Hibernate

By Chandrasekaran Janardhanan

Contents

Session pre-requisite

- Core Java

Install and Setup MySQL

Download MySQL

- In browser go to https://dev.mysql.com/downloads/mysql/
- Click the download button adjacent to "Windows (x86, 64-bit) ZIP Archive"
- Extract the root folder "mysql-8.0.20-winx64" into D:

Setup Database

- Create new folder named "data" in D:\mysql-8.0.20-winx64
- In windows file explorer go to D:\mysql-8.0.20-winx64\bin folder
- Click in the empty space next to "bin" folder on the top of the window.
- Type "cmd" and press enter.
- This open command prompt in the bin folder of mysql.
- Execute the following command:

```
mysqld --initialize --console --basedir=D:\mysql-8.0.20-winx64 --datadir=D:\mysql-8.0.20-winx64\data
```

IMPORTANT NOTE: If the folder or version number is different, then ensure appropriate folder name.

- Make note of the password generated in the console, look for the line below:

```
A temporary password is generated for root@localhost: YQU6E,xraigx
```

- Select the password and press enter.
- Open Notepad and press Ctrl+V which will paste the password.
- If the MySQL server installation fails or gives error for some reasons, the use the below link to download MySQL server zip file:

  https://drive.google.com/file/d/1CRRJ-xUtDapgFIZPK2dJ3qh1IHJCeLH6/view?usp=sharing

Change password for 'root' user

- Now execute the following command to start the server.

```
mysqld --console
```

- Wait till it displays the message "ready for connections"
- Open a new command prompt window in mysql\bin folder.
- Run the following command:

```
mysql -u root -p
```

- Copy the password from notepad
- Open the command prompt window and right click, which will paste the password.
- Press enter and providing the password.
- It will open a mysql> prompt
- Execute the following command:

```
alter user 'root'@'localhost' identified by 'password';
```

- Execute the following command and press enter

```
exit
```

- Run the following command:

```
mysql -u root -p
```

- Enter the password as “password”
- It should not login in the new password.

Setup moviedb

- Create a new folder moviedb in D: using windows explorer.
- Open the URL https://drive.google.com/file/d/1eQmq_iHqlFRigbJW8Wc1E4W8bshdsT9Y/view?usp=sharing
- Download moviedb.sql into D:\moviedb folder
- Go to MySQL client command prompt and execute the following command.

```
source d:\moviedb\moviedb.sql
```

- This will create the tables and data required for movie database.
- Use the below command to change to the moviedb database

```
use moviedb;
```

- Run the following queries one by one to check if train data is available correctly.

```
select * from movie;
select * from genre;
select * from mpaa_rating;
select * from movie_genre;
select * from user;
```

Stopping MySQL Server

- Open the command window were the server is running
- Press Ctrl+C to stop the server.

Starting MySQL Server and MySQL Client

- Open command prompt in mysql bin folder.
- Execute the following command to start the server:

```
mysqld --console
```

- Open another command prompt in mysql bin folder.
- Execute the following to start the client. Enter ‘password’ as password when prompted.

```
mysqld -u root -p
```

Display list of Languages from MySQL database using Hibernate

Problem

Display list of languages from MySQL database using Hibernate.

Expected Output in Eclipse Console

```
1 en English
2 es Spanish
3 zh Chinese
4 hi Hindi
5 ja Japanese
```

Solution

Prepare the database

- Open MySQL client in command prompt and execute the below commands one by one:

```
create schema newsdb;
use newsdb;
create table language(la_id int not null auto_increment, la_code varchar(2),
la_name varchar(45), primary key (la_id));
insert into language (la_id, la_code, la_name) values (1, 'en', 'English');
insert into language (la_id, la_code, la_name) values (2, 'es', 'Spanish');
insert into language (la_id, la_code, la_name) values (3, 'zh', 'Chinese');
insert into language (la_id, la_code, la_name) values (4, 'hi', 'Hindi');
insert into language (la_id, la_code, la_name) values (5, 'ja', 'Japanese');
```

Create Project

- Open https://start.spring.io/ and provide details as below:
  - Group: com.company
  - ArtifactId: hiblearn
  - Dependencies: 1) Spring Data JPA 2) MySQL Driver
- Click Generate to download and extract 'hiblearn' folder to Eclipse workspace
- Open Eclipse > File > Import > Existing Maven Projects > Next
- Click "Browse" and select the 'hiblearn' folder, the click “Finish”

Setup Hibernate

- Right click on ‘src/main/resources’ > New > Other... > Folder
- Provide folder name as ‘/META-INF’
- Right click on ‘META-INF’ > New > Other... > File
- Enter file name as ‘persistence.xml’ and click Finish
- Copy and paste the text below:

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
             version="2.1">
    <persistence-unit name="news">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <class>org.hibernate.documentation.userguide.Document</class>
        <properties>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/newsdb" />
            <property name="javax.persistence.jdbc.user" value="root" />
            <property name="javax.persistence.jdbc.password" value="password" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.hbm2ddl.auto" value="update" />
        </properties>
    </persistence-unit>
```

```
</persistence>
```

Create Model Class

- Create new package 'com.company.hiblearn.model'
- Create new class Language with below properties:
  - id: long
  - code: String
  - name: String
- Generate all parameter constructor, getters, setters and toString().
- Include annotations at class level:

```
import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name="language")
public class Language {
```

- Annotations for id. Import all classes from javax.persistence.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "la_id")
private long id;
```

- Annotations for code and name.

```
@Column(name = "la_code")
private String code;

@Column(name = "la_name")
private String name;
```

Implement main() method

- Comment the existing spring application run method and implement as specified below:

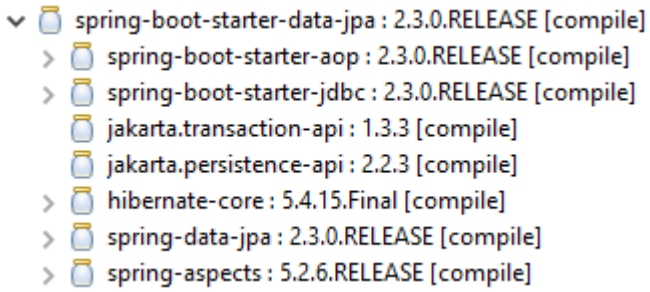
```
public static void main(String[] args) {
    //SpringApplication.run(HiblearnApplication.class, args);
    EntityManagerFactory factory = Persistence
            .createEntityManagerFactory("news");
    EntityManager entityManager = factory.createEntityManager();

    entityManager.getTransaction().begin();

    List<Language> languageList = entityManager.createQuery(
            "select language from Language language",
            Language.class).getResultList();

    for (Language language : languageList) {
        System.out.print(language.getId() + " ");
        System.out.print(language.getCode() + " ");
        System.out.println(language.getName());
    }

    entityManager.getTransaction().commit();
}
```

- Run and check if the languages are displayed.

Explanation

Spring Data JPA Starter

- The spring-boot-starter-data-jpa starter pack helps to get the dependencies for accessing data. Refer screenshot from pom.xml dependency hierarchy:



- Following are the key dependencies:
  - JDBC – Java Database Connectivity
  - JTA - Java Transaction API
  - JPA - Java Persistence API
  - Hibernate
  - Spring Data JPA – Framework provided by spring to avoid boiler plate code in Hibernate
- We have also included the MySQL Connector dependency which contains the Java library to connect

persistence.xml

- Configure the database to be connected using META-INF/persistence.xml
- Persistence Unit in persistence.xml defines reference to a database.
- The JDBC Driver class is defined as com.mysql.cj.jdbc.Driver. This class is part of MySQL Connector jar file.
- The JDBC URL to connect to the database server.
- User and Password to connect to the database.
- We will discuss the setting 'hibernate.hbm2ddl.auto' after a while.

