**Hibernate Tutorial**

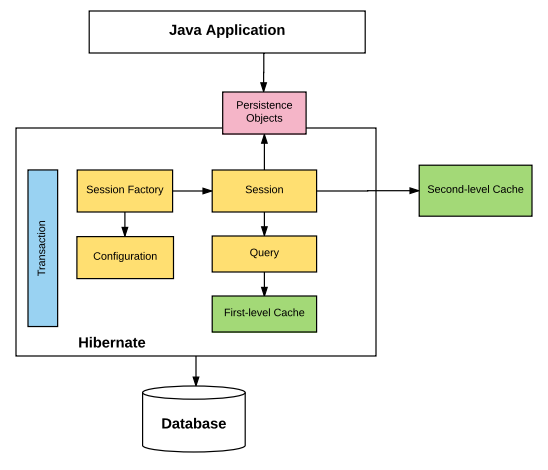
Hibernate is an open source **Java persistence (data transaction)** framework project. It performs powerful object-relational mapping and query databases using HQL (hibernate query language) and SQL.

Hibernate is a great tool for **ORM (Object Relationship Mapping) mappings** in Java. It can cut down a lot of complexity and thus defects as well from your application, which may otherwise find a way to exist. This is specially boon for developers with limited knowledge of SQL.

Initially started as an ORM framework, Hibernate has spun off into many projects, such as *Hibernate Search*, *Hibernate Validator*, *Hibernate OGM* (for NoSQL databases), and so on.

**Hibernate Architecture**

The following diagram summarizes the main building blocks in hibernate architecture.



**Let’s understand what each block represents.**

1. **Configuration** : Generally written in hibernate.properties or hibernate.cfg.xml files. For Java configuration, you may find class annotated with @Configuration. It is used by Session Factory to work with Java Application and the Database. It represents an entire set of mappings of an application Java Types to an SQL database.
2. **Session Factory** : Any user application requests Session Factory for a session object. Session Factory uses configuration information from above listed files, to instantiates the session object appropriately.
3. **Session** : This represents the interaction between the application and the database at any point of time. This is represented by the org.hibernate.Session class. The instance of a session can be retrieved from the SessionFactory bean.
4. **Query** : It allows applications to query the database for one or more stored objects. Hibernate provides different techniques to query database, including NamedQuery and Criteria API.
5. **First-level cache** : It represents the default cache used by Hibernate Session object while interacting with the database. It is also called as session cache and caches objects within the current session. All requests from the Session object to the database must pass through the first-level cache or session cache. One must note that the first-level cache is available with the session object until the Session object is live.
6. **Transaction** : enables you to achieve data consistency, and rollback incase something goes unexpected.
7. **Persistent objects** : These are plain old Java objects (POJOs), which get persisted as one of the rows in the related table in the database by hibernate.They can be configured in configurations files (hibernate.cfg.xml or hibernate.properties) or annotated with @Entity annotation.
8. **Second-level cache** : It is used to store objects across sessions. This needs to be explicitly enabled and one would be required to provide the cache provider for a second-level cache. One of the common second-level cache providers is EhCache.

## Salient features of the Hibernate framework

#### Object/Relational Mapping

Hibernate, as an ORM framework, allows the mapping of the Java domain object with database tables and vice versa. As a result, business logic is able to access and manipulate database entities via Java objects. It helps to speed up the overall development process by taking care of aspects such as transaction management, automatic primary key generation, managing database connections and related implementations, and so on.

#### JPA provider

Hibernate does support the **Java Persistence API (JPA)** specification. JPA is a set of specifications for accessing, persisting, and managing data between Java objects and relational database entities.

#### Idiomatic persistence

Any class that follows object-oriented principles such as inheritance, polymorphism, and so on, can be used as a persistent class.

#### High performance and scalability

Hibernate supports techniques such as different fetching strategies, lazy initialization, optimistic locking, and so on, to achieve high performance, and it scales well in any environment.

#### Easy to maintain

Hibernate is easier to maintain as it requires no special database tables or fields. It generates SQL at system initialization time. It is much quicker and easier to maintain compared to JDBC.

