Hibernate Tutorial

<https://examples.javacodegeeks.com/enterprise-java/hibernate/hibernate-one-one-example/>

SessionFactory is a factory class for Session objects. It is available for the whole application while a Session is only available for particular transaction.

12

**SessionFactory** is an interface. SessionFactory can be created by providing Configuration object, which will contain all DB related property details pulled from either hibernate.cfg.xml file or hibernate.properties file. SessionFactory is a factory for Session objects.

We can create one SessionFactory implementation per database in any application. If your application is referring to multiple databases, then you need to create one SessionFactory per database.

The SessionFactory is a heavyweight object; it is usually created during application start up and kept for later use. The SessionFactory is a thread safe object and used by all the threads of an application.

**A Session** is used to get a physical connection with a database. The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database. Persistent objects are saved and retrieved through a Session object.

The session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed them as needed. The main function of the Session is to offer, create, read, and delete operations for instances of mapped entity classes.

The main difference between the first level and second level cache in Hibernate is that **the first level is maintained at the Session level and accessible only to the Session, while the second level cache is maintained at the SessionFactory level and available to all Sessions**.

<https://www.baeldung.com/hibernate-5-bootstrapping-api>

<https://www.javaguides.net/2019/03/jsp-servlet-hibernate-crud-example.html>

JoinColumn vs mappedBy

https://www.baeldung.com/jpa-joincolumn-vs-mappedby

Overview of JPA/Hibernate Cascade Types

## What Is Cascading?

Entity relationships often depend on the existence of another entity, for example the Person–Address relationship. Without the Person, the Address entity doesn't have any meaning of its own. When we delete the Person entity, our Address entity should also get deleted.

Cascading is the way to achieve this. **When we perform some action on the target entity, the same action will be applied to the associated entity.**

### PA Cascade Type

All JPA-specific cascade operations are represented by the javax.persistence.CascadeType enum containing entries:

* ALL
* PERSIST
* MERGE
* REMOVE
* REFRESH
* DETACH

https://www.viralpatel.net/hibernate-many-to-many-annotation-mapping-tutorial/

Many to Many

**@ManyToMany** – Is used to create many-to-many relationship between Employee and Meeting entities. If the Collection is defined using generics to specify the element type, the associated target entity class does not need to be specified; otherwise it must be specified. Every many-to-many association has two sides, the owning side and the non-owning, or inverse, side. The join table is specified on the owning side. If the association is bidirectional, either side may be designated as the owning side.

Note that in above entity classes, Employee is defined as relationship owner as @JoinColumn is define in Employee class and mappedBy is specified in Meeting class.

**@JoinTable** – Is used to define the join table (link table) for many-to-many relationship. It is specified on the owning side of a many-to-many association, or in a unidirectional one-to-many association. In this case the join table is EMPLOYEE\_MEETING.

If the JoinTable annotation is missing, the default values of the annotation elements apply. The name of the join table is assumed to be the table names of the associated primary tables concatenated together (owning side first) using an underscore.

**@JoinColumn** – Is used to define the join column (linking column) in both main tables.

Note that we have used SET to map meetings with employee and vice versa. A <set> is similar to except that it can only store unique objects. That means no duplicate elements can be contained in a set. When you add the same element to a set for second time, it will replace the old one. A set is unordered by default but we can ask it to be sorted. The corresponding type of a <set> in Java is java.util.Set.

<https://stackabuse.com/a-guide-to-jpa-with-hibernate-relationship-mapping/>

<https://www.javaguides.net/2019/03/jsp-servlet-hibernate-crud-example.html>

<https://www.baeldung.com/jdbc-resultset>

## Navigating the ResultSet

When we obtain a ResultSet, the position of the cursor is before the first row. Moreover, by default, the ResultSet moves only in the forward direction. But, we can use a scrollable ResultSet for other navigation options.

In this section, we'll discuss the various navigation options.

### 4.1. ResultSet Types

ResultSet type indicates how we'll steer through the dataset:

* TYPE\_FORWARD\_ONLY – the default option, in which the cursor moves from start to end
* TYPE\_SCROLL\_INSENSITIVE – our cursor can move through the dataset in both forward and backward directions; if there are changes to the underlying data while moving through the dataset, they are ignored; the dataset contains the data from the time the database query returns the result
* TYPE\_SCROLL\_SENSITIVE – similar to the scroll insensitive type, however for this type, the dataset immediately reflects any changes to the underlying data

## Updating Data in a ResultSet

By default, the ResultSet is read-only. However, we can use an updatable ResultSet to insert, update, and delete the rows.

### ResultSet Concurrency

The concurrency mode indicates if our ResultSet can update the data.

The CONCUR\_READ\_ONLY option is the default and should be used if we don't need to update the data using our ResultSet.

However, if we need to update the data in our ResultSet, then the CONCUR\_UPDATABLE option should be used.

**Not all databases support all the concurrency modes for all ResultSet types**. Therefore, we need to check if our desired type and concurrency mode are supported using the supportsResultSetConcurrency() method:

**DatabaseMetaData** dbmd = dbConnection.getMetaData();

**boolean** isSupported = dbmd.supportsResultSetConcurrency(

ResultSet.TYPE\_SCROLL\_SENSITIVE, ResultSet.CONCUR\_UPDATABLE);

Copy

### 5.2. Obtaining an Updatable ResultSet

To obtain an updatable ResultSet, we need to pass an additional parameter when we prepare the Statement. For that, let’s use CONCUR\_UPDATABLE as the third parameter while creating a statement:

### Updating a Row

In this section, we'll update a row using the updatable ResultSet created in the previous section.

We can update data in a row by calling updateX() methods, passing the column names and values to update. We can use any supported data type in place of X in the updateX() method.

Let's update the “salary” column, which is of type double:

rs.updateDouble("salary", 1100.0);Copy

Note that this just updates the data in the ResultSet, but the modifications are not yet saved back to the database.

Finally, let’s call updateRow() to **save the updates to the database**:

rs.updateRow();

Copy

Instead of the column names, we can pass the column index to the updateX() methods. This is similar to using the column index for getting the values using getX() methods. Passing either the column name or index to the updateX() methods yields the same result:

rs.updateDouble(4, 1100.0);

rs.updateRow();

Copy

### 5.4. Inserting a Row

Now, let's insert a new row using our updatable ResultSet.

First, we'll use moveToInsertRow() to move the cursor to insert a new row:

rs.moveToInsertRow();Copy

Next, we must call updateX() methods to add the information to the row. We need to provide data to all the columns in the database table. If we don't provide data to every column, then the default column value is used:

rs.updateString("name", "Venkat");

rs.updateString("position", "DBA");

rs.updateDouble("salary", 925.0);Copy

Then, let's call insertRow() to insert a new row into the database:

rs.insertRow();Copy

Finally, let's use moveToCurrentRow(). This will take the cursor position back to the row we were at before we started inserting a new row using the moveToInsertRow() method:

rs.moveToCurrentRow();