Università di Pisa

Computer Engineering

Large Scale and Multi-Structured Databases

TUTORIAL JPA

STUDENTS:
Cima Lorenzo
Ferrante Nicola
Pampaloni Simone
Scatena Niccolò

Academic Year 2019/2020

Contents

1 JPA: Tutorial and Implementation

JPA is a Java API specification for relational data management in applications. JPA defines a standard way for simplifying database programming defining Java Persistence Query Language (JPQL) which is an object-oriented query language that operates against Java objects rather than directly with database tables.

JPA is Java standard specification for ORM (Object Relational Mapping), a programming technique for the integration of object-oriented software with RDBMS: with JPA a class (an Entity) represents a table stored in a database, an instance of a class represents a row in the table and a member of an instance represents a column.

There are various implementations of JPA. In this tutorial the Hibernate implementation will be adopted.

In this tutorial we will show a simple implementation of JPA with Hibernate.

1.1 Configuration of JPA with Hibernate

If using Maven as dependency manager, in order to use JPA with Hibernate you must add the dependency in the POM file, in particular:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.4.7.Final</version>
</dependency>
```

Listing 1.1: Hibernate dependency for Maven POM file.

Furthermore the file `persistence.xml` must be created.

The `persistence.xml` file must be placed in the folder `src/main/resources/META-INF`. This file (shown in Listing 1.2) is a standard configuration file. Its purpose is to provide the `EntityManager` with the information necessary to save/update/query the database and configure the mapping layer. The file defines the persistence-units, groups of persistent classes with their settings, providing them with a unique identifier that can be used by the application.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" 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.prg/xml/ns/persistence/
   ↪ persistence_2.1.xsd">
  <persistence-unit name="ristogo">
    <class>ristogo.server.storage.entities.User_</class>
    <class>ristogo.server.storage.entities.Restaurant_</
   ↪ class>
    <class>ristogo.server.storage.entities.Reservation_</
   ↪ class>
    <properties>
      <property name="hibernate.dialect" value="org.
   ↪ hibernate.dialect.MySQL8Dialect" />
      <property name="javax.persistence.jdbc.url" value="
   ↪ jdbc:mysql://localhost:3306/ristogo?serverTimezone=UTC
   ↪ " />
      <property name="javax.persistence.jdbc.user" value="
   ↪ root" />
      <property name="javax.persistence.jdbc.password" value
   ↪ ="" />
      <property name="javax.persistence.jdbc.driver" value="
   ↪ com.mysql.cj.jdbc.Driver" />
      <property name="hibernate.show_sql" value="false" />
      <property name="hibernate.format_sql" value="true" />
    </properties>
  </persistence-unit>
</persistence>
```

Listing 1.2: `persistence.xml`

2 | Entities and Relationships

2.1 Entities

The first step is to define the entities, i.e. the classes that map the relational database tables.

In this case we preferred to define a super class `Entity` from which to derive the child classes `User`, `Restaurant` and `Reservation`.

```java
import javax.persistence.*;

@MappedSuperclass
public abstract class Entity_
{
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  @Column(name="id", nullable=false)
  protected int id;

  //... constructors, getters, setters ...
}
```

Listing 2.1: `Entity_.java`

Then we define the three classes that represent the tables of our database. The class structure is shown below.

```java
import java.util.*;
import javax.persistence.*;
import org.hibernate.annotations.*;

@javax.persistence.Entity
@Table(name="users")
public class User_ extends Entity_
{
  @Column(name="username", length=32,
    nullable=false, unique=true)
  protected String username;

  @Column(name="password", length=32, nullable=false)
  protected String password;

  @OneToMany(mappedBy="user", fetch=FetchType.LAZY)
  @LazyCollection(LazyCollectionOption.EXTRA)
  @Where(clause="date >= CURRENT_DATE()")
  protected List<Reservation_> activeReservations = new
   ↪ ArrayList<>();

  @OneToOne(mappedBy="owner", fetch=FetchType.LAZY)
  @LazyCollection(LazyCollectionOption.EXTRA)
  protected Restaurant_ restaurant;

  //... constructors, getters, setters ...
}
```

