Übung W2: Persistenz mit JPA

In dieser Aufgabe starten wir mit der MovieRental Applikation. Sie finden dazu auf dem Server-Laufwerk `\\Fsemu18.edu.ds.fhnw.ch\e_18_data11$\E1811_Unterrichte_Bachelor\E1811_Unterrichte_I\5Iw\eaf` im Verzeichnis Lektion02\Uebung02 entsprechende Gradle-Projekte, welche den Code für diese Filmausleihapplikation enthalten. Wenn Sie die ZIP-Dateien lokal auf Ihrem Rechner ausgepackt haben, so können Sie diese direkt mit

File -> Import… -> Gradle Project

in ihr Eclipse (oder IntelliJ) importieren.

movierental.memory
Das Projekt movierental.memory implementiert einen Server, der die Daten in-memory verwaltet und diese über eine REST Schnittstelle bereitstellt. Die REST Schnittstelle ist mit Hilfe von SpringMVC realisiert. Wenn Sie das Programm starten, so können Sie z.B. mit einem GET Request auf

http://localhost:8080/movierental/movies/

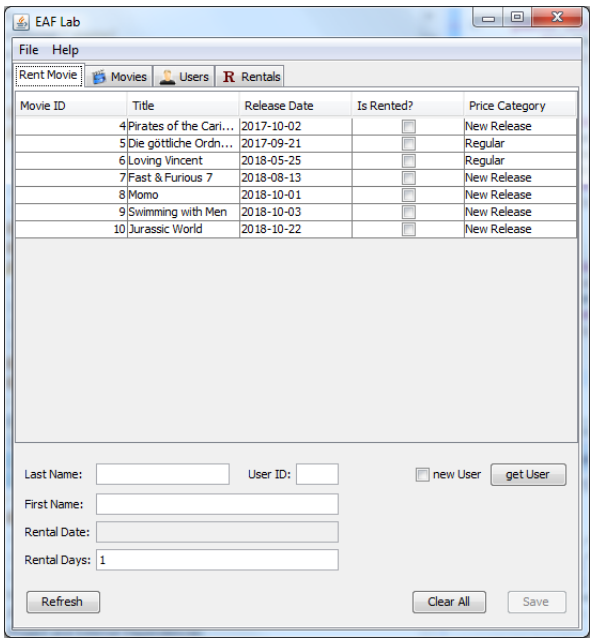
auf die Filme zugreifen.

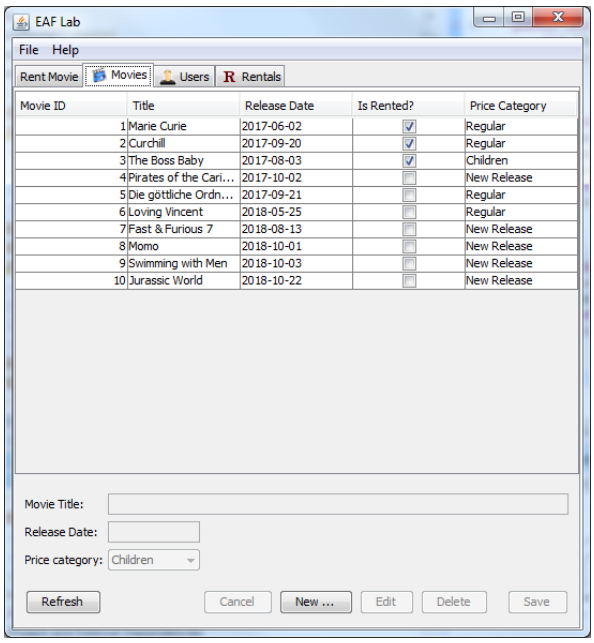
movierenatal.client
Das Projekt movierental.client implementiert das GUI welches über die REST Schnittstelle auf den Server zugreift. Das GUI wird mit der Klasse `ch.fhnw.edu.rental.MovieRentalClient` gestartet oder direkt mit dem Gradle-Task

```
gradlew bootRun
```

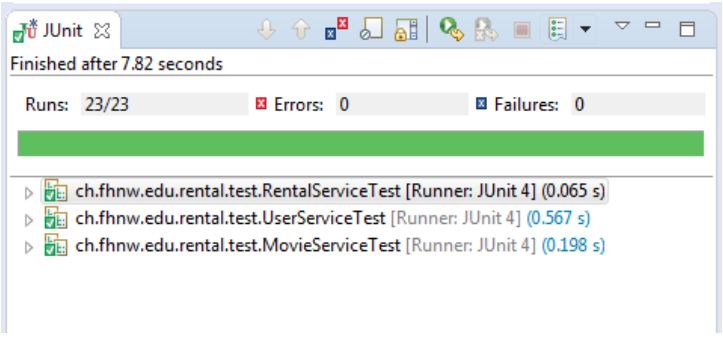
Es handelt sich bei dieser Klasse um eine `@SpringBootApplication` welche ein Swing GUI startet.