Language.java

- @Entity – Defines an entity with class name. In the HQL query to get the list of languages, we can find 'Language' in the query which represents Language class. If @Entity is not defined then the Hibernate will not be able to resolve 'Language' in HQL.
- @Table – Links a database with the class where this annotation is defined. If name is not defined then the
- @Id – Defines the respective instance variable as primary key column.
- @GeneratedValue – Defines that the respective field will be auto generated
- Generation.Type as IDENTITY mentions that the database has to generated the value
- @Column – Links the instance variable with the respective database table column.

About hibernate.hbm2ddl.auto configuration

- This configuration helps hibernate to decide about the Data Definition Language (DDL).
- CREATE, ALTER and DROP comes under DDL.
- Hibernate wants to know how to behave if a table defined in the model class is not available in database. This behavior is performed when creating the EntityManagerFactory or SessionManagerFactory. Following are the possible values and its respective behavior.

- validate – A validation is run to check the existence of tables and columns in database, wherever @Table and @Column is defined. An exception is thrown if there is any discrepancy.
- update – Creates table or column if it does not exist
- create-only – Database creation will be generated
- create-drop – Creates table when starting hibernate SessionFactory and closes when the application ends.
- none – No validation or creation of tables is done. This is the default value.

❖ It is recommended to create the database schema and tables and then either use the update or validate option.

HiblearnApplication.java

❖ Read the newsdb configuration from persistence.xml using Persistence.createEntityManagerFactory() method.

❖ EntityManagerFactory instance helps to manage all communications of an application and there should be only one instance of this class created.

❖ Using createEntityManager() method in EntityManagerFactory an EntityManager can be created.

❖ The createQuery() method in EntityManager helps to create a HQL query.

❖ The getResultList() method helps to obtain the query result as a list.

What is JDBC, Hibernate and JPA? How are they related to each other?

About JDBC

❖ JDBC stands for Java Database Connectivity

❖ JDBC API provides universal data access from the Java programming language.

❖ JDBC is a specification standard for connection to relational databases.

❖ JDBC as such does not have any concrete implementation.

❖ JDBC represents a set of documents and Java interfaces.

❖ Implementation of JDBC has to be done by the respective database provider. For example, the JDBC driver for connecting to Oracle Database needs to be provided by Oracle.

❖ The JDBC driver implement can be implemented in four ways:

- **Type 1** – This kind of driver depends on a native library. (Example: JDBC-ODBC drive, ODBC is a DLL provided by Microsoft to connect to database. The JDBC driver will be making calls to this DLL to perform any database operations)
- **Type 2** – These kinds of drivers are written partly in Java and partly in native code.
- **Type 3** – These are pure Java drivers but uses a middleware server to connect to the database.
- **Type 4** – These are pure Java drivers that directly connects with the database. The MySQL Connector library is a pure Java driver.

Getting started with Hibernate

❖ Hibernate is an Object/Relational Mapping (ORM) framework

❖ Hibernate is concerned with data persistence to relational databases

❖ Hibernate performs database operations using JDBC

History of Hibernate and JPA

- 2001
  - Hibernate implemented by Gavin King as an alternative to EJB 2.0
  - The primary objective is to simplify the complexities in implementing using EJB 2.0.
  - Version 1 was released without annotations and XML based configuration
  - Many persistence frameworks evolved during this period as an alternative to EJB
  - JPA as a specification was initiated through Java Community Process to standardize the alternate persistence frameworks
- 2006 – JPA 1.0 was released. Hibernate 3.2 released with JPA 1.0 compliance.
- 2010 – Hibernate 3.5 released with JPA 2.0 compliance
- 2013 – Hibernate 4.3 released with JPA 2.1 compliance
- 2018 – Hibernate 5.3 released with JPA 2.2 compliance
- 2020 – Latest version of Hibernate as of Jun 2020 is Version 5.4

Hibernate Architecture

- Hibernate sits between Java application's data access layer and database.
- For performing the actual database operations Hibernate uses JDBC. Refer diagram below:

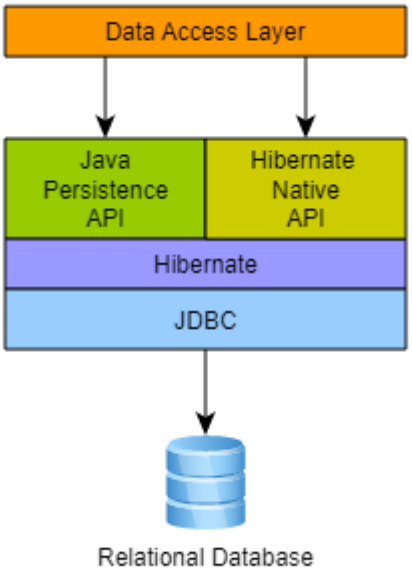


Image courtesy: docs.jboss.org

Hibernate API vs JPA API

- Comparison diagram for Hibernate and JPA

Hibernate API	JPA API
• SessionFactory • Session	• EntityManagerFactory • EntityManager

- Hibernate configuration file hibernate.cfg.xml equivalent to persistence.xml:

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory>
      <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
      <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
      <property name = "hibernate.connection.url">jdbc:mysql://localhost/test</property>
      <property name = "hibernate.connection.username">root</property>
      <property name = "hibernate.connection.password">root123</property>
      <mapping resource = "Employee.hbm.xml"/>
   </session-factory>
</hibernate-configuration>
```

- DB mapping using XML configuration file. Equivalent to annotations defined in Language.java.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
   <class name = "Employee" table = "EMPLOYEE">

      <meta attribute = "class-description">
         This class contains the employee detail.
      </meta>

      <id name = "id" type = "int" column = "id">
         <generator class="native"/>
      </id>

      <property name = "firstName" column = "first_name" type = "string"/>
      <property name = "lastName" column = "last_name" type = "string"/>
      <property name = "salary" column = "salary" type = "int"/>

   </class>
</hibernate-mapping>
```

- Creation of SessionFactory

```
private static SessionFactory factory;
public static void main(String[] args) {

   try {
      factory = new Configuration().configure().buildSessionFactory();
   } catch (Throwable ex) {
      System.err.println("Failed to create sessionFactory object." + ex);
      throw new ExceptionInInitializerError(ex);
   }
```

- Code to get the list of all entities:

```
public void listEmployees( ){
   Session session = factory.openSession();
   Transaction tx = null;

   try {
      tx = session.beginTransaction();
      List employees = session.createQuery("FROM Employee").list();
      for (Iterator iterator = employees.iterator(); iterator.hasNext();){
         Employee employee = (Employee) iterator.next();
         System.out.print("First Name: " + employee.getFirstName());
         System.out.print("  Last Name: " + employee.getLastName());
         System.out.println("  Salary: " + employee.getSalary());
      }
      tx.commit();
   } catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
   } finally {
      session.close();
   }
}
```

Code Snippet courtesy: https://www.tutorialspoint.com/hibernate/hibernate_configuration.htm

- It is recommended to use JPA, since it is a specification standard, which makes it easier to switch to switch to any JPA implementation provider other than Hibenate.
- All our examples will be implemented using JPA.