## Hello world application

#### ****Hibernate 3 introduction and writing hello world application****

In this notes, I will try to detail out more information on hibernate and then will identify the basic steps to use hibernate for our first running **java hibernate example** application.

## Basic concepts

#### How to build SessionFactory in hibernate 4

If you have been watching previous hibernate releases then you must have noticed that they have deprecated a lot of classes in quick succession. Deprecated classes are AnnotationConfiguration, ServiceRegistryBuilder and so on.

Example of building hibernate SessionFactory without using deprecated classes mentioned above. I am using the **latest hibernate version** i.e. Hibernate 4.3.6.Final, so you can make sure that you are using the latest approach for building session factory.

#### Entities Equality and Identity Concepts

Many times in our application, we face a situation where we have to compare two objects to check their equality for satisfying some business rules. In core java, we have already much knowledge about checking equality of objects, but in hibernate, we need to take care of a few extra things as well. Let’s learn what are those extra concepts.

#### Defining Association Mappings between Hibernate Entities

When we annotate the java classes with JPA annotations and make them persistent entities, we can face situations where two entities can be related and must be referenced from each other, in either uni-direction or in bi-direction. Let’s understand a few basic things before actually creating references between hibernate entities.

#### Entity / Persistence LifeCycle States Concepts

Given an instance of an object that is mapped to Hibernate, it can be in any one of four different states: transient, persistent, detached, or removed.

#### Using In-memory Database With Hibernate

In this example, I am using HSQLDB Database for creating and accessing in-memory database through our hibernate code.

#### Hibernate JPA Cascade Types

To enable **cascade and inverse** effect, we had used “CascadeType” attribute in entities. In this tutorial, we will learn about various type of available options for cascading via CascadeType.

#### Pros and Cons of Hibernate Annotations Vs Mappings

As you may know that prior to the inline annotations, the only way to create hibernate mappings was through XML files. Although various tools from Hibernate and third-party projects allowed part or all of these mappings to be generated from Java source code automatically. Today annotations are the newest way to define mappings but it is not automatically the best way to do so.

#### Hibernate Query Language [HQL]

HQL is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties. It is a superset of the JPQL, the Java Persistence Query Language; a JPQL query is a valid HQL query, but not all HQL queries are valid JPQL queries. HQL is a language with its own syntax and grammar.

Let’s learn HQL using the following examples:

1. Basic HQL Syntax
   * Update Operation
   * Delete Operation
   * Insert Operation
   * Select Operation
2. The from Clause and Aliases
3. The select Clause and Projection
4. Using Named Parameters
5. Paging Through the Result Set
6. Obtaining a Unique Result
7. Sorting Results with the order by Clause
8. Associations
9. Aggregate Methods
10. Named Queries
11. Using Native SQL
12. Enable Logging and Commenting

#### Hibernate Criteria Queries

The Criteria Query API lets you build nested, structured query expressions in Java, providing a compile-time syntax checking that is not possible with a query language like HQL or SQL. The Criteria API also includes query by example (QBE) functionality. This lets you supply example objects that contain the properties you would like to retrieve instead of having to step-by-step spell out the components of the query. It also includes projection and aggregation methods, including count(). Let’s explore it’s different features in detail.

1. Basic Usage Example
2. Using Restrictions with Criteria
3. Paging Through the Result Set
4. Obtaining a Unique Result
5. Obtaining Distinct Results
6. Sorting the Query’s Results
7. Performing Associations (Joins)
8. Adding Projections
9. Query By Example (QBE)
10. Summary

#### Lazy Loading in Hibernate

In this this tutorial, I will be discussing a must-known feature in hibernate known as lazy loading. This is useful specially if you working in a very large application.

## CRUD Operation Examples

#### ****Hello world insert data****

In this tutorial, I am giving example of inserting data in a single table.

#### ****Hibernate named query tutorial****

Named queries in hibernate is a technique to group the HQL statements in single location, and lately refer them by some name whenever need to use them. It helps largely in code cleanup because these HQL statements are no longer scattered in whole code.