Listing 2.2: `User_.java`

```
import java.util.*;
import javax.persistence.*;
import org.hibernate.annotations.*;

@javax.persistence.Entity
@Table(name="restaurants")
@DynamicUpdate //(1)
public class Restaurant_ extends Entity_
{
  @OneToOne(fetch=FetchType.EAGER)
  @JoinColumn(name="ownerId", nullable=false)
  protected User_ owner;

  @Column(name="name", length=45, nullable=false)
  protected String name;

  @Enumerated(EnumType.STRING)
  @Column(name="genre", nullable=true)
  protected Genre genre;

  @Enumerated(EnumType.STRING) //(2)
  @Column(name="price", nullable=true)
  protected Price price;

  @Column(name="city", length=32, nullable=true)
  protected String city;

  @Column(name="address", length=32, nullable=true)
  protected String address;

  @Column(name="description", nullable=true)
  protected String description;

  @Column(name="seats", nullable=false)
  protected int seats;

  @Enumerated(EnumType.STRING)
  @Column(name="openingHours", nullable=false)
  protected OpeningHours openingHours;

  @OneToMany(mappedBy="restaurant", fetch=FetchType.LAZY)
  @LazyCollection(LazyCollectionOption.EXTRA)
  @Where(clause="date >= CURRENT_DATE()")
```

```java
  protected List<Reservation_> activeReservations = new
    ↪ ArrayList<>();

  //... constructors, getters, setters, methods ...
}

/* (1) @DynamicUpdate ensures that Hibernate uses only the
 *  modified columns in any SQL UPDATE statement that it
 *  generates (avoid to update unchanged fields).
 * (2) @Enumerated(EnumType.STRING) to declare that this
 *  field is an Enum type and should be mapped to an ENUM
 *  type of the DBMS (in MySQL, an enum is a string type).
 */
```

Listing 2.3: `Restaurant_.java`

```java
import java.time.LocalDate;
import javax.persistence.*;

@javax.persistence.Entity
@Table(name = "reservations")
public class Reservation_ extends Entity_
{
  @ManyToOne(fetch=FetchType.EAGER)
  @JoinColumn(name="userId", nullable=false)
  protected User_ user;

  @ManyToOne(fetch=FetchType.EAGER)
  @JoinColumn(name="restaurantId", nullable=false)
  protected Restaurant_ restaurant;

  @Column(name = "date", nullable=false)
  protected LocalDate date; //(1)

  @Enumerated(EnumType.STRING)
  @Column(name = "time", nullable=false)
  protected ReservationTime time;

  @Column(name = "seats", nullable=false)
  protected int seats;

  //... constructors, getters, setters, methods ...
}

/* (1) The LocalDate type must be converted to the SQL date
 *  type. This is done by creating an AttributeConverted,
 *  as shown below.
 */

// LocalDateAttributeConverter.java
import java.sql.Date;
import java.time.LocalDate;
import javax.persistence.*;

@Converter(autoApply=true) //(1)
```

```
public class LocalDateAttributeConverter implements
    ↪ AttributeConverter<LocalDate, Date>
{
  //(2)
  @Override
  public Date convertToDatabaseColumn(LocalDate attribute)
  {
    return attribute == null ? null : Date.valueOf(attribute
    ↪ );
  }

  @Override
  public LocalDate convertToEntityAttribute(Date dbData)
  {
    return dbData == null ? null : dbData.toLocalDate();
  }
}

/* (1) This is an Hibernate Attribute Converter and it must
 *  be auto-applied to all field of type LocalDate.
 * (2) The two defined methods implement the conversion
 *  between java.time.LocalDate and java.sql.Date
 */
```

Listing 2.4: `Reservation_.java`

2.2 Relationships

2.2.1 One-to-one Relationships

In our case a one-to-one relationship is between a user and a restaurant. In fact, a restaurant is owned by a user. So the one-to-one relationship is defined in the `User` and `Restaurant` classes as follows:

```
@OneToOne(fetch=FetchType.EAGER) //(1)
@JoinColumn(name="ownerId", nullable=false) //(2)
protected User_ owner;

/* (1) @OneToOne specifies that this is a one-to-one
 *  relationship. fetch=FetchType.EAGER means that this
 *  field should be initialized when the parent
 *  (Restaurant_) object in created.
 * (2) @JoinColumn specifies the ownerId field as the join
 *  column for Restaurant_.
 */
```

Listing 2.5: One-to-one relationship in `Restaurant_.java`

```
@OneToOne(mappedBy="owner", fetch=FetchType.LAZY) //(1)
@LazyCollection(LazyCollectionOption.EXTRA)
protected Restaurant_ restaurant;

/* (1) mappedBy defines the name of the relation field in
```

```
*  the opposite entity. fetch=FetchType.LAZY specifies
*  that this field should be initialized only when it is
*  accessed by its getter and not when the object is
*  created.
*/
```

Listing 2.6: One-to-one relationship in `User_.java`

Unlike `Restaurant`, in the annotation `@OneToOne` in the `User` class we specified the attribute `mappedBy`, in fact a bidirectional relationship brings with it the concept of "owner side" and "reverse side". Owner is the `Entity` whose report annotation (`@OneToOne`) does not specify the `mappedBy`. In essence, for a bidirectional relationship it is always the entity linked to the table that holds the relational constraint of foreign keys.