Check your understanding

- List the dependencies used for implementing this example?
- Which starter pack of Spring Boot was used for Hibernate?
- What is the expanded form of JDBC?
- What exactly is JDBC?
- What is a JDBC Driver?
- What JDBC Driver did we use for connecting to the MySQL database?
- Where should be the persistence.xml file placed?
- What is the significance of persistence.xml file and what does it contain?
- List the annotations required for mapping tables, columns and primary key columns.
- List the steps required to run a query and get the result in a list.
- How is Hibernate and JPA related?
- Brief about the difference between SessionFactory and EntityManagerFactory.
- What is the recommended to be used SessionFactory or EntityManagerFactory?
- Before annotations, how was hibernate related configuration done?

Configuring as Spring Boot application

Activity

Convert the language retrieval into a Spring Boot application.

Expected Output in Eclipse Console

```
1 en English
2 es Spanish
3 zh Chinese
4 hi Hindi
5 ja Japanese
```

Steps to implement

application.properties

- Copy and paste the properties below:

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/newsdb
spring.datasource.username=root
spring.datasource.password=password
spring.datasource.pool-size=30

spring.jpa.hibernate.ddl-auto=update

# Hibernate logs for displaying executed SQL, input and output
logging.level.org.hibernate.SQL=trace
logging.level.org.hibernate.type.descriptor.sql=trace

logging.level.com.company.hiblearn=debug
```

LanguageDao.java

- Create package com.company.hiblearn.dao
- Create class LanguageDao
- Include annotations @Repository and @Transactional

```
@Repository
@Transactional
public class LanguageDao {
```

- Include logger
- Include EntityManager

```
@PersistenceContext
private EntityManager entityManager;
```

- Include list() method

```
public List<Language> list() {
    LOGGER.info("Start");
    return entityManager.createQuery("FROM Language",
            Language.class).getResultList();
}
```

LanguageService.java

- Create package com.company.hiblearn.service
- Create class LanguageService
- Include annotation @Service

```
@Service
public class LanguageService {
```

- Include logger
- Auto wire dao

```
@Autowired
private LanguageDao languageDao;
```

- Include list() method

```
public List<Language> list() {
    LOGGER.info("Start");
    return languageDao.list();
}
```

Runner.java

- Create new class Runner in com.company.hiblearn with annotation @Component
- Include logger.
- Auto wire LanguageService

```
@Autowired
private LanguageService languageService;
```

- Include method

```
private List<Language> listLanguages() {
    LOGGER.info("Start");
    List<Language> languages = languageService.list();
    LOGGER.debug("Language list: " + languages);
    return languages;
}
```

- Include run() method

```
@Override
public void run(String... args) throws Exception {
    LOGGER.info("Start");
    listLanguages();
    LOGGER.info("End");
}
```

- Run HiblearnApplication and check if the language from the database is listed.
- Manually add a new entry in database and check if the new entry is reflected in the list of languages returned.

Explanation

application.properties

- The configuration done in persistence.xml is moved to application.properties
- The pool property defines the connection pooling. Spring creates necessary connection pooling to manage the connections.
- Connection pooling helps to optimally use connections to improve the performance.
- Spring Boot applications default log level is info, hence debug helps to see the log messages that we have included.

LanguageDao.java

- This class is defined to interact with the database.
- The @Repository annotation represents that this class handles database connections.

- The @Transactional annotation avoids the need to manually handling transactions. Include the following property in application.properties and check the logs, it should generate the logs on when the transaction was initiated and committed.

```
logging.level.org.springframework.orm.jpa.JpaTransactionManager=debug
logging.pattern.console=%d{dd-MM-yy} %d{HH:mm:ss.SSS} [%-10.10thread] %5p %-30.30logger{25} %25M %-3L %m%n
```

- The @Transaction annotation can be defined at class level and method level of Service and DAO classes.
- Defining at class level ensures all the methods are transaction enabled.
- Defining at method level ensures that the specific method is transaction enabled.
- The ideal location is to place it at the method level in Service class
- Place the @Transaction annotation in the class level and check if the transaction works.
- The @PersistentContext annotation helps in injecting the EntityManagerFactory and providing us an instance of EntityManager. The creation of EntityManagerFactory is done based on the properties defined in application.properties file.
- The createQuery() method helps to create a query based on the Java classes that are mapped to the database. The 'Language' mentioned in the query refers to the Java class, based on the annotations defined in the Language class, HQL will frame the select query and execute it in the database.
- Check out how drastically the number of lines of code get reduced for performing a database operation in list() method.

LanguageService.java

- This class is a placeholder to manage business logic.
- In this class we are auto wiring DAO and calling the respective method.

Runner.java

- This class helps in creating a command line runner.
- Spring framework calls the run() method after completion of the main method.
- This also makes it easier to inject the LanguageService class.
- The listLanguages() method calls the respective service method and gets the list of languages and prints the same.
- The listLanguages() method is called from run() method.

Integration with spring-learn (applicable only if Spring REST is part of learning)

- Open spring-learn project
- Copy application.properties entries
- Copy LanguageDao
- Modify LanguageService method
- Auto wire LangaugeService in LanguageController
- In respective LanguageController method call list() method from LanguageService.
- Test the service from Postman.

Get language based on language code

Problem

Get the language based on language code.

Steps

LanguageDao.java

- Implement method.

```
public Language findByCode(String code) {
    LOGGER.info("Start");
    LOGGER.debug("Language code: " + code);
    String query = "from Language where code = ?1";
    return (Language) entityManager.createQuery(query).setParameter(1, code)
            .getSingleResult();
}
```

LanguageService.java

- Implement method.

```
public Language findByCode(String code) {
    LOGGER.info("Start");
    LOGGER.debug("Language code: " + code);
    return languageDao.findByCode(code);
}
```

Runner.java

- Implement method

```
public void findByCode(String code) {
    LOGGER.info("Start");
    Language language = languageService.findByCode(code);
    LOGGER.debug("Language: " + language);
}
```

- Invoke new method in run method.

```
@Override
public void run(String... args) throws Exception {
    LOGGER.info("Start");
    //listLanguages();
    findByCode("en");
    LOGGER.info("End");
}
```

- Run HiblearnApplication and check if English language object is printed in the console.

Integration with spring-learn (applicable only if Spring REST is part of learning)

- Update LanguageDao and LanguageService accordingly, then test from Postman.

Explanation

- The ‘Language’ in query refers to the class
- The ‘code’ in the query refers to the property in Language.
- JPA query parameters are defined with ‘?’ and index number.
- The value for code is set using setParameter() method
- The getSingleResult() directly returns the single item.
- Check the logs for the select query that is executed and view in a SQL formatter

Add new language

Problem

Implement adding language.

Steps

LanguageDao.java

- Implement method.

```
public void add(Language language) {
    LOGGER.info("Start");
    entityManager.persist(language);
    LOGGER.info("End");
}
```

LanguageService.java

- Implement method.

```
public void add(Language language) {
    LOGGER.info("Start");
    languageDao.add(language);
    LOGGER.info("End");
}
```

Runner.java

- Implement method

```
private void addLanguage() {
    LOGGER.info("Start");
    languageService.add(new Language("fr", "France"));
    LOGGER.info("End");
}
```

- Invoke new method in run method.

```
@Override
public void run(String... args) throws Exception {
    LOGGER.info("Start");
    //listLanguages();
    addLanguage();
    findByCode("fr");
    LOGGER.info("End");
}
```

- Run HiblearnApplication and check if the new language is added in the database.

Integration with spring-learn (applicable only if Spring REST is part of learning)

- Update LanguageDao and LanguageService accordingly, then test from Postman.

Explanation

- The persist() method inserts data into the database table.
- Check the SQL generated in the logs

Update an existing language

Problem

Implement updating language.

Steps

LanguageDao.java

- Implement method.

