fehlerfall potentiell Abfangen von Exceptions, und (erneutes) Werfen als Instanzen von Subklassen von **`WebApplicationException`**. Diese werden nach dem Marshaling durch einen vorkonfigurtierten Exception-Mapper (**`RestResponseCodeProvider`**) auf ihre assoziierten HTTP Codes abgebildet; alle nicht abgefangenen „normalen“ Exception-Typen werden dagegen auf HTTP 500 abgebildet.

2. Deklaration von **`Response`** als Rückgabetyp der Service-Methoden, Abfangen aller Exceptions, und direkte Rückgabe geeigneter HTTP Codes im Normalfall wie im Fehlerfall.

Setzt bei den bestehenden Services die erste Vorgehensweise um, indem ihr alle Fehlerfälle abfangt die auf Illegale Parameter, verletzte Constraints, oder sonstige Client-Fehler hindeuten, und in diesen Fällen eine Instanz von **`ClientErrorException`** mit passenden HTTP-Fehlercode werft. Alle anderen Arten von Fehlern sollen dagegen nicht abgebildet werden, womit sich für die Services folgendes Verhalten ergibt:

- **Code 200 / keine Exception & Resultat deklariert:** falls die Verarbeitung fehlerfrei verlief und potentiell etwas zurück gegeben wird.
- **Code 204 / keine Exception & void deklariert:** falls die Verarbeitung fehlerfrei verlief aber generell nichts zurück gegeben wird.
- **Code 400 / ClientErrorException(400)**: falls ein Parameter oder Header illegal geformt ist oder einen illegalen Wert (inklusive null) aufweist.
- **Code 404 / ClientErrorException(404)**: falls keine passende Entität mit einer im Pfad oder im Body gegebenen Identität existiert.
- **Code 409 / ClientErrorException(409)**: falls ein Commit wegen Verletzung einer Datenbank-Integritätsbedingung zurück gerollt wurde, i.e. bei `RollbackException`
- **Code 500 / normale Exceptions**: falls ein anderes serverseitiges Problem vorliegt.

Fangt zudem illegale Parameter der Service-Methoden durch geeignete Annotationen der Methoden-Parameter ab. Falls eine komplette Entität übergeben wird kann dies durch Verwendung von `@Valid` geschehen, bei skalaren Werten bieten sich dagegen die Feldannotationen `@NotNull`, `@Min`, `@Max`, `@Size`, `@Pattern` usw. an. Invalide Parameter werden dann automatisch auf HTTP Code 400 bzw. 404 abgebildet.

Probiert wie in Aufgabe 3.3 beschrieben aus wie sich diese Änderungen auswirken.

4.3 HTTP Basic Authentifizierung

Benennt die Klasse `AuthenticatorSkeleton` nach `Authenticator` um. Implementiert anschließend die bestehende Methode **`authenticate(Basic)`** passend zum Methodenkommentar.

Fpgt jeder Servicemethode nun einen zusätzlichen String-Parameter „authentication“ hinzu indem ihr den Inhalt des Header-Feldes „Authorization“ (i.e. „authentication“) mittels der JAX-RS Annotation **`@HeaderParam`** übernehmt. Führt die Authentifikation danach folgendermaßen durch:

```
final Person requestor = Authenticator.authenticate(
     RestCredentials.newBasicInstance(authentication)
);
```

Beachtet dass sich durch die Authentifizierung zusätzliche HTTP Codes pro Methode ergeben:

- **Code 401 / ClientErrorException(400, “Basic“)**: Aufforderung zum wiederholten Versand der Anfrage mit gültiger HTTP Basic Authentifizierung – wird bereits durch Authenticator#`authenticate()` geworfen.
- **Code 403 / ClientErrorException(403)**: Der Aufrufende wurde Authentifiziert, jedoch ist dieser nicht autorisiert die Operation durchzuführen. Ein Neuversand der Anfrage ist daher sinnlos – muss bei Bedarf vom Service geworfen werden.

Durch den erweiterten Kontext (die Person-Entität des Aufrufenden aka der „requester“) werden zudem bei den REST-Services funktionelle Erweiterungen sinnvoll:

- **`PUT /people`**: Requesters that are not part of group ADMIN are both forbidden to alter other people, and to set their own group to ADMIN.
- **`PUT /auctions`**: Additionally constrain that an auction may only be altered if the requester is the same as the auction's seller.
- **`GET /people/{identity}/bids`**: Additionally returns bids on open auctions if the requester is the person targeted.
- **`GET /people/requester`** (new): Returns the authenticated requester, which is useful for login operations.
- **`GET /auctions/{identity}/bid`** (new): Returns the requesters bid for the given auction, or null if none exists.
- **`PUT /auctions/{identity}/bid`** (new): Creates or modifies the requesters bid for the given auction, depending on the requester and the price (in cent) within the given request body. If the price is zero, then the requesters bid is removed instead.