The reverse side indicates exactly this information ("I am not the Owner, look at the other the indicated entity") by inserting in the attribute mappedBy the name of the relation field in the opposite entity (owner of the Entity Restaurant).

2.2.2 One-to-many / Many-to-one Relationships

This type of relationship is usually modeled with a foreign key from one table to another so that different records in one table can reference the same record in the other. In our case we have a user or a restaurant that can make multiple reservations.

```
@OneToMany(mappedBy="user", fetch=FetchType.LAZY)
@LazyCollection(LazyCollectionOption.EXTRA)
@Where(clause="date >= CURRENT_DATE()")
protected List<Reservation_> activeReservations = new
   ↪ ArrayList<>();
```

Listing 2.7: One-to-many relationship in `User_.java`

```
@OneToMany(mappedBy="restaurant", fetch=FetchType.LAZY)
@LazyCollection(LazyCollectionOption.EXTRA)
@Where(clause="date >= CURRENT_DATE()")
protected List<Reservation_> activeReservations = new
   ↪ ArrayList<>();
```

Listing 2.8: One-to-many relationship in `Restaurant_.java`

```
@ManyToOne(fetch=FetchType.EAGER)
@JoinColumn(name="userId", nullable=false)
@Attribute(name="userId", isEntity=true)
protected User_ user;

@ManyToOne(fetch=FetchType.EAGER)
@JoinColumn(name="restaurantId", nullable=false)
@Attribute(name="restaurantId", isEntity=true)
protected Restaurant_ restaurant;
```

Listing 2.9: Many-to-one relationship in `Reservation_.java`

Also in this case we use `@JoinColumn` (with `@ManytoOne`) to specify the `Reservation`'s columns that hold the foreign keys of User and Restaurant respectively.

In the `User` and `Restaurant` classes we use `@OneToMany` with the `mappedBy` attribute to define in which field the relationship with Reservation occurs.

2.2.3 Many-to-many Relationships

There is also a `@ManyToMany` annotation that can be used to represent many-to-many relationships. This annotation is used in conjunction with the `@JoinTable` annotation that specifies the name of the table used for the relationship along with the join columns:

```
@ManyToMany
@JoinTable(name="table_name",
  joinColumns={@JoinColumn(name="column_name")},
  inverseJoinColumns={@JoinColumn(name="inverse_column_name"
   ↪ )}
// field definition
```

Listing 2.10: Many-to-many annotations

3 | EntityManager

JPA also defines an `EntityManager`, whose purpose is the runtime management of queries and transactions on persistent objects.

An `EntityManager` instance is associated with a persistence context, and it is used to interact with the database. A persistence context is a set of entity instances, which are actually the objects or instances of the model classes.

The `EntityManager` is used to manage entity instances and their life cycle, such as create entities, entities, remove entities, find and query entities.

An `EntityManager` instance can be created as described below, using `EntityManagerFactory`.

```java
import javax.persistence.EntityManagerFactory;

public abstract class EntityManager implements AutoCloseable
{
  private static EntityManagerFactory factory; //(1)
  private static final ThreadLocal<javax.persistence.
  ↪ EntityManager> threadLocal; //(2)

  static {
    threadLocal = new ThreadLocal<javax.persistence.
  ↪ EntityManager>();
  }

  public static void init(Properties properties)
  {
    factory = Persistence.createEntityManagerFactory("
  ↪ ristogo", properties); //(3)
  }

  public static void createEM() //(4)
  {
    javax.persistence.EntityManager em = factory.
  ↪ createEntityManager();
    em.setFlushMode(FlushModeType.COMMIT);
    threadLocal.set(em);
  }

  protected static javax.persistence.EntityManager getEM()
  ↪ //(5)
  {
    return threadLocal.get();
  }

  public void close() //(6)
  {
    closeEM();
  }

  public static void closeFactory() //(7)
  {
    if (getLevelDBManager() != null)
      getLevelDBManager().close();
    if (factory != null)
      factory.close();
  }

  //... methods for CRUD operations ...
}

/* (1) The EntityManagerFactory is used to create the
 *  EntityManager instance.
 * (2) There is an EntityManager instance for each server
 *  thread.
 * (3) Creates the EntityManagerFactory.
 * (4) Creates the EntityManager
 * (5) Returns the EntityManager for the current running
 *  thread (if needed, the method creates the instance of
 *  EntityManager via the createEntityManager method.
 * (6) Closes the EntityManager instance.
 * (7) Closes the EntityManagerFactory.
```