#### Loading entity from database example

Examples of loading an hibernate entity using either load or get method.

#### Save() and saveOrUpdate() for Saving Entities

Please not that creating an instance of a class, you mapped with a hibernate annotations, does not automatically persist the object to the database. It must be save explicitly after attaching it to a valid hibernate session. Let’s learn how to do it.

#### Merging and Refreshing Entities

I am discussing few thoughts around refresh() and merge() method present in hibernate session class.

## Hibernate entity mappings

#### ****Hibernate one to one mapping using annotations****

Let’s discuss variations of one-to-one mappings supported in hibernate:

1. Using foreign key association
2. Using a common join table
3. Using shared primary key

#### ****Hibernate one to many mapping using annotations****

Discuss variations of one-to-many mappings supported in hibernate:

1. Using foreign key association
2. Using a join table

#### ****Hibernate many to many mapping using annotations****

Discuss variations of many-to-many mappings supported in hibernate.

## Hibernate Connection Pooling and Caching

#### C3P0 Connection Pool Configuration Tutorial

By default, Hibernate uses JDBC connections in order to interact with a database. Creating these connections is expensive—probably the most expensive single operation Hibernate will execute in a typical-use case. Since JDBC connection management is so expensive that possibly you will advise to use a pool of connections, which can open connections ahead of time (and close them only when needed, as opposed to “when they’re no longer used”).

C3P0 is an example of an external connection pool. In this tutorial, we will learn to use it with hibernate.

#### ****Hibernate EhCache configuration****

Caching is facility provided by ORM frameworks which help users to get fast running web application, while help framework itself to reduce the number of queries made to the database in a single transaction. Hibernate also provide this caching functionality at first level and second level.

In this tutorial, I am giving an example using ehcache configuration as second level cache in hibernate.

#### ****Hibernate first level cache with example****

Fist level cache in hibernate is enabled by default and you do not need to do anything to get this functionality working. Let’s learn more about it.

#### ****Hibernate second level cache with example****

In this tutorial, I am giving concepts around hibernate second level cache and give example using code snippets.

## Hibernate best practices

#### ****Hibernate @NaturalId usage and example****

Hibernate 4 has bring lots of improvements and @NaturalId is one of such nice improvements. As you know @Id annotation is used as meta data for specifying the primary key of an entity. But sometimes, entity is usually used in DAO layer code with id which not not primary key but its logical or natural id. In such cases, @NaturalId annotation will prove good replacement of named queries in hibernate.

#### ****Get entity reference for lazy loading****

Lazy loading is a design pattern commonly used in computer programming to defer initialization of an object until the point at which it is needed. Hibernate lazy loading can be done by specifying “fetch= FetchType.LAZY” in hibernate mapping annotations.

**Hibernate Entity Relationship or Associates:**

* 1. One to one : example (one employee can have one salary)
  2. One to many : example (one employee can be multiple attendance)
  3. Many to many : example (multiple customer can buy multiple product)

# Hibernate One to One Mapping Annotation Example

**Various supported techniques**

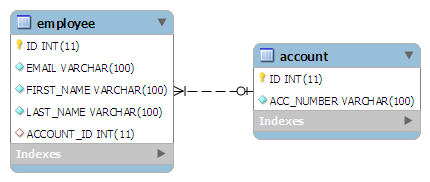
In hibernate there are 3 ways to create one-to-one relationships between two entities (class). Either way we have to use **@OneToOne** annotation.

1. First technique is most widely used and uses a **foreign key column in one of the tables**.
2. Second technique uses a rather known solution of having a **third table to store mapping** between first two tables.
3. Third technique is something new which uses a **common primary key value in both the tables**.

**1. Hibernate one to one mapping with foreign key association**

In this kind of association, a foreign key column is created in **owner entity**. For example, if we make **EmployeeEntity** owner, then a extra column "ACCOUNT\_ID" will be created in Employee table. This column will store the foreign key for Account table.