```
public void update(Language language) {
    LOGGER.info("Start");
    entityManager.merge(language);
    LOGGER.info("End");
}
```

LanguageService.java

- Implement method.

```
public void update(Language language) {
    LOGGER.info("Start");
    entityManager.merge(language);
    LOGGER.info("End");
}
```

Runner.java

- Implement method

```
private void updateLanguage() {
    LOGGER.info("Start");
    Language language = languageService.findByCode("fr");
    language.setName("French");
    languageService.update(language);
    LOGGER.info("End");
}
```

- Invoke new method in run method.

```
@Override
public void run(String... args) throws Exception {
    LOGGER.info("Start");
    //listLanguages();
    //addLanguage();
    updateLanguage();
    findByCode("fr");
    LOGGER.info("End");
}
```

- Run HiblearnApplication and check if the new language is added in the database.

Integration with spring-learn (applicable only if Spring REST is part of learning)

- Update LanguageDao and LanguageService accordingly, then test from Postman.

Explanation

- The merge() method updates data into the database table.
- Check the SQL generated in the log file.

Delete an existing language

Problem

Implement deleting language.

Steps

LanguageDao.java

- Implement method.

```
public void delete(Language language) {
    LOGGER.info("Start");
    entityManager.remove(language);
    LOGGER.info("End");
}
```

LanguageService.java

- Implement method.

```
public void delete(Language language) {
    LOGGER.info("Start");
    languageDao.delete(language);
    LOGGER.info("End");
}
```

Runner.java

- Implement method

```
private void deleteLanguage() {
    LOGGER.info("Start");
    Language language = new Language();
    language.setId(11);
    languageService.delete(language);
    LOGGER.info("End");
}
```

- Invoke new method in run method.

```
@Override
public void run(String... args) throws Exception {
    LOGGER.info("Start");
    //listLanguages();
    //addLanguage();
    //updateLanguage();
    deleteLanguage();
    //findByCode("fr");
    LOGGER.info("End");
}
```

- Run HiblearnApplication and check if the new language is added in the database.
- As the delete operation fails, modify LanguageDao delete() method with following code:

```
public void delete(Language language) {
    LOGGER.info("Start");
    entityManager.remove(entityManager.contains(language) ? language
            : entityManager.merge(language));
    LOGGER.info("End");
}
```

Integration with spring-learn (applicable only if Spring REST is part of learning)

- Update LanguageDao and LanguageService accordingly, then test from Postman.
- Check the SQL generated in the logs

Explanation

- The remove() method deletes data in the database table.
- The detached object refers to an object instance that Hibernate is not sure if it is an entity coming from database.
- The merge() method get the object instance from database based on the id and the deletion is initiated based on this object.
- For this reason, we had implemented the contains() and merge() methods.

Integrate Hibernate with Spring Security

Problem

NOTE: This section is applicable only when Spring Boot learning is also part of the learning topics.

Refer respective section in Spring Boot documentation to implement this.

Get single Movie information based on Id

Problem

Get a single movie information from the database.

Solution (Step #1 – Get Movie and MPAA Rating)

application.properties

- Change the database to moviedb.

Genre.java and MpaaRating.java

- Copy file from spring-learn project.
- Include respective @Entity, @Table, @Id and @Column annotations.

Movie.java

- Copy file from spring-learn project.
- Include @Entity, @Table, @Id
- Include @Column for id, title, budget, bookingsOpen, rating and releaseDate properties.
- Define annotations for instance variable mpaaRating.

```
@ManyToOne
@JoinColumn(name="mv_mr_id")
private MpaaRating mpaaRating;
```

MovieDao.java

- Create new class MovieDao in dao package
- Include @Repository and @Transactional annotations
- Include logger
- Include entityManager variable with @PersistenceContext annotation
- Include method below:

```
public Movie findById(long id) {
    LOGGER.info("Start");
    LOGGER.debug("Movie Id: " + id);
    return entityManager.find(Movie.class, id);
}
```

MovieService.java

- Create new class MovieService in service package
- Include @Service annotation
- Auto wire MovieDao
- Implement method below.

```
public Movie findById(long id) {
    LOGGER.info("Start");
    LOGGER.debug("Movie id: " + id);
    return movieDao.findById(id);
}
```

Runner.java

- Implement method

```
private void getMovieById() {
    LOGGER.info("Start");
    Movie movie = movieService.findById(1);
    LOGGER.debug("Movie: " + movie.getTitle());
    LOGGER.debug("MPAA Rating: " + movie.getMpaaRating());
    LOGGER.info("End");
}
```

- Invoke new method in run method.

```
@Override
public void run(String... args) throws Exception {
    LOGGER.info("Start");
    //listLanguages();
    //addLanguage();
    //updateLanguage();
    //deleteLanguage();
    //findByCode("fr");
    getMovieById();
    LOGGER.info("End");
}
```

- Run HiblearnApplication and check if the following results:
  - There will be a lazy initialization exception in the console log
  - Copy the SQL query from the console log and paste it in an online SQL Formatter. There will a join query on movie and mpaa_rating tables.

```
SELECT movie0_.mv_id             AS mv_id1_2_0_,
       movie0_.mv_bookings_open  AS mv_booki2_2_0_,
       movie0_.mv_budget         AS mv_budge3_2_0_,
       movie0_.mv_duration       AS mv_durat4_2_0_,
       movie0_.mv_mr_id          AS mv_mr_id8_2_0_,
       movie0_.mv_rating         AS mv_ratin5_2_0_,
       movie0_.mv_release_date   AS mv_relea6_2_0_,
       movie0_.mv_title          AS mv_title7_2_0_,
       mpaarating1_.mr_id        AS mr_id1_4_1_,
       mpaarating1_.mr_name      AS mr_name2_4_1_
FROM   movie movie0_
       LEFT OUTER JOIN mpaa_rating mpaarating1_
                    ON movie0_.mv_mr_id = mpaarating1_.mr_id
WHERE  movie0_.mv_id = ?
```

  - When displaying movie in Runner class it tries to get the genres, which again tries to initiate the database call, but the previous session is already closed and it is not able to retrieve the data.
  - To overcome this error, remove the toString() method in Movie.
  - Comment the genres instance variable in Movie.
  - Modify method in Runner to read only the movie title and display.
  - Now it should work fine displaying the movie title and MPAA Rating.

Explanation

- The @ManyToOne annotation helps in mapping the mpaa_rating table.

```
@ManyToOne
@JoinColumn(name="mv_mr_id")
private MpaaRating mpaaRating;
```

- As per JPA specification by default, Eager Fetch is applied For ManyToOne and OneToOne relationships. Hence MPAA Rating details as well is joined and fetched by Hibernate.

- The method find() in EntityManager class helps to get a movie based on the primary key field 'id'.
- This method is the one that initiates the execution of the above-mentioned SQL query.

Solution (Step #2 – Get Genres of a Movie using eager fetch)

Movie.java

- Uncomment genres instance variable.
- Include fetch type for genres.

```
@ManyToMany(fetch = FetchType.EAGER)
@JoinTable(name = "movie_genre",
    joinColumns = @JoinColumn(name = "mg_mv_id"),
    inverseJoinColumns = @JoinColumn(name = "mg_ge_id"))
private List<Genre> genres;
```

Genre.java

- Include fetch type for movies.

```
@ManyToMany(mappedBy="genres", fetch = FetchType.EAGER)
private List<Movie> movies;
```

- Include getter and setter method for movies.

```
public List<Movie> getMovies() {
    return movies;
}

public void setMovies(List<Movie> movies) {
    this.movies = movies;
}
```

MpaaRating.java

- Include fetch type for movies

```
@OneToMany(mappedBy="mpaaRating", fetch = FetchType.EAGER)
private List<Movie> movies;
```

- Include getter and setter method for movies.

