Guide to the Hibernate EntityManager

Last modified: December 13, 2018 by baeldung

1. Introduction

***EntityManager* is a part of the Java Persistence API. Chiefly, it implements the programming interfaces and lifecycle rules defined by the JPA 2.0 specification.**

Moreover, we can access the Persistence Context, by using the APIs in *EntityManager*.

In this tutorial, we'll look at the configuration, types, and various APIs of the *EntityManager*.

2. Maven Dependencies

First, we need to include the dependencies of Hibernate:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.4.0.Final</version>
</dependency>
```

We will also have to include the driver dependencies, depending upon the database that we're using:

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.13</version>
</dependency>
```

The hibernate-core and mysql-connector-java dependencies are available on Maven Central.

3. Configuration

Now, let's demonstrate the *EntityManager*, by using a *Movie* entity which corresponds to a MOVIE table in the database.

Over the course of this article, we'll make use of the *EntityManager* API to work with the *Movie* objects in the database.

3.1. Defining the Entity

Let's start by creating the entity corresponding to the MOVIE table, using the *@Entity* annotation:

```
1    @Entity
2    @Table(name = "MOVIE")
3    public class Movie {
4
5      @Id
6      private Long id;
7
8      private String movieName;
9
10     private Integer releaseYear;
11
12     private String language;
13
14     // standard constructor, getters, setters
15   }
```

3.2. The *persistence.xml* File

When the *EntityManagerFactory* is created, **the persistence implementation searches for the *META-INF/persistence.xml* file in the classpath**.

This file contains the configuration for the *EntityManager*:

```
1  <persistence-unit name="com.baeldung.movie_catalog">
2    <description>Hibernate EntityManager Demo</description>
3    <class>com.baeldung.hibernate.pojo.Movie</class>
4    <exclude-unlisted-classes>true</exclude-unlisted-classes>
5    <properties>
6      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5Dialect"/>
7      <property name="hibernate.hbm2ddl.auto" value="update"/>
8      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
9      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://127.0.0.1:3306/moviecatalog"/>
10     <property name="javax.persistence.jdbc.user" value="root"/>
11     <property name="javax.persistence.jdbc.password" value="root"/>
12   </properties>
13 </persistence-unit>
```

To explain, we define the persistence-unit that specifies the underlying datastore managed by the *EntityManager*.

Furthermore, we define the dialect and the other JDBC properties of the underlying datastore. Hibernate is database-agnostic. **Based on these properties, Hibernate connects with the underlying database.**

4. Container and Application Managed *EntityManager*

Basically, **there are two types of *EntityManager* – Container Managed and Application Managed.**

Let's have a closer look at each type.

4.1. Container Managed *EntityManager*

Here, the container injects the *EntityManager* in our enterprise components.
In other words, the container creates the *EntityManager* from the *EntityManagerFactory* for us:

```
1    @PersistenceContext
2    EntityManager entityManager;
```

This also means **the container is in charge of beginning, committing, or rolling back the transaction.**

4.2. Application Managed *EntityManager*

Conversely, the lifecycle of the *EntityManager* is managed by the application in here.

In fact, we'll manually create the *EntityManager.* Furthermore, we'll also manage the lifecycle of the *EntityManager* we've created.

First, let's create the *EntityManagerFactory:*

```
1    EntityManagerFactory emf = Persistence.createEntityManagerFactory("com.baeldung.movie_catalog");
```

In order to create an *EntityManager*, we must explicitly call *createEntityManager()* in the *EntityManagerFactory*:

```
1    public static EntityManager getEntityManager() {
2       return emf.createEntityManager();
3    }
```

5. Hibernate Entity Operations

The *EntityManager* API provides a collection of methods. We can interact with the database, by making use of these methods.

5.1. Persisting Entities

In order to have an object associated with the EntityManager, we can make use of the *persist()* method :

```
1
2    public void saveMovie() {
3        EntityManager em = getEntityManager();
4        em.getTransaction().begin();
5        Movie movie = new Movie();
6        movie.setId(1L);
7        movie.setMovieName("The Godfather");
8        movie.setReleaseYear(1972);
9        movie.setLanguage("English");
10       em.persist(movie);
11       em.getTransaction().commit();
12   }
```

Once the object is saved in the database, it is in the *persistent* state.

5.2. Loading Entities

For the purpose of retrieving an object from the database, we can use the *find()* method.
Here, the method searches by primary key. In fact, the method expects the entity class type and the primary key:

```
1    public Movie getMovie(Long movieId) {
2       EntityManager em = getEntityManager();
3       Movie movie = em.find(Movie.class, new Long(movieId));
4       em.detach(movie);
5       return movie;
6    }
```

However, if we just need the reference to the entity, we can use the *getReference()* method instead. In effect, it returns a proxy to the entity:

```
1     Movie movieRef = em.getReference(Movie.class, new Long(movieId));
```

5.3. Detaching Entities

In the event that we need to detach an entity from the persistence context, **we can use the *detach()* method**. We pass the object to be detached as the parameter to the method:

```
1     em.detach(movie);
```

Once the entity is detached from the persistence context, it will be in the detached state.

5.4. Merging Entities

In practice, many applications require entity modification across multiple transactions. For example, we may want to retrieve an entity in one transaction for rendering to the UI. Then, another transaction will bring in the changes made in the UI.

We can make use of the *merge()* method, for such situations. **The merge method helps to bring in the modifications made to the detached entity, in the managed entity, if any:**

```
1   public void mergeMovie() {
2      EntityManager em = getEntityManager();
3      Movie movie = getMovie(1L);
4      em.detach(movie);
5      movie.setLanguage("Italian");
6      em.getTransaction().begin();
7      em.merge(movie);
8      em.getTransaction().commit();
9   }
```

5.5. Querying for Entities

Furthermore, we can make use of JPQL to query for entities. We'll invoke *getResultList()* to execute them.

Of course, we can use the *getSingleResult(),* if the query returns just a single object:

```
1   public List<?> queryForMovies() {
2       EntityManager em = getEntityManager();
3       List<?> movies = em.createQuery("SELECT movie from Movie movie where movie.language = ?1")
4         .setParameter(1, "English")
5         .getResultList();
6       return movies;
7   }
```

5.6. Removing Entities

Additionally, **we can remove an entity from the database using the *remove()* method**. It's important to note that, the object is not detached, but removed.

Here, the state of the entity changes from persistent to new:

```
1   public void removeMovie() {
2       EntityManager em = HibernateOperations.getEntityManager();
3       em.getTransaction().begin();
4       Movie movie = em.find(Movie.class, new Long(1L));
5       em.remove(movie);
6       em.getTransaction().commit();
7   }
```

6. Conclusion

In this article, we have explored the *EntityManager* in *Hibernate*. We've looked at the types and configuration, and we learned about the various methods available in the API for working with the *persistence context.*

As always, the code used in the article is available over at Github.