Nach erfolgreichem Start der Applikation meldet sich das GUI:





Im Verzeichnis `src/test/java` des Projektes `movierental.memory` liegen Tests vor, die ebenfalls bereits ausgeführt werden können;



movierental.jpa

Das Projekt movierental.jpa ist die Ausgangslage für diese Übungsaufgabe. Sie sollen darin die Persistenzschicht mit Hilfe von JPA realisieren, d.h. die Repositories, die im Projekt movierental.memory *in-memory* realisiert sind, sollen durch JPA Repositories ersetzt werden.

Ihre Aufgabe in dieser Übung ist es, die Entitäten mit JPA Annotationen zu versehen und die Klassen

- JpaMovieRepository
- JpaRentalRepository
- JpaUserRepository
- JpaPriceCategoryRepository

mit Hilfe von JPA, d.h. mit Hilfe des Entity-Managers zu realisieren. Diese Repository-Klassen sind bereits als leere mit TODOs versehene Klassenrahmen vorhanden.

Aus den Service-Implementierungen könnte man auch direkt auf den JPA Entity-Manager zugreifen, dann bräuchte es die Repository-Kassen nicht mehr, aber ich habe mich entschieden, die Repository-Klassen noch nicht zu entfernen, da diese mit Hilfe von Spring Data auch generiert werden können (werden wir später sehen).

Um die Persistenz mit Hilfe von JPA zu realisieren sind grundsätzlich zwei Schritte nötig:

1) Modell-Klassen mit JPA Annotationen versehen
2) Implementierung der Repository-Klassen mit Hilfe des Entity-Managers.

Die MovieRental Applikation können Sie wiederum über die Klasse `MovieRentalServer` starten (sobald Sie die Persistierung mit Hilfe von JPA implementiert haben) und dann kann vom GUI her über die REST-Schnittstelle auf den Server zugegriffen werden oder direkt über
http://localhost:8080/movierental/movies
http://localhost:8080/movierental/users
http://localhost:8080/movierental/rentals

Testklassen (für die Repositories und die Services) sind im Projekt ebenfalls enthalten. Mit `@Transactional` wird sichergestellt, dass am Ende jeder Testmethode die aktuelle Transaktion mit einem Rollback abgebrochen wird (damit werden alle in diesem Test gemachten Änderungen zurückgefahren).

```
@RunWith(SpringRunner.class)
@SpringBootTest
@Transactional
public class MovieServiceTest { … }
```

Folgende Schritte müssen nun ausgeführt werden:

(A) **Anpassungen in den Modellklassen**
Wir verwenden dasselbe Datenbankschema das wir auch in der letzten Übung verwendet haben, und daher müssen die JPA-Entity-Klassen nun so annotiert werden, dass sie auf dieses Schema passen. Man könnte auch auf diesen Schritt verzichten und dann das Schema von JPA generieren lassen, aber damit Sie die zur Verfügung stehenden Annotationen kennenlernen gehen wir den umgekehrten Weg.

(B) **Implementierung der Repository-Klassen**
In diesem Schritt werden die Repository-Klassen so implementiert dass Sie JPA verwenden, um auf die Datenbank zuzugreifen.

(C) **Ausführen der Tests**
Mit diesen Änderungen können nun die Tests ausgeführt werden.

(D) **Starten der Applikation**
Als letzter Schritt werden Sie den MovieRental-Klienten starten um auf den Server zuzugreifen.

(A) Anpassungen der Modellkassen

Wählen Sie beim Import von Annotationen / Interfaces jeweils jene aus dem Paket `javax.persistence` aus und nicht jene aus dem Paket `org.hibernate.annotations`!