Table structure will be like this:



To make such association, refer the Account entity in EmployeeEntity class as follow:

|  |
| --- |
| EmployeeEntity.java |
| @OneToOne  @JoinColumn(name="ACCOUNT\_ID")  private AccountEntity account; |

The join column is declared with the **@JoinColumn** annotation which looks like the **@Column** annotation. It has one more parameters named referencedColumnName. This parameter declares the column in the targeted entity that will be used to the join.

If no @JoinColumn is declared on the owner side, the defaults apply. A join column(s) will be created in the owner table and its name will be the concatenation of the name of the relationship in the owner side, \_ (underscore), and the name of the primary key column(s) in the owned side.

In a bidirectional relationship, one of the sides (and only one) has to be the owner. The owner is responsible for the association column(s) update. *To declare a side as not responsible for the relationship, the attribute****mappedBy****is used*. ‘mappedBy’ refers to the property name of the association on the owner side.

|  |
| --- |
| AccountEntity.java |
| @OneToOne(mappedBy="account")  private EmployeeEntity employee; |

Above “mappedBy” attribute declares that it is dependent on owner entity for mapping.

Lets test above mappings in running code:

|  |
| --- |
| TestForeignKeyAssociation.java |
| public class TestForeignKeyAssociation {        public static void main(String[] args) {        Session session = HibernateUtil.getSessionFactory().openSession();          session.beginTransaction();            AccountEntity account = new AccountEntity();          account.setAccountNumber("123-345-65454");            // Add new Employee object          EmployeeEntity emp = new EmployeeEntity();          emp.setEmail("demo-user@mail.com");          emp.setFirstName("demo");          emp.setLastName("user");            // Save Account          session.saveOrUpdate(account);          // Save Employee          emp.setAccount(account);          session.saveOrUpdate(emp);            session.getTransaction().commit();          HibernateUtil.shutdown();      }  } |

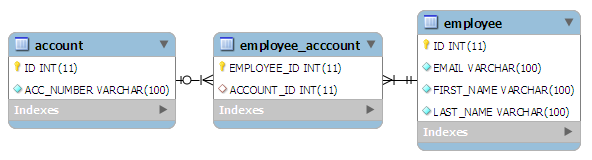
Running above code creates desired schema in database and run these SQL queries.

|  |
| --- |
| Console |
| Hibernate: insert into ACCOUNT (ACC\_NUMBER) values (?)  Hibernate: insert into Employee (ACCOUNT\_ID, EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?, ?) |

You can verify the data and mappings in both tables when you run above program.

**2. Hibernate one to one mapping with common join table**

This approach is not new to all of us. Lets start with targeted DB structure in this technique.



In this technique, main annotation to be used is **@JoinTable**. **This annotation is used to define the new table name (mandatory) and foreign keys from both of the tables**. Lets see how it is used:

|  |
| --- |
| EmployeeEntity.java |
| @OneToOne(cascade = CascadeType.ALL)  @JoinTable(name="EMPLOYEE\_ACCCOUNT", joinColumns = @JoinColumn(name="EMPLOYEE\_ID"),  inverseJoinColumns = @JoinColumn(name="ACCOUNT\_ID"))  private AccountEntity account; |

**@JoinTable** annotation is used in EmployeeEntity class. It declares that a new table EMPLOYEE\_ACCOUNT will be created with two columns EMPLOYEE\_ID (primary key of EMPLOYEE table) and ACCOUNT\_ID (primary key of ACCOUNT table).

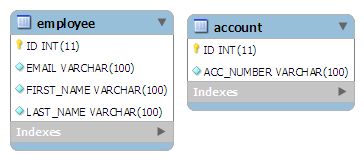
Testing above entities generates following SQL queries in log files:

|  |
| --- |
| Console |
| Hibernate: insert into ACCOUNT (ACC\_NUMBER) values (?)  Hibernate: insert into Employee (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)  Hibernate: insert into EMPLOYEE\_ACCCOUNT (ACCOUNT\_ID, EMPLOYEE\_ID) values (?, ?) |

**3. Hibernate one to one mapping with shared primary key**

In this technique, hibernate will ensure that it will use a common primary key value in both the tables. This way primary key of EmployeeEntity can safely be assumed the primary key of AccountEntity also.

