Arbeitsblatt JPA

Ziele
Ziel dieses Arbeitsblattes ist es, einzelne Aspekte von JPA genauer kennen zu lernen, zu verstehen oder zu vertiefen. Da wir letzte Woche fast keine Zeit hatten an diesen Aufgaben zu arbeiten habe ich mir erlaubt, die Fragen von letzter Woche auf diesem Arbeitsblatt zu wiederholen. Falls Sie diese bereits gelöst haben, dann können Sie direkt mit Aufgabe 5 starten.

Ausgangslage
Im Gitlab-Projekt https://gitlab.fhnw.ch/eaf/hs21/03 steht das Projekt lab-jpa-03 zur Verfügung welches für diese Aufgaben verwendet werden soll. Importieren Sie dieses Projekt in ihre IDE.

Wie im Movie-Rental Projekt verwenden wir auch in diesem Projekt Spring Boot um den EntityManager und die Transaktionen zu erzeugen. Bei den einzelnen Testprogrammen (die jedoch nicht als JUnit-Tests realisiert sind sondern als einfache Java-Programme mit einer main-Methode) wird mit der `@EntityScan`-Annotation jeweils angegeben, in welchem Paket die Modellklassen definiert sind. Im Gegensatz zum Projekt von letzter Woche sind die Modellklassen in den jeweiligen Paketen enthalten.

Jedes Testprogramm hat folgenden Rahmen:

```
@SpringBootApplication
@EntityScan(basePackageClasses=TestX.class)
public class TestX implements CommandLineRunner {

	@PersistenceContext
	EntityManager em;

	public static void main(String[] args) {
		new SpringApplicationBuilder(TestX.class).run(args);
	}

	@Override
	@Transactional
	public void run(String... args) throws Exception {
	}
}
```

Die Properties in der Datei `application.properties` sind so vorkonfiguriert, dass einfach von einer in-memory auf eine lokale Datenbank gewechselt werden kann (die Kommentarmarkierungen müssen einfach umgesetzt werden). Die in-memory Datenbank hat dabei die URL `jdbc:h2:mem:lab-jpa-db`.

Für Task 7 wechseln wir auf eine lokale Datenbank.

```
spring.datasource.url=jdbc:h2:mem:lab-jpa-db
#spring.datasource.url=jdbc:h2:file:./build/lab-jpa-db
#spring.datasource.url=jdbc:h2:tcp://localhost/~/lab-jpa-db
```

Beachten Sie, dass jeweils nur ein Prozess auf eine lokale Datenbank zugreifen kann (falls mehrere Prozesse gleichzeitig auf die Datenbank zugreifen müssen so muss diese als separater Prozess gestartet werden).

Spring-Boot ist zudem so konfiguriert, dass jeweils auch ein Webserver gestartet wird damit die Datenbank mit der h2-Konsole betrachtet werden kann. Die Konsequenz ist, dass ein gestartetes Programm jeweils weiterläuft bis es explizit gestoppt wird, und ein zweiter Webserver kann unter demselben Port nicht geöffnet werden. Einige der Programme enden daher explizit mit der Anweisung `System.exit(0)`.

Die Nummern der Testprogramme (`Test1`, `Test2`, …) beziehen sich jeweils auf die folgenden Aufgaben.

Aufgaben:

1) **Unifizierung im Persistenzkontext**
Im Programm `Test1` wird der Kunde mit ID=1 zweimal mit der Methode `find` des Entity-Managers geladen:

```
Customer c1 = em.find(Customer.class, 1);
Customer c2 = em.find(Customer.class, 1);
```

Vergleichen Sie danach diese beiden Objekte mit einem Referenzvergleich (==). Sind diese beiden Objekte identisch oder nicht?

Was für ein Resultat erhalten Sie beim Referenzvergleich, wenn Sie zwischen den beiden `em.find`-Aufrufen den Persistenzkontext leeren?

```
Customer c1 = em.find(Customer.class, 1);
em.clear();
Customer c2 = em.find(Customer.class, 1);
```

Was für ein Resultat erhalten Sie beim Referenzvergleich, wenn Sie nach dem laden der Entität c1 diese mit `em.detach(c1)` explizit aus dem Persistenzkontext entfernen?

Wiederholen Sie dieses Experiment mit zwei unterschiedlichen Entity-Managern:

```
Customer c1 = emf.createEntityManager().find(Customer.class, 1);
Customer c2 = emf.createEntityManager().find(Customer.class, 1);
```

In diesem Beispiel haben wir uns von Spring eine Instanz der Klasse `EntityManagerFactory` einstecken lassen, über die wir dann `EntityManager`-Instanzen erzeugen können. Die Annotation lautet `@PersistenceUnit`.

```
@PersistenceUnit
EntityManagerFactory emf;
```

2) **OneToOne Association**
Zwischen den beiden Entitäten `Customer` und `Address` ist eine „bidirektionale“ 1:1 Assoziation definiert, d.h. beide Entitäten verweisen aufeinander, aber das `mappedBy` Attribut fehlt auf beiden Seiten:

```
@Entity
public class Customer {

    @OneToOne
    private Address address;

    …
}
```

```
@Entity
public class Address {

    @OneToOne
    private Customer customer;

    …
}
```

a) Wie sieht das generierte DB Schema aus? In welcher Tabelle ist der Fremdschlüssel definiert?

b) Was für ein Objektmodell haben Sie damit abgebildet?