1) Fügen Sie bei *allen* Modellklassen die `@Entity` Annotation hinzu (`javax.persistence.Entity`). Dies gilt auch für die Unterklassen von `PriceCategory`. Stellen Sie sicher, dass alle Entity-Klassen einen nicht-privaten no-arg-Konstruktor haben (Sichtbarkeit public, protected oder nichts).

2) Versehen Sie alle Primärschlüssel mit den Annotationen
```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
```

3) Weisen Sie den Entitäten folgende Tabellennamen zu (im Gegensatz zu den Klassen verwenden wir für die Bezeichnung der Datenbanktabellen den Plural).
```
Rental                  => RENTALS
User                    => USERS
Movie                   => MOVIES
PriceCategory           => PRICECATEGORIES
```

4) Weisen Sie den folgenden Feldern Kolonnennamen zu (`@Column(name="…")`):
```
PriceCategory.id        => PRICECATEGORY_ID
Movie.id                => MOVIE_ID
Movie.title             => MOVIE_TITLE
Movie.rented            => MOVIE_RENTED
Movie.releaseDate       => MOVIE_RELEASEDATE
User.id                 => USER_ID
User.email              => USER_EMAIL
User.lastName           => USER_NAME
User.firstName          => USER_FIRSTNAME
Rental.id               => RENTAL_ID
Rental.rentalDate       => RENTAL_RENTALDATE
Rental.rentalDays       => RENTAL_RENTALDAYS
```

5) Markieren Sie folgende Referenzen mit der richtigen Annotation: `@OneToOne`, `@OneToMany`, `@ManyToOne` oder `@ManyToMany`. Markieren Sie bidirektionale Annotationen auf der richtigen Seite mit dem Attribut `mappedBy`.
```
Movie.priceCategory
Rental.movie
Rental.user
User.rentals
```

6) Definieren Sie für die Entität `PriceCategory` mit der Annotation `@DiscriminatorColumn` die Kolonne, in welcher der Entitätstyp abgespeichert ist und geben Sie bei den Unterklassen mit der Annotation `@DiscriminatorValue` an, welcher Wert in dieser Kolonne eingetragen ist (`Regular`, `NewRelease` oder `Children`).

7) Setzen Sie die mit der Annotation `@JoinColumn(name="…")` die Namen der FK Kolonnen
```
Movie.priceCategory     => PRICECATEGORY_FK
Rental.user             => USER_ID
Rental.movie            => MOVIE_ID
```

Nach diesen Änderungen können sie die Applikation `MovieRentalServer` starten. Wenn diese ohne Fehler startet, dann haben Sie sich bei den Annotationen nicht vertippt.

(B) Implementierung der Repository-Klassen

8) Deklarieren Sie in jeder Repository Klasse eine Referenz auf den Entity-Manager:
```
@PersistenceContext
private EntityManager em;
```

9) Implementieren Sie alle `findById`-Methoden mit (am Beispiel von `JpaRentalRepository`)
```
return Optional.ofNullable(em.find(Rental.class, id));
```

10) Implementieren Sie alle `save`-Methoden mit
```
return em.merge(entity);
```
Beachten Sie, dass die `merge` Methode den übergebenen Parameter *nicht* ändert.

11) Implementieren Sie alle `delete`-Methoden mit
```
em.remove(entity);
```
und die `deleteById-Mmethoden` mit. (am Beispiel von `JpaUserRepository`)
```
em.remove(em.getReference(User.class, id));
```

12) Implementieren Sie die `findAll` Methoden mit (am Beispiel von `JpaMovieRepository`)
```
TypedQuery<Movie> q = em.createQuery("SELECT m FROM Movie m", Movie.class);
return q.getResultList();
```

13) Implementieren Sie die beiden Methoden `JpaMovieRepository.findByTitle` und `JpaUserRepository.findByLastName/findByFirstName/findByEmail`. Die Implementierung für `JpaMovieRepository.findByTitle` sieht wie folgt aus:
```
TypedQuery<Movie> q = em.createQuery(
        "SELECT m FROM Movie m WHERE m.title = :title",
        Movie.class);
q.setParameter("title", title);
return q.getResultList();
```
Offen ist danach nur noch die Methode `JpaRentalRepository.findByUser`. Diese Methode wurde in der JDBC-Version verwendet, um die Rentals eines Benutzers laden zu können. In der JPA-Version wird diese Methode jedoch nicht mehr referenziert, da JPA beim Laden eines Benutzers automatisch auch die Ausleihen lädt. Diese Methode kann nun wie folgt implementiert werden:
```
public List<Rental> findByUser (User user) {
        return user.getRentals();
}
```