Listing 3.1: `EntityManager.java`

The class shown in Listing 3.1 is a wrapper class for the Hibernate's `EntityManager`. In our example we have created three `EntityManager`'s child classes for each entity of our system: `UserManager`, `RestaurantManager` and `ReservationManager`.

4 | CRUD Operations

Whenever you want to manipulate data with JPA within an `EntityManager` you must perform the following operations (except read operation, that does not require to open a transaction):

1. Begin a transaction.
2. Manage entity instances (create, update, remove etc).
3. Commit the transaction.

How to begin and commit a transition is shown below:

```java
public static void beginTransaction()
{
  createEM();
  EntityTransaction tx = getEM().getTransaction(); //(1)
  if (tx != null && !tx.isActive())
    getEM().getTransaction().begin(); //(2)
}

/* (1) getEM() returns the EntityManager instance for the
 *  current thread. getTransaction() gets the current
 *  open transaction (if any).
 * (2) Starts a transaction (if not already active).
 */
```

Listing 4.1: Begin a transaction.

```java
public static void commitTransaction()
{
  EntityTransaction tx = getEM().getTransaction();
  if (tx != null && tx.isActive())
    getEM().getTransaction().commit(); //(1)
  if (isLevelDBEnabled())
    getLevelDBManager().commitBatch();
  closeEM();
}

/* (1) Commits the transaction. */
```

Listing 4.2: Commit a transaction.

In the following listings are shown the four CRUD operations applied to the entities defined above:

```java
public void insert(Entity_ entity)
{
  try {
    beginTransaction();
    persist(entity);
    commitTransaction();
  } catch (PersistenceException ex) {
    rollbackTransaction(); //(1)
    throw ex;
  }
}

public void persist(Entity_ entity)
{
  getEM().persist(entity); //(2)
}

/* (1) Rollback transaction if an error occured.
 * (2) Store the entity in the database.
 */
```

Listing 4.3: Create

```java
public Entity_ get(Class<? extends Entity_> entityClass, int
    ↪  entityId) //(1)
{
  return getEM().find(entityClass, entityId); //(2)
}

/* (1) We use entityClass to derive what kind of entity is
 *  being used (User_, Restaurant_ or Reservation_).
 * (2) find() permit to find and retrieve an entity from the
 *  database using its primary key (entityId).
 */
```

Listing 4.4: Read

```
public void update(Entity_ entity)
{
  try {
    beginTransaction();
    merge(entity);
    commitTransaction();
  } catch (PersistenceException ex) {
    rollbackTransaction();
    throw ex;
  }
}

public void merge(Entity_ entity)
{
  getEM().merge(entity); //(1)
}

/* (1) Merge modification of the entity with the entity
 * stored in the db.
 */
```

Listing 4.5: Update

```
public void delete(Class<? extends Entity_> entityClass, int
   ↪  entityId)
{
  try {
    beginTransaction();
    Entity_ entity = getEM().getReference(entityClass,
   ↪ entityId); //(1)
    remove(entity);
    commitTransaction();
  } catch (PersistenceException ex) {
    rollbackTransaction();
    throw ex;
  }
}

public void remove(Entity_ entity)
{
  getEM().remove(entity); //(2)
}

/* (1) Returns a reference to an entity. The entity is found
 *  by its primary key (entityId).
 * (2) Removes the entity from the database.
 */
```

Listing 4.6: Delete

5 | Build Custom Queries

JPA provides **Criteria API**, an alternative way for defining JPA queries, which is mainly useful for building dynamic queries with user-provided parameters. In particular, we will see `CriteriaBuilder`.