Table structure will be like this:



In this approach, **@PrimaryKeyJoinColumn** is the main annotation to be used. Let see how to use it.

|  |
| --- |
| EmployeeEntity.java |
| @OneToOne(cascade = CascadeType.ALL)  @PrimaryKeyJoinColumn  private AccountEntity account; |

In AccountEntity side, it will remain dependent on owner entity for the mapping.

|  |
| --- |
| AccountEntity.java |
| @OneToOne(mappedBy="account", cascade=CascadeType.ALL)  private EmployeeEntity employee; |

# Hibernate one to many mapping annotation example

## When to use one to many mapping

Use one to mapping to create **1..N relationship** between entities or objects.

For example, we have to write two entities i.e. EmployeeEntity and AccountEntity such that multiple accounts can be associated with a single employee, but one single account can not be shared between two or more employees.

## Hibernate one to many mapping solutions

This problem can be solved in two different ways.

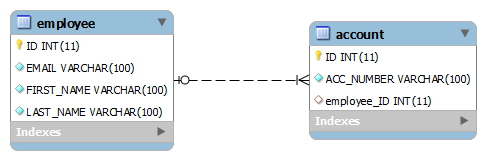
1. One is to have a **foreign key column** in account table i.e. EMPLOYEE\_ID. This column will refer to primary key of Employee table. This way no two accounts can be associated with multiple employees. Obviously, account number needs to be unique for enforcing this restriction.
2. Second approach is to have a common **join table** lets say EMPLOYEE\_ACCOUNT. This table will have two column i.e. EMP\_ID which will be foreign key referring to primary key in EMPLOYEE table and similarly ACCOUNT\_ID which will be foreign key referring to primary key of ACCOUNT table.

## 1. Hibernate one to many mapping with foreign key association

In this approach, **both entity will be responsible for making the relationship** and maintaining it. EmployeeEntity should declare that relationship is one to many, and AccountEntity should declare that relationship from its end is many to one.

#### 1.1. Design one to many mapping relationship

Lets first see the schema design.



#### 1.2. Entity classes

Write entity classes.

|  |
| --- |
| EmployeeEntity.java |
| package hibernate.test.oneToMany.foreignKeyAsso;    import java.io.Serializable;  import java.util.Set;    import javax.persistence.CascadeType;  import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.JoinColumn;  import javax.persistence.OneToMany;  import javax.persistence.Table;  import javax.persistence.UniqueConstraint;    @Entity(name = "ForeignKeyAssoEntity")  @Table(name = "Employee", uniqueConstraints = {  @UniqueConstraint(columnNames = "ID"),  @UniqueConstraint(columnNames = "EMAIL") })  public class EmployeeEntity implements Serializable {        private static final long serialVersionUID = -1798070786993154676L;        @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)      @Column(name = "ID", unique = true, nullable = false)      private Integer employeeId;        @Column(name = "EMAIL", unique = true, nullable = false, length = 100)      private String email;        @Column(name = "FIRST\_NAME", unique = false, nullable = false, length = 100)      private String firstName;        @Column(name = "LAST\_NAME", unique = false, nullable = false, length = 100)      private String lastName;        @OneToMany(cascade=CascadeType.ALL)      @JoinColumn(name="EMPLOYEE\_ID")      private Set<AccountEntity> accounts;        //Getters and setters  } |

Write **AccountEntity.java**.

|  |
| --- |
| AccountEntity.java |
| package hibernate.test.oneToMany.foreignKeyAsso;    import java.io.Serializable;    import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.ManyToOne;  import javax.persistence.Table;  import javax.persistence.UniqueConstraint;    @Entity(name = "ForeignKeyAssoAccountEntity")  @Table(name = "ACCOUNT", uniqueConstraints = {  @UniqueConstraint(columnNames = "ID")})  public class AccountEntity implements Serializable  {      private static final long serialVersionUID = -6790693372846798580L;        @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)      @Column(name = "ID", unique = true, nullable = false)      private Integer accountId;        @Column(name = "ACC\_NUMBER", unique = true, nullable = false, length = 100)      private String accountNumber;        @ManyToOne      private EmployeeEntity employee;        //Getters and setters  } |