14) Die `count`-Methoden kann (am Beispiel der des `User`-Repository) wie folgt implementiert werden:
```
return em.createQuery("SELECT COUNT(u) FROM User u", Long.class)
                                                        .getSingleResult();
```

15) Die `existsById`-Methode kann mit der Anweisung
```
        return findById(id).isPresent();
```
implementiert werden, dabei wird jedoch die entsprechende Entität in den Persistenzkontext geladen. Folgende Anweisung prüft, ob eine Entität in der Datenbank existiert, ohne diese zu laden:
```
TypedQuery<Long> q = em.createQuery(
        "SELECT COUNT(u) FROM User u WHERE u.id = :id", Long.class);
q.setParameter("id", id);
return q.getSingleResult() > 0;
```

Viele dieser Methoden können auch in eine abstrakte Basisklasse verschoben werden. Sinnvollerweise übergeben Sie dabei den Typ der Entität (vom Typ Class<?>) der Basisklasse.

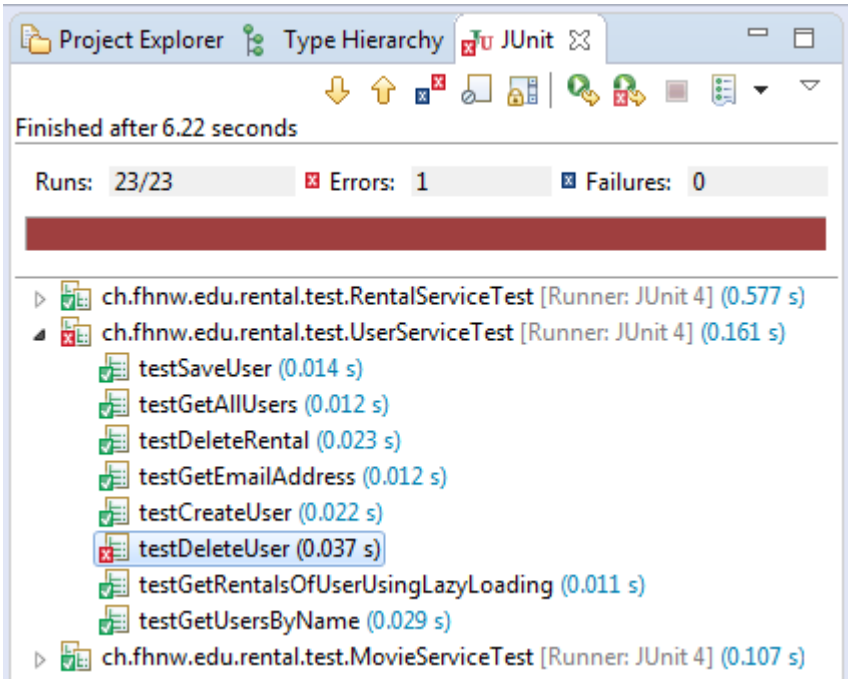
(C) Tests

Mit diesen Implementierungen sollte das Testresultat wie folgt aussehen:

Der Test `testDeleteUser` schlägt fehl, da die mit einem Benutzer assoziierten Ausleihobjekte nicht automatisch gelöscht werden. Diese Ausleihobjekte würden dann mit ihrem Fremdschlüssel auf ein Objekt verweisen, welches nicht mehr existiert.

Auch dies kann in JPA mit einer Annotation erreicht werden; das werden wir uns nächste Woche anschauen.

Falls die Tests fehlschlagen:
- Vielleicht fehlen noch parameterliste Konstruktoren in den Entity-Klassen
- Vielleicht haben Sie nach dem Kopieren von Annotationen nötige Anpassungen nicht mehr gemacht.