```
public List<Movie> getMovies() {
    return movies;
}

public void setMovies(List<Movie> movies) {
    this.movies = movies;
}
```

Runner.java

- Log Genres in getMovieById() method:

```
private void getMovieById() {
    LOGGER.info("Start");
    Movie movie = movieService.findById(1);
    LOGGER.debug("Movie: " + movie.getTitle());
    LOGGER.debug("MPAA Rating: " + movie.getMpaaRating());
    LOGGER.debug("Genres: " + movie.getGenres());
    LOGGER.info("End");
}
```

- Run HiblearnApplication.
- We will get the expected result, but we can see numerous queries executed to fetch the data. The eager fetch, tries to get more details based on the results returned.

Explanation

- The @ManyToMany annotation in Movie class helps to map many to many relationships.
- The @JoinTable annotation helps to associate the movie_genre mapping table. This annotation also helps to define the mapping table columns in the join table.
- The @ManyToMany annotation in Genre class helps to map the many to many relationships with movie. The value "genres" represent the respective instance variable in Movie class.
- By default, fetch type is lazy for ManyToMany relationship.
- We tried including fetch type as eager to see how it works.
- Given the nature of how fetch type eager works it is recommended to avoid using this fetch type.
- In a project, there can be any kind of requirement, it can be to get genres list for a Movie or the reverse is also possible, that is to get the list of movies of a genre. Similar two way data retrieval is also possible for mpaaRating. If both ways of data retrieval is required, then using fetch type eager is not going to solve the problem.
- In the next section, we will see how to handle this by using HQL and fetch.

Integration with REST API

- Include necessary annotations in model classes Movie, MpaaRating and Genre of SpringLearnApplication
- Copy MovieDao class to spring-learn
- Autowire MovieDao in MovieService
- Modify getMovie() method in MovieService to call findById() method in MovieDao.
- Modify application.properties to point to moviedb.
- Run SpringLearnApplication and check in Postman
- A huge amount of data should be returned and the Eclipse console log should have recursive failure.

Solution (Step #3 – Get Genres of a Movie using HQL fetch)

Movie.java, Genre.java and MpaaRating.java

- Remove eager fetch type in Movie, Genre and MpaaRating

MovieDao.java

- Modify findById() method as specified below:

```
public Movie findById(long id) {
    LOGGER.info("Start");
    LOGGER.debug("Movie Id: " + id);
    //return entityManager.find(Movie.class, id);
    return (Movie) entityManager.createQuery("select m from Movie m left join "
            + "m.mpaaRating mr left join m.genres where m.id = ?1")
            .setParameter(1, id).getSingleResult();
}
```

- Run HiblearnApplication.
- It will fail with lazy initialization error for displaying genres.
- Modify findById() method in MovieDao as specified below. The fetch keyword is included with each join.

```
public Movie findById(long id) {
    LOGGER.info("Start");
    LOGGER.debug("Movie Id: " + id);
    //return entityManager.find(Movie.class, id);
    return (Movie) entityManager.createQuery("select m from Movie m left join "
            + "fetch m.mpaaRating mr left join fetch m.genres where m.id = ?1")
            .setParameter(1, id).getSingleResult();
}
```

- Now run the application, which should fetch the genre details.
- Get SQL from logs, format it and check if genres included in the join query.
- We can also see that only one query is executed which fetches all the details.

Explanation

- The fetch keyword in HQL query helps to conditionally fetch what data we need.
- The 'm' in HQL refers alias name for movie. Variable after period refers to the property.
- If fetch is removed on any one of the joins, then that particular data will not be fetched.
- Copy the SQL from console log and format it in an online SQL formatter.
- Execute the query in MySQL client
- We can see that the movie data is repeated based on the number of genres available in a movie.
- Hibernate takes cares of this redundant data is ensures that it returns it as one movie object having an array of genres.
- Below is a sample JDBC code for joining two tables and getting data.

```java
public ArrayList<Train> getTrains() {
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    String query = "select"
            + "   tr_id, tr_number, tr_name, tr_from_st_id, s1.st_code from_code,"
            + "   s1.st_name from_name, tr_to_st_id, s2.st_code to_code, "
            + "   s2.st_name to_name, tr_launch_date, tr_active, tr_type "
            + "from"
            + "   train "
            + "   join"
            + "      station s1 "
            + "      on tr_from_st_id = s1.st_id "
            + "   join"
            + "      station s2" + "      on tr_to_st_id = s2.st_id";
    ArrayList<Train> trainList = new ArrayList<>();
    try {
        connection = ConnectionManager.getConnection();
        statement = connection.prepareStatement(query);
        resultSet = statement.executeQuery();
        while (resultSet.next()) {
            Train train = new Train();
            train.setId(resultSet.getLong("TR_ID"));
            train.setNumber(resultSet.getInt("TR_NUMBER"));
            train.setName(resultSet.getString("TR_NAME"));
            train.setLaunchDate(resultSet.getDate("TR_LAUNCH_DATE"));
            train.setActive(
                    resultSet.getString("TR_ACTIVE").equals("Y") ? true
                            : false);
            train.setType(resultSet.getString("TR_TYPE"));

            Station fromStation = new Station();
            fromStation.setId(resultSet.getLong("TR_FROM_ST_ID"));
            fromStation.setCode(resultSet.getString("FROM_CODE"));
            fromStation.setName(resultSet.getString("FROM_NAME"));
            train.setDepartStation(fromStation);

            Station toStation = new Station();
            toStation.setId(resultSet.getLong("TR_TO_ST_ID"));
            toStation.setCode(resultSet.getString("TO_CODE"));
            toStation.setName(resultSet.getString("TO_NAME"));
            train.setArrivalStation(toStation);
            trainList.add(train);
        }
    } catch (SQLException e) {
        e.printStackTrace();
        throw new AppException(e.getMessage());
    } finally {
        ConnectionManager.close(connection, statement);
    }
    System.out.println(trainList);
    return trainList;
}
```

Implement the following services

GenreDao.java and GenreService

- Implement list() and findById() methods.

MpaaRatingDao.java and MpaaRatingService

- Implement list() and findById() methods.

MovieDao.java and MovieService.java

- Implement list() using HQL and fetch type.

Runner.java

- Autowire services.

```
@Autowired
private GenreService genreService;

@Autowired
private MpaaRatingService mpaaRatingService;
```

- Include method.

```
private void displayMovieData() {
    LOGGER.info("Start");
    LOGGER.debug("Movie List {}: ", movieService.list());
    LOGGER.debug("Genre List {}: ", genreService.list());
    LOGGER.debug("MpaaRating List {}: ", mpaaRatingService.list());
    LOGGER.debug("Genre findById: {}", genreService.findById(1));
    LOGGER.debug("MpaaRating findById: {}", mpaaRatingService.findById(1));
    LOGGER.info("End");
}
```

- Include method invocation in main.