`CriteriaBuilder` is used to construct criteria queries, compound selections, expressions, predicates, orderings. An example of use is shown below considering the `User` entity of this tutorial. In particular, we want to find a user by searching for him through his `username` (and not through his `Id` as the `find` method allows).

```java
public class UserManager extends EntityManager
{
  public User_ getUserByUsername(String username)
  {
    createEM();
    javax.persistence.EntityManager em = getEM();
    CriteriaBuilder cb = em.getCriteriaBuilder(); //(1)
    CriteriaQuery<User_> cq = cb.createQuery(User_.class); //(2)
    Root<User_> from = cq.from(User_.class); //(3)
    CriteriaQuery<User_> select = cq.select(from);
    ParameterExpression<String> usernamePar = cb.parameter(String.class); //(4)
    select.where(cb.equal(from.get("username"), usernamePar)); //(5)
    TypedQuery<User_> query = em.createQuery(cq); //(6)
    query.setParameter(usernamePar, username); //(7)
    User_ user;
    try {
      user = query.getSingleResult(); //(8)
    } catch (NoResultException ex) {
      user = null;
    } finally {
      closeEM();
    }
    return user;
  }

  //... other methods ...
}

/* (1) Returns an instance of CriteriaBuilder for the
 *  creation of a criteria query.
 * (2) Create a CriteriaQuery object for the specified
 *  entity type (User_).
 * (3) Defines from which table we want to fetch data (The
 *  table associated to the User_ entity).
 * (4) Defines a parameter of type String. The value of the
 *  parameter will be inserted in the built query.
 * (5) Defines the where clause of the query. cb.equal()
 *  creates an equality constraint (predicate) between
 *  the username column (from.get("username") and the
 *  username parameter.
 * (6) Creates the final query. TypedQuery are a special
 *  kind of query that may contains parameters.
 * (7) Bind the value (username) with its parameter
 *  (usernamePar).
 * (8) Executes the query and returns a single result. To
 *  obtain the entire result set, getResultList() can be
 *  used.
 */
```

Listing 5.1: `CriteriaBuilder` example.

6 Examples

In this chapter we will show some most used methods of our example application which uses JPA.

- Example 1

```java
private ResponseMessage handleLogin(RequestMessage reqMsg)
{
  User user = (User)reqMsg.getEntity();
  if (!user.hasValidUsername() || !user.hasValidPassword())
    return new ResponseMessage("Invalid username or password.");
  User_ savedUser = userManager.getUserByUsername(user.getUsername()); //Listing 20: CriterisBuilder example
  if (savedUser == null || !user.checkPasswordHash(savedUser.getPassword()))
    return new ResponseMessage("Invalid username or password.");
  loggedUser = savedUser;
  return new ResponseMessage(loggedUser.toCommonEntity((restaurantManager.getRestaurantByOwner(loggedUser) == null) ? UserType.CUSTOMER : UserType.OWNER));
}
```

Listing 6.1: `handleLogin` example.

- Example 2

```java
private ResponseMessage handleDeleteReservation(RequestMessage reqMsg)
{
  Reservation reservation = (Reservation)reqMsg.getEntity();
  if (!hasReservation(loggedUser, reservation.getId()))
    return new ResponseMessage("You can only delete your own reservations.");
  Reservation_ reservation_ = reservationManager.get(reservation.getId());
  if (reservation_ == null)
    return new ResponseMessage("Can not find the specified reservation.");
  try {
    reservationManager.delete(reservation.getId()); // Listing 19: Delete
  } catch (PersistenceException ex) {
    return new ResponseMessage("Error while deleting the reservation from the database.");
  }
  return new ResponseMessage();
}
```

Listing 6.2: `handleDeleteReservation` example.

- Example 3

```java
private ResponseMessage handleEditRestaurant(RequestMessage reqMsg)
{
  Restaurant restaurant = (Restaurant)reqMsg.getEntity();
  if (!hasRestaurant(loggedUser, restaurant.getId()))
    return new ResponseMessage("You can only edit restaurants that you own.");
  Restaurant_ restaurant_ = restaurantManager.get(restaurant.getId());
  if (!restaurant_.merge(restaurant))
    return new ResponseMessage("Some restaurant's fields are invalid.");
  restaurant_.setOwner(loggedUser);
  try {
    restaurantManager.update(restaurant_); // Listing 18: Update
  } catch (PersistenceException ex) {
    return new ResponseMessage("Error while saving the restaurant to the database.");
  }
  restaurantManager.refresh(restaurant_);
  return new ResponseMessage(restaurant_.toCommonEntity());
}
```

Listing 6.3: `handleEditRestaurant` example.