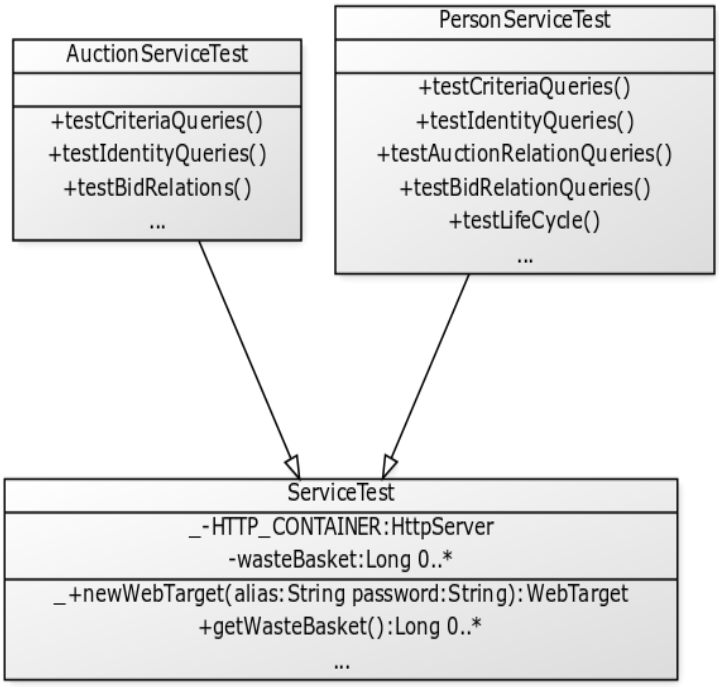
Probiert wie in Aufgabe 3.3 beschrieben aus wie sich diese Änderungen auswirken. Ihr könnt dazu die Default-Benutzer „ines“ (Passwort „ines“) und „sascha“ (Passwort „sascha“) verwenden, oder Euch eigene in der Datenbank anlegen.

5 JUnit Regression-Test der REST-Services

Die Test-Methoden sollen so entworfen sein dass sie auf einem definierten (z.B. leeren) Datenbestand basieren, und diesen nach Test-Ausführung stets wieder herstellen. Dazu ist es wieder notwendig dass sich die Test-Klassen die Identitäten persistierter Entitäten merken, und diese nach Ausführung jeder Test-Methode verlässlich löschen, unabhängig davon wie der Test ausging, und ohne bereits vorher bestehende Entitäten zu löschen. Verwendet daher die vorhandene Klasse **`ServiceTest`** nach folgendem Muster:

`ServiceTest` garantiert (mittels der JUnit-Annotationen `@BeforeClass` und `@AfterClass`) dass vor jedem Testlauf ein eingebetteter HTTP-Container gestartet wird welcher Eure REST-Services zur Verfügung stellt; ihr habt diesen bereits (als Java-Applikation gestartet) für die Sanity-Checks genutzt.

Zudem wird (mittels der JUnit-Annotation `@After`) garantiert dass Entitäten mit im Papierkorb hinterlegten Identitäten nach Ausführung einer Testmethode gelöscht werden – dazu wird die vorhandene Service-Methode **`DELETE /entities/{identity}`** genutzt.



Erzeugt daher Eure Test-Klassen als Subklasse von `ServiceTest`, und definiert dort mindestens die im Klassenschema erwähnten Test-Methoden. Diese nutzen das **JAX-RS Client-API** um Service-Aufrufe Schritt für Schritt aufzubauen und dann durchzuführen. Dies wird durch die statische Methode `newWebTarget(alias, password)` erleichtert welche ein Basis-Ziel mit HTTP Basic-Authentifikation für einen gegebenen Benutzer erzeugt. Dies dient in der Folge als Bootstrap um Anfragen mit erweiterten Pfaden, Zugriffsmethode, Parametern, usw. auszustatten, und schließlich zu versenden.

Reichert beide Testklassen nun mit Test-Methoden an um das Verhalten Eurer REST-Services ausführlich zu testen, und zwar sowohl für JSON, als auch für XML Marshaling; die Services unterstützen ja beides. Dies umfasst neben dem erfolgreichen Aufruf der Service-Methoden zudem Aufrufe die die dokumentierten Fehler-Codes mittels fehlerhafter Authentifizierung, illegalen Parametern, usw. provozieren. Dies stellt sicher dass sich die Services und ihre Dokumentation nicht auseinander entwickeln.

Im Einzelfall kann es sein dass sich die Ergebnisse einzelner Service-Methoden wegen Problemen beim Unmarshaling von XML nicht mittels des JAX-RS Client APIs abrufen lassen, und das obwohl die Methoden im Web-Browser aufrufbar sind. Diese Probleme hängen mit dem Entwicklungsstand des XML-Marshalers zusammen, testet daher in diesem Fall nur das JSON-Marshaling.

Ziel ist dass die Testfälle – soweit zeitlich und funktionell möglich - das komplette Spektrum an möglichen Service-Aufrufen und Fehlerfällen untersuchen. Dies garantiert dass die REST-Services fehlerfrei und konsistent reagieren bevor wir zur Entwicklung der Web 2.0 Applikation in JavaScript schreiten, denn dort sind solche Fehler u.A. aufgrund der typlosen Natur der Programmiersprache deutlich unangenehmer. Zudem erlauben die Testfälle eine sorgfältige Untersuchung ob Bug-Fixes oder Änderungen im Service- oder Model-Code Teile des APIs brechen, was entweder (teure) Änderungen in der Web-2.0 Anwendung, oder aber kompatiblere Änderungen an den Services oder im Model bewirkt.