```
@Override
public void run(String... args) throws Exception {
    LOGGER.info("Start");
    //listLanguages();
    //addLanguage();
    //updateLanguage();
    //deleteLanguage();
    //findByCode("en");
    //getMovieById();
    displayMovieData();
    LOGGER.info("End");
}
```

Integration with REST API

- Copy and paste findById() method from MovieDao in springlearn to hiblearn.
- Copy and paste GenreDao, MpaaRatingDao, GenreService and MpaaRatingService from hiblearn to springlearn.
- Create or modify MovieController, GenreController and MpaaRatingController
- Test getting all genres and specific genre
- Test getting all mpaa ratings and specific mpaa rating.
- Testing getting a specific movie it will fail with recursive error.
- Include new DTOs MovieDto, GenreDto and MpaaRatingDto with exact same field names in Movie, Genre and MpaaRating.
- Modify method in MovieController

```
@GetMapping("/{movieId}")
public MovieDto getMovie(@PathVariable long movieId) {
    LOGGER.info("Start");
    Movie movie = movieService.getMovie(movieId);
    MovieDto dto = toMovieDto(movie);
    return dto;
}
```

- Include new private method.

```java
private MovieDto toMovieDto(Movie movie) {
    LOGGER.info("Start");
    MovieDto movieDto = new MovieDto();
    movieDto.setId(movie.getId());
    movieDto.setTitle(movie.getTitle());
    movieDto.setDuration(movie.getDuration());
    movieDto.setBudget(movie.getBudget());
    movieDto.setBookingsOpen(movie.isBookingsOpen());
    movieDto.setRating(movie.getRating());
    movieDto.setReleaseDate(movie.getReleaseDate());

    MpaaRatingDto mpaaRatingDto = new MpaaRatingDto();
    MpaaRating mpaaRating = movie.getMpaaRating();
    mpaaRatingDto.setId(mpaaRating.getId());
    mpaaRatingDto.setName(mpaaRating.getName());
    movieDto.setMpaaRating(mpaaRatingDto);

    List<GenreDto> genreDtoList = new ArrayList<GenreDto>();
    for (Genre genre : movie.getGenres()) {
        GenreDto genreDto = new GenreDto();
        genreDto.setId(genre.getId());
        genreDto.setName(genre.getName());
        genreDtoList.add(genreDto);
    }
    movieDto.setGenres(genreDtoList);
    LOGGER.debug("MovieDto: {}", movieDto);
    LOGGER.info("End");
    return movieDto;
}
```

- Run SpringLearnApplication and check in Postman if data is fetched from database
- Check if only one query is executed.
- Implement above log for getting all movies as well using DTO.
- There is also a library available to do this conversion. Refer link below.
  https://www.baeldung.com/entity-to-and-from-dto-for-a-java-spring-application
- To be sure that the data comes from the database modify movie title in the database and check if respective results are obtained.

Get all movies

Problem

Include a new method to get all the movies from the database.

Solution

MovieDao.java

- Include new method

```
@SuppressWarnings("unchecked")
public List<Movie> list() {
    LOGGER.info("Start");
    List<Movie> movieList = entityManager.createQuery("select m from Movie m inner join "
            + "fetch m.mpaaRating mr inner join fetch m.genres")
            .getResultList();
    return movieList;
}
```

MovieService.java

- Include method below

```
public List<Movie> list() {
    LOGGER.info("Start");
    return movieDao.list();
}
```

Runner.java

- Include method below

```
private void displayMovies() {
    LOGGER.info("Start");
    LOGGER.debug("Movie list: {}", movieService.list());
    LOGGER.info("End");
}
```

- Modify main method to display movies.

```
@Override
public void run(String... args) throws Exception {
    LOGGER.info("Start");
    //listLanguages();
    //addLanguage();
    //updateLanguage();
    //deleteLanguage();
    //findByCode("en");
    //getMovieById();
    //displayMovieData();
    //addMovie();
    //updateMovie();
    displayMovies();
    LOGGER.info("End");
}
```

- Run HiblearnApplication and check if the movies logged in the logger.
- Open the movie list into notepad and check the result. This should contain some duplicate results.
- Copy the generated SQL query into mysql client and check the result. The result as well contains duplicate entries.
- This is the expected result of the SQL query.
- There are multiple genres associated with each movie. To bring out all the mapping results, the query has to duplicate wherever there is more than one genre.

- For the movie "The Lord of the Rings", there is only one genre, hence there is only one entry returned.
- For all other movies there are two genres associated, hence we can see two entries for each one of the movies.
- To avoid this data duplication, following changes needs to be done.

MovieDao.java

- Modify the list() method as specified below:

```
@SuppressWarnings("unchecked")
public LinkedHashSet<Movie> list() {
    LOGGER.info("Start");
    List<Movie> movieList = entityManager.createQuery("select m from Movie m inner join "
            + "fetch m.mpaaRating mr inner join fetch m.genres")
            .getResultList();
    LOGGER.debug("Movie List: {}", movieList);
    return new LinkedHashSet<Movie>(movieList);
}
```

MovieService.java

- Modify list() method as specified below

```
public LinkedHashSet<Movie> list() {
    LOGGER.info("Start");
    return movieDao.list();
}
```

- Now run HiblearnApplication and check if it is getting distinct results.

Integration with REST API

- Copy list() method in MovieDao from hiblearn project to springlearn project.
- Modify list() method in MovieService. Copy code from hiblearn project to springlearn project.
- Modify getMovies() method in MovieController as specified below:

```
@GetMapping
public List<MovieDto> getMovies() {
    LOGGER.info("Start");
    Set<Movie> movies = movieService.list();
    List<MovieDto> movieList = new ArrayList<>();
    for (Movie movie : movies) {
        movieList.add(toMovieDto(movie));
    }
    return movieList;
}
```

- Test link http://localhost:8080/booking/api/movies in postman with GET method to check if all movie details are returned as JSON.

Movie getting movie details using Named Queries

Activity

Convert the following movie data retrieval using named queries.

- Get all movies
- Get a movie based on movie id

Solution

- Copy and paste the following annotations into Movie at the class level.

```
@NamedQueries({
    @NamedQuery(
        name="Movie.findAll",
        query="select m from Movie m "
            + "inner join fetch m.mpaaRating "
            + "inner join fetch m.genres"
    ),
    @NamedQuery(
        name="Movie.findById",
        query="select m from Movie m "
            + "inner join fetch m.mpaaRating "
            + "inner join fetch m.genres "
            + "where m.id = :movieId"
    )
})
public class Movie {
```

- Modify the respective MovieDao methods by copying and pasting the code below. Comment out the existing code.

```
    public List<Movie> list() {
        LOGGER.info("Start");
        return entityManager
                .createNamedQuery("Movie.findAll", Movie.class)
                .getResultList();
    }

    public Movie findById(long id) {
        LOGGER.info("Start findById");
        LOGGER.debug("Movie Id: " + id);
        return entityManager
            .createNamedQuery("Movie.findById", Movie.class)
            .setParameter("movieId", id)
            .getSingleResult();
    }
```

- Check the output of the above methods using the methods in Runner class.

Explanation

- If a project has lots of queries, then the code in DAO get difficult to manage.
- Instead we can define it in a centralized place where the model class is defined.
- The named query can be accessed using meaningful names.
- @NamedQueries annotation is used to encapsulate multiple queries
- @NamedQuery helps to associate a query to a specific name.
- In EntityManager, we need to use createNamedQuery() method instead of createQuery() to get query associated with the name.
- Notice how parameters are set using names instead of numbers.

Advantages

- Named queries are compiled when EntityManagerFactory is instantiated during the application startup. The advantage is that named queries are validated before start of the application avoiding potential errors in the HQL.
- Easier to maintain

Disadvantages

- The query is static and cannot be changed during runtime.
- A change in named query required server restart

Add new movie

Problem

Add a new movie in the database using Hibernate using the below movie details.

Title: avatar

Duration: 160

Budget: 200,000,000

Bookings Open: Yes

Rating: 2.3

Release Date: 19 Dec 2015

MPAA Rating: R

Genres: Action, Comedy

NOTE: Incorrect values are provided intentionally. The values will be corrected when updating movie.

Solution

MovieDao.java

- Include new method