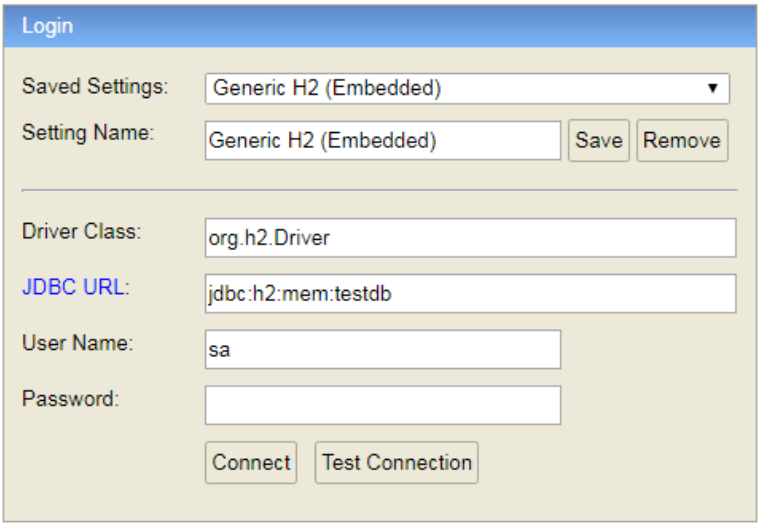


(D) MovieRental Applikation starten

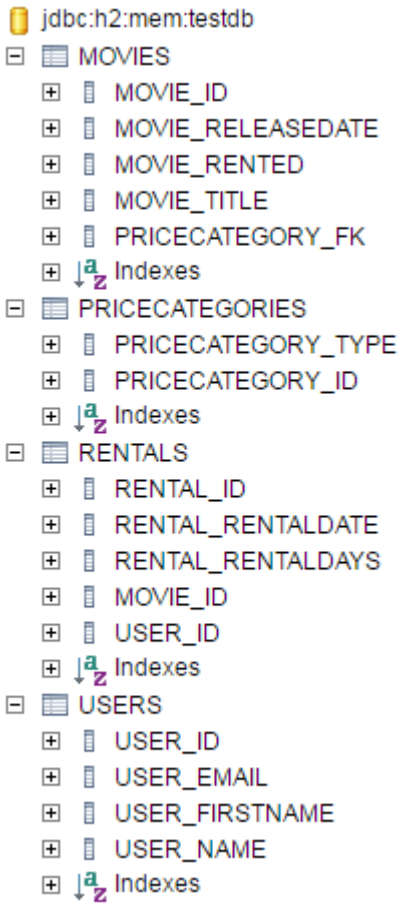
In der Datei `application.properties` wird mit den Deklarationen

```
spring.h2.console.enabled=true
spring.h2.console.path=/h2
```

die H2-Konsole aktiviert. Die Konsole steht dann unter der URL http://localhost:8080/h2 zur Verfügung. Falls der Port 8080 auf Ihrer Maschine bereits besetzt ist so können Sie mit dem Property `server.port` einen anderen Port definieren. Verwenden Sie die Settings „Generic H2 (Embedded)“ und als JDBC-URL jdbc:h2:mem:testdb). Sie sehen in der Datenbank dann auch bereits die in der Datei `data.sql` definierten Daten.



Wenn nun das Programm `MovieRentalServer` starten, dann können Sie bei laufender Applikation die H2-Konsole öffnen. Sie sehen dann die von JPA generierten Tabellen (wie rechts dargestellt). Die in den Tabellen abgelegten Daten sind in der Date `data.sql` im Verzeichnis *src/main/resources* definiert.



Mit einem Browser (oder einem Tool wie Postman) können Sie auf die RETS-Schnittstelle zugreifen. Die Liste der Filme können Sie mit einem GET auf localhost:8080/movierental/movies abfragen.

Sie können jedoch auch die Applikation `MovieRentalClient` starten. Die Applikation funktioniert, d.h. Sie können Filme ausleihen und zurückgeben. Wenn Sie den Klienten neu starten sehen Sie immer noch die geänderten Daten solange der Server läuft.

Das Problem, dass mit einem Benutzer nicht auch die Ausleihen gelöscht werden ist für die Applikation kein Problem, denn das GUI prüft ebenfalls vor dem Löschen eines Nutzers, ob dieser noch Ausleihen hat oder nicht. Wir werden in den kommenden Lektionen sehen, wie die Implementierung geändert werden muss, damit auch der `testDeleteUser`-Test nicht mehr fehlschlägt.

Wenn Sie in der Konfigurationsdatei `application.properties` im Serverprogramm den Eintrag
`spring.jpa.show-sql=true`
hinzufügen, dann werden die von JPA generierten SQL Statements ausgegeben.

Ausgabe: 21. 09. 2020
Abgabe: 05. 10. 2020