3) **Bidirectional OneToOne Association**
In diesem Beispiel ist die Klasse `Address` nun so definiert, dass das attribut `customer` als inverses Attribut des Feldes `Customer.address` interpretiert wird.

```
@Entity
public class Address {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String street;
    private String city;

    @OneToOne(mappedBy = "address")
    private Customer customer;
```

Starten Sie nun das Testprogramm `Test3` und betrachten Sie das generierte Schema. Was hat sich geändert? Mit dem Programm Test3 wurde folgender Code ausgeführt:

```
Customer c = new Customer("Gosling", 55);
Address a = new Address("Infinite Loop 1", "Cupertino");
c.setAddress(a);

em.persist(a);
em.persist(c);
```

Fragen:

a) Ist die Reihenfolge der beiden Aufrufe `em.persist(a)` und `em.persist(c)` wichtig?

b) Ist der Aufruf von `em.persist(a)` nötig?

4) **Unidirectional OneToMany Association**
In der Klasse `Customer` ist eine **unidirektionale 1:n** Beziehung (`OneToMany`) zur Klasse Order definiert.

```
@OneToMany
private List<Order> orders = new ArrayList<>();
```

Damit kann ein Kunde mehrere Bestellungen haben. Mit der Definition der Assoziation wurde auch eine Methode `addOrder` definiert mit der neue Bestellung einem Kunden zugeordnet werden können.

a) Wie sieht das von JPA generierte DB Schema aus?

b) Kann die Assoziation auch anders in der Datenbank abgelegt werden?
Hinweis: Annotation `@JoinColumn`

c) Was passiert (bei beiden Varianten), wenn im Testprogramm `Test4` der Kommentar bei `c2.addOrder(o1);` gelöscht wird?

5) **Fetchtype**
Zwischen den Entitäten `Customer` und `Address` ist eine (unidirektionale) 1:1 Beziehung definiert. Ändern Sie den `FetchType` dieser Assoziation von `EAGER` auf `LAZY` und geben Sie aus, welchen Typ das von der Methode `getAddress` zurückgegebene Objekt hat. Der Typ kann wie folgt ausgegeben werden:

```
System.out.println(c.getAddress().getClass());
```

Welchen Typ hat die Referenz wenn die Assoziation nicht *lazy* deklariert ist?

Messen Sie den Zeitaufwand für den Zugriff auf ein Feld des assoziierten `Address`-Objektes mit

```
long start = System.currentTimeMillis();
for(int i=0; i < 1_000_000; i++){
      c.getAddress().getStreet();
}
System.out.println(System.currentTimeMillis() - start);
```

und vergleichen Sie die Zeiten zwischen `EAGER` und `LAZY` Loading (vielleicht müssen Sie auf Ihrem Rechner die Anzahl Schleifendurchläufe erhöhen).

6) **Flush Mode**
Mit der Methode `setFlushMode` kann auf dem Entity-Manager eingestellt werden, wie/wann der Persistenzkontext mit der Datenbank synchronisiert werden soll.

In der Datenbank existieren zwei Kunden und zwei Adressen (beide mit city=Windisch).

Wir betrachten nun folgende Anweisungen aus Klasse `Test6`:

```
Customer c = em.find(Customer.class, 1);
c.getAddress().setCity("Basel");

TypedQuery<String> q =
      em.createQuery("select a.city from Address a", String.class);
List<String> cities = q.getResultList();
for (String city : cities) {
      System.out.println(city);
}
```

a) Welches Resultat erhält man auf der Konsole mit
`em.setFlushMode(FlushModeType.COMMIT);`
bzw. mit
`em.setFlushMode(FlushModeType.AUTO);`

Welche Auswirkung hat somit der flush-Mode?

b) Welches Resultat erhält man, wenn man nach dem Ändern der Adresse explizit einen Aufruf von `em.flush()` ausführt?

7) **Primärschlüsselgenerierung**
Für diesen Task wechseln wir auf eine lokale Datenbank, d.h. setzen Sie die folgenden Properties:

```
spring.jpa.hibernate.ddl-auto=update
spring.sql.init.mode=never
spring.datasource.url=jdbc:h2:file:./build/lab-jpa-db
```

Im Paket `ch.fhnw.edu.jpa.test7` ist die Entität `Customer` definiert. Das Testprogramm `Test7` erzeugt zwei Kunden (Meier / Müller) und gibt deren Primärschlüssel aus. Führen Sie dieses Programm mehrfach auf der lokalen Datenbank aus. Das Datenbankfile wird im Verzeichnis `build/` erzeugt und hat den Namen `lab-jpa-db.mv.db`.

Falls Sie mit der h2 Konsole auf diese lokale Datenbank zugreifen wollen dann müssen Sie als URL jdbc:h2:file:./build/lab-jpa-db angeben, und die Felder „User Name“ und „Password“ müssen beide leer sein!

a) Die Strategie mit der die Primärschlüssel erzeugt werden ist initial auf `IDENTITY` gesetzt:
`@GeneratedValue(strategy=GenerationType.IDENTITY)`

Welche Schlüssel werden generiert, wenn das Programm nacheinander mehrfach ausgeführt wird? Welche Methode wird dabei von der H2-Datenbank verwendet?

b) Wie werden Primärschlüssel vergeben, wenn die Strategie auf `TABLE` gesetzt wird?
Löschen Sie die Datenbank (die Dateien `build/lab-jpa-db.*`) bevor sie neue Entitäten mit dem Testprogramm in die Datenbank einfügen nachdem sie die PK-Strategie geändert haben).

Schauen Sie nach, welche Tabellen dadurch zusätzlich in der Datenbank angelegt wurden.
Die Datenbank erreichen Sie über die URL `jdbc:h2:file:./build/lab-jpa-db`.