```
public void add(Movie movie) {
    LOGGER.info("Start");
    LOGGER.debug("{}", movie);
    entityManager.persist(movie);
    LOGGER.info("End");
}
```

MovieService.java

- Include method below

```
public void add(Movie movie) {
    LOGGER.info("Start");
    movieDao.add(movie);
    LOGGER.info("End");
}
```

Runner.java

- Include method below:

```java
private void addMovie() throws ParseException {
    Movie movie = new Movie();
    movie.setTitle("avatar");
    movie.setBudget(200000000);
    movie.setDuration(160);
    movie.setRating(2.3f);
    SimpleDateFormat format = new SimpleDateFormat("dd/MM/yyyy");
    movie.setReleaseDate(format.parse("19/12/2015"));

    MpaaRating mpaaRating = new MpaaRating();
    mpaaRating.setId(4);
    movie.setMpaaRating(mpaaRating);

    Genre genre1 = new Genre();
    genre1.setId(1);

    Genre genre2 = new Genre();
    genre2.setId(4);

    List<Genre> genres = new ArrayList<>();
    genres.add(genre1);
    genres.add(genre2);

    movie.setGenres(genres);
    LOGGER.debug("{}", movie);
    movieService.add(movie);
}
```

- Call this method in run() method of Runner class and check if a new movie is getting added in the database.

Explanation

- Key points to note when inserting data of a complex data model.
- Notice when creating a movie object the id is not set. This indicates Hibernate that this is a new object.
- For the related MPAA Rating and Genres, we have provided only the respective primary key ids, Hibernate internally recognized them as foreign keys.
- The method persist() in EntityManager helps in saving the movie data. Notice that this method takes care of executing the following SQL statements:

```
insert into movie (mv_bookings_open, mv_budget, mv_duration, mv_mr_id, mv_rating,
mv_release_date, mv_title) values (?, ?, ?, ?, ?, ?, ?)
```

```
parameter [1] as [BOOLEAN] - [false]
parameter [2] as [BIGINT] - [200000000]
parameter [3] as [INTEGER] - [160]
parameter [4] as [BIGINT] - [4]
parameter [5] as [FLOAT] - [2.3]
parameter [6] as [TIMESTAMP] - [Sat Dec 19 00:00:00 IST 2015]
parameter [7] as [VARCHAR] - [avatar]
```

```
insert into movie_genre (mg_mv_id, mg_ge_id) values (?, ?)
```

```
parameter [1] as [BIGINT] - [18]
parameter [2] as [BIGINT] - [1]
```

```
insert into movie_genre (mg_mv_id, mg_ge_id) values (?, ?)
```

```
parameter [1] as [BIGINT] - [18]
parameter [2] as [BIGINT] - [4]
```

- Hibernate takes care of the many to many mappings and makes necessary insert in join table.

Integration with REST API

- Copy and paste the addMovie() method from MovieDao of hiblearn to springlearn
- Copy and paste the addMovie() method from MovieService of hiblearn to springlearn
- Include below method in controller:

```
@PostMapping
public void addMovie(@RequestBody Movie movie) {
    LOGGER.info("Start");
    LOGGER.debug("{}", movie);
    movieService.addMovie(movie);
    LOGGER.info("End");
}
```

- Call from Postman with the following options:
  - URL - http://localhost:8080/booking/api/movies
  - Method Type: POST
  - Method Body Type: JSON
  - Copy and paste the body text below which will

```
{
    "title": "avatar",
    "duration": 160,
    "budget": 200000000,
    "bookingsOpen": false,
    "rating": 2.3,
    "releaseDate": "18/12/2015",
    "mpaaRating": {
    "id": 4,
        "name": "R"
    },
    "genres": [
        {
            "id": 1,
            "name": "Action"
        },
        {
            "id": 4,
            "name": "Comedy"
        }
    ]
}
```

  - Check in database whether a new record is inserted into the movie table.
  - Check in movie_genre table whether the respective Genres are added.

Modify new movie

Problem

The avatar movie added earlier were having incorrect data, correct it with the right values as given below:

Title: Avatar

Duration: 162

Budget: 237,000,000

Bookings Open: No

Rating: 7.8

Release Date: 18 Dec 2009

MPAA Rating: PG-13

Genres: Action, Adventure

Solution

MovieDao.java

- Include new method

```
public void update(Movie movie) {
    LOGGER.info("Start");
    LOGGER.debug("{}", movie);
    entityManager.merge(movie);
    LOGGER.info("End");
}
```

MovieService.java

- Include method below

```
public void update(Movie movie) {
    LOGGER.info("Start");
    movieDao.update(movie);
    LOGGER.info("End");
}
```

Runner.java

- Include method below:

```
private void updateMovie() throws ParseException {
    LOGGER.info("Start");
    Movie movie = new Movie();
    movie.setId(17);
    movie.setTitle("Avatar");
    movie.setBudget(237000000);
    movie.setDuration(162);
    movie.setRating(7.8f);
    SimpleDateFormat format = new SimpleDateFormat("dd/MM/yyyy");
    movie.setReleaseDate(format.parse("18/12/2009"));

    MpaaRating mpaaRating = new MpaaRating();
    mpaaRating.setId(2);
    movie.setMpaaRating(mpaaRating);

    Genre genre1 = new Genre();
    genre1.setId(1);

    Genre genre2 = new Genre();
    genre2.setId(2);

    List<Genre> genres = new ArrayList<>();
    genres.add(genre1);
    genres.add(genre2);

    movie.setGenres(genres);
    LOGGER.debug("{}", movie);
    movieService.update(movie);
    LOGGER.info("End");
}
```

- Call this method in run() method of Runner class and check if a new movie is getting update in the database.
- Also check if the genres are removed and new genres inserted into movie_genre table.

Explanation

- Key points to note when inserting data of a complex data model.
- Notice when creating a movie object the id is set. This indicates Hibernate to modify the based on the primary key id instance variable.
- For the related MPAA Rating and Genres, we have provided only the respective primary key ids, Hibernate internally recognized them as foreign keys.
- The method merge() in EntityManager helps in updating the movie data. Notice that this method takes care of executing the following SQL statements:

```
update movie set mv_bookings_open=?, mv_budget=?, mv_duration=?, mv_mr_id=?,
mv_rating=?, mv_release_date=?, mv_title=? where mv_id=?
```

```
parameter [1] as [BOOLEAN] - [true]
parameter [2] as [BIGINT] - [200000000]
parameter [3] as [INTEGER] - [160]
parameter [4] as [BIGINT] - [4]
parameter [5] as [FLOAT] - [2.3]
parameter [6] as [TIMESTAMP] - [Fri Dec 18 05:30:00 IST 2015]
parameter [7] as [VARCHAR] - [avatar]
parameter [8] as [BIGINT] - [19]
```

```
delete from movie_genre where mg_mv_id=?
```

```
parameter [1] as [BIGINT] - [19]
```

```
insert into movie_genre (mg_mv_id, mg_ge_id) values (?, ?)
```

```
parameter [1] as [BIGINT] - [19]
```

```
parameter [2] as [BIGINT] - [1]
insert into movie_genre (mg_mv_id, mg_ge_id) values (?, ?)
parameter [1] as [BIGINT] - [19]
parameter [2] as [BIGINT] - [14]
```

- Hibernate takes care of the many to many mappings and makes necessary insert in join table.

Integration with REST API

- Copy and paste the updateMovie() method from MovieDao of hiblearn to springlearn
- Copy and paste the updateMovie() method from MovieService of hiblearn to springlearn
- Include below method in controller:

```
@PutMapping
public void updateMovie(@RequestBody Movie movie) {
    LOGGER.info("Start");
    LOGGER.debug("{}", movie);
    movieService.update(movie);
    LOGGER.info("End");
}
```

- Call from Postman with the following options:
  - URL - http://localhost:8080/booking/api/movies
  - Method Type: PUT
  - Method Body Type: JSON
  - Use the same request body used for add and modify the values with correct values for Avatar movie
  - Also include the id attribute with the respective of the 'avatar' movie created earlier. Get this from the MySQL database.
  - Check in database records are removed from the table.
- Check in movie_genre table whether the respective Genres are removed.

Remove movie

Problem

Remove the avatar movie added earlier.

Solution

- Use the exact same approach used for deleting a language for deleting a movie.
- For Spring REST API integration, the API should get the if of movie to be deleted in the URL. Then create a movie object based on the id and call the service.

Apply dynamic filtering based on Criteria API

Problem

In a web page a movie can be searched based on Genre and MPAA Rating. Write a hibernate method that accepts the Genre and MPAA Rating as input and gives the result based on that data.

Solution

- Copy and paste the below method into MovieDao

```
public List<Movie> findMovies(String mpaaRating, String genre) {
        LOGGER.info("Start");
        CriteriaBuilder builder = entityManager.getCriteriaBuilder();
        CriteriaQuery<Movie> criteria = builder.createQuery(Movie.class);
        Root<Movie> root = criteria.from(Movie.class);
        criteria.select(root);
        root.fetch("mpaaRating");
        root.fetch("genres");
        Join<Object, Object> genreJoin = root.join("genres");
        Join<Object, Object> mpaaRatingJoin = root.join("mpaaRating");
        criteria.where(builder.equal(genreJoin.get("name"), genre));
        builder.and(builder.equal(mpaaRatingJoin.get("name"), mpaaRating));
        TypedQuery<Movie> query = entityManager.createQuery(criteria);
        LOGGER.info("End");
        return query.getResultList();
}
```

- Copy and paste the below method into MovieService.

```
public List<Movie> findMovies(String mpaaRating, String genre) {
        LOGGER.info("Start");
        return movieDao.findMovies(mpaaRating, genre);
}
```

- Copy and paste the below method into Runner

```
private void displayMovieUsingCriteria() {
        LOGGER.info("Start");
        for (Movie movie : movieService.findMovies("R", "Horror")) {
                LOGGER.debug("Movie: " + movie.getTitle());
                LOGGER.debug("MPAA Rating: " + movie.getMpaaRating());
                LOGGER.debug("Genres: " + movie.getGenres());
        }
        LOGGER.info("End");
}
```

- Execute HiblearnApplication and check if "Pet Sematary" movie is returned as output. In this scenario both MPAA Rating and Genre is passed as input. Let us find the steps to implement if one value is selected, but not the other.
- Replace the below two line of code in findMovies() method:

```
criteria.where(builder.equal(genreJoin.get("name"), genre));
builder.and(builder.equal(mpaaRatingJoin.get("name"), mpaaRating));
```

as specified below:

```
if (mpaaRating != null && genre != null) {
        criteria.where(builder.equal(genreJoin.get("name"), genre));
        builder.and(builder.equal(mpaaRatingJoin.get("name"), mpaaRating));
} else if (mpaaRating != null && genre == null) {
        criteria.where(builder.equal(mpaaRatingJoin.get("name"), mpaaRating));
} else if (mpaaRating == null && genre != null) {
        criteria.where(builder.equal(genreJoin.get("name"), genre));
}
```

- In Runner class modify the respective method calls with below specified options one by one and and check the result:

```
// Option 1
movieService.findMovies("R", "Horror")
// Option 2
movieService.findMovies("R", null)
// Option 3
movieService.findMovies(null, "Horror")
```

Explanation

- Criteria API is an alternative way to define HQL or JPQL
- Criteria API helps in dynamically generating queries programmatically
- It is possible to execute queries for GROUP BY clause using Criteria API
- Steps to implement Criteria API:
    - Create CriteriaBuilder object using EntityManager
    - CriteriaQuery is used to represent a query
    - Root helps in defining the FROM clause
    - The fetch() method in root helps defining the JOIN and FETCH.
    - The Join class along with and() method from CriteriaBuilder helps in defining the filter conditions.

Get Genres using Native Query

Activity

Get all Genres using native query

Solution

- Copy and paste the below method into GenreDao

```
public List<Genre> listUsingNative() {
        LOGGER.info("Start");
        Query query = entityManager.createNativeQuery("select * from genre");
        List<Object[]> objects = query.getResultList();
        for (Object[] object : objects) {
                LOGGER.debug("Genre Id: {}", object[0]);
                LOGGER.debug("Genre Name: {}", object[1]);
        }
        LOGGER.info("End");
        return null;
}
```

- Copy and paste the below method into MovieService.

```
public List<Genre> listUsingNativeQuery() {
        LOGGER.info("Start");
        return genreDao.listUsingNative();
}
```

- Copy and paste the below method into Runner

```
private void displayGenreUsingNative() {
        LOGGER.info("Start");
        List<Genre> genres = genreService.listUsingNativeQuery();
        LOGGER.info("End");
}
```

- Modify the runner() method to call this new method.
- Execute HiblearnApplication and check if Genres are displayed.

Explanation

- Native queries help to execute actual SQL queries
- Use Native Queries when there is a database specific feature that needs to be implemented.
- For example, in Oracle database we can include hints which can improve the performance of a query.
- Hint is a specific feature related to Oracle and hence such a feature is not available in Hibernate. Using Native Query is the option available to execute these types of queries.
- The createNativeQuery() method helps in creating a native SQL.
- The getResultList() method on the native query returns List of Object Array.
- Each item in the list represents a row.
- Each row data is represented as an Object Array.
- It is also possible to map the column results with a class using @SqlResultSetMapping annotation. Refer example available in the link below:

  https://thorben-janssen.com/jpa-native-queries/
- This link also contains examples on how to implemented Named Native Queries.

Miscellaneous Topics

Batching

- When executing large number of inserts, updates or deletes, JDBC supports batch operations.
- Using JDBC batch operations a good number of statements are executed as a batch operation instead of executing each statement on by one.
- This feature of JDBC can be leverage through Hibernate.
- Refer link below that has more details

  https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#batch

Caching

- Caching is a mechanism to hold data in memory, which avoids having access to the database, ultimately it makes it possible to implement queries run much faster.
- Hibernate has caching mechanism in-built using JCache. In addition to that other third party libraries like Ehcache and Infinispan can be integrated with Hibernate.
- Refer link below for more details

  https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#caching

HQL and JPQL

- Apart from the SELECT, JOIN and FETCH query operations that we did using HQL, there are a large set of option available to implement HQL. A few items are included below for reference.
  - UPDATE Statements
  - DELETE Statements
  - INSERT Statements
  - Explicit Joins
  - Implicit Joins
  - Distinct
  - Polymorphism
  - Expressions
  - Concatenation
  - Aggregate functions
  - CASE Expression
  - Relational comparisons
  - Like predicate
  - Group by
  - Order by

  The full list can be found here:
  https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#hql

Fetching

- Eager fetching is always a bad choice.
- Prior to JPA, Hibernate used to have all associations as LAZY by default.
- After JPA, @ManyToOne and @OneToOne associations are made EAGER by default.
- The ideal option is to define HQL/JPQL query with fetch option.

Interview Questions

Hibernate

https://www.journaldev.com/3633/hibernate-interview-questions-and-answers