#### 1.3. Demo

|  |
| --- |
| package hibernate.test.oneToMany;    import hibernate.test.HibernateUtil;  import hibernate.test.oneToMany.foreignKeyAsso.AccountEntity;  import hibernate.test.oneToMany.foreignKeyAsso.EmployeeEntity;    import java.util.HashSet;  import java.util.Set;    import org.hibernate.Session;    public class TestForeignKeyAssociation  {        public static void main(String[] args)      {          Session session = HibernateUtil.getSessionFactory().openSession();          session.beginTransaction();            AccountEntity account1 = new AccountEntity();          account1.setAccountNumber("Account detail 1");            AccountEntity account2 = new AccountEntity();          account2.setAccountNumber("Account detail 2");            AccountEntity account3 = new AccountEntity();          account3.setAccountNumber("Account detail 3");            //Add new Employee object          EmployeeEntity firstEmployee = new EmployeeEntity();          firstEmployee.setEmail("demo-user-first@mail.com");          firstEmployee.setFirstName("demo-one");          firstEmployee.setLastName("user-one");            EmployeeEntity secondEmployee = new EmployeeEntity();          secondEmployee.setEmail("demo-user-second@mail.com");          secondEmployee.setFirstName("demo-two");          secondEmployee.setLastName("user-two");            Set<AccountEntity> accountsOfFirstEmployee = new HashSet<AccountEntity>();          accountsOfFirstEmployee.add(account1);          accountsOfFirstEmployee.add(account2);            Set<AccountEntity> accountsOfSecondEmployee = new HashSet<AccountEntity>();          accountsOfSecondEmployee.add(account3);            firstEmployee.setAccounts(accountsOfFirstEmployee);          secondEmployee.setAccounts(accountsOfSecondEmployee);          //Save Employee          session.save(firstEmployee);          session.save(secondEmployee);            session.getTransaction().commit();          HibernateUtil.shutdown();      }  } |

Program Output:

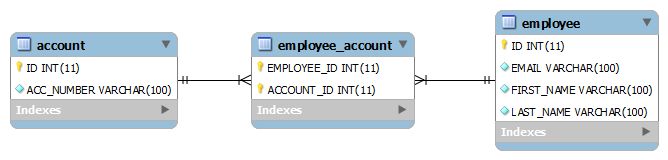
|  |
| --- |
| Console |
| Hibernate: insert into Employee (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)  Hibernate: insert into ACCOUNT (ACC\_NUMBER, employee\_ID) values (?, ?)  Hibernate: insert into ACCOUNT (ACC\_NUMBER, employee\_ID) values (?, ?)  Hibernate: insert into Employee (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)  Hibernate: insert into ACCOUNT (ACC\_NUMBER, employee\_ID) values (?, ?)  Hibernate: update ACCOUNT set EMPLOYEE\_ID=? where ID=?  Hibernate: update ACCOUNT set EMPLOYEE\_ID=? where ID=?  Hibernate: update ACCOUNT set EMPLOYEE\_ID=? where ID=? |

## 2. Hibernate one to many mapping with join table

This approach uses a **join table** to store the associations between account and employee entities. **@JoinTabl**e annotation has been used to make this association.

#### 2.1. Design

Lets see how the database schema will look like:

one To Many association in hibernate using join table

#### 2.2. Entity classes

|  |  |
| --- | --- |
| EmployeeEntity.java | |
| package hibernate.test.oneToMany.joinTable;    import java.io.Serializable;  import java.util.Set;    import javax.persistence.CascadeType;  import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.JoinColumn;  import javax.persistence.JoinTable;  import javax.persistence.OneToMany;  import javax.persistence.Table;  import javax.persistence.UniqueConstraint;    @Entity(name = "JoinTableEmployeeEntity")  @Table(name = "Employee", uniqueConstraints = {  @UniqueConstraint(columnNames = "ID"),  @UniqueConstraint(columnNames = "EMAIL") })  public class EmployeeEntity implements Serializable  {      private static final long serialVersionUID = -1798070786993154676L;        @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)      @Column(name = "ID", unique = true, nullable = false)      private Integer employeeId;        @Column(name = "EMAIL", unique = true, nullable = false, length = 100)      private String email;        @Column(name = "FIRST\_NAME", unique = false, nullable = false, length = 100)      private String firstName;        @Column(name = "LAST\_NAME", unique = false, nullable = false, length = 100)      private String lastName;        @OneToMany(cascade=CascadeType.ALL)      @JoinTable(name="EMPLOYEE\_ACCOUNT", joinColumns={@JoinColumn(name="EMPLOYEE\_ID", referencedColumnName="ID")}      , inverseJoinColumns={@JoinColumn(name="ACCOUNT\_ID", referencedColumnName="ID")})      private Set<AccountEntity> accounts;        //Getters and setters  } | |
| AccountEntity.java |
| package hibernate.test.oneToMany.joinTable;    import java.io.Serializable;    import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.Table;  import javax.persistence.UniqueConstraint;    @Entity(name = "JoinTableAccountEntity")  @Table(name = "ACCOUNT", uniqueConstraints = {  @UniqueConstraint(columnNames = "ID")})  public class AccountEntity implements Serializable  {        private static final long serialVersionUID = -6790693372846798580L;        @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)      @Column(name = "ID", unique = true, nullable = false)      private Integer accountId;        @Column(name = "ACC\_NUMBER", unique = true, nullable = false, length = 100)      private String accountNumber;        //Getters and setters  } |

#### 2.3. Hibernate configuration

**We have available both entities to runtime, we have to add them in hibernate.cfg.xml file. Please note that only one set of entities should be configured in configuration file otherwise unexpected results can occur.**

|  |
| --- |
| hibernate.cfg.xml |
| < ?xml version="1.0" encoding="utf-8"?>  < !DOCTYPE hibernate-configuration PUBLIC  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">  <hibernate-configuration>      <session-factory>          <property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>          <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/hibernatetest</property>          <property name="hibernate.connection.password">XXXXXX</property>          <property name="hibernate.connection.username">root</property>          <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>          <property name="show\_sql">true</property>          <property name="hbm2ddl.auto">create</property>          <mapping clas="hibernate.test.oneToMany.foreignKeyAsso.AccountEntity"></mapping>          <mapping clas="hibernate.test.oneToMany.foreignKeyAsso.EmployeeEntity"></mapping>      </session-factory>  </hibernate-configuration> |

#### 2.4. Demo

Now, its time to test the code. I have written following code to test above entities.

|  |
| --- |
| TestJoinTable.java |
| package hibernate.test.oneToMany;    import hibernate.test.HibernateUtil;  import hibernate.test.oneToMany.joinTable.AccountEntity;  import hibernate.test.oneToMany.joinTable.EmployeeEntity;    import java.util.HashSet;  import java.util.Set;    import org.hibernate.Session;    public class TestJoinTable  {      public static void main(String[] args)      {          Session session = HibernateUtil.getSessionFactory().openSession();          session.beginTransaction();            AccountEntity account1 = new AccountEntity();          account1.setAccountNumber("123-345-65454");            AccountEntity account2 = new AccountEntity();          account2.setAccountNumber("123-345-6542222");            //Add new Employee object          EmployeeEntity emp = new EmployeeEntity();          emp.setEmail("demo-user@mail.com");          emp.setFirstName("demo");          emp.setLastName("user");            Set<AccountEntity> accounts = new HashSet<AccountEntity>();          accounts.add(account1);          accounts.add(account2);            emp.setAccounts(accounts);          //Save Employee          session.save(emp);            session.getTransaction().commit();          HibernateUtil.shutdown();      }  } |

# Hibernate many to many mapping annotation example

## 1. Hibernate many to many mapping design

To demonstrate many to many mapping using hibernate annotations, we will associate two entities i.e. ReaderEntity and SubscriptionEntity.

Their database schema should look like this. Using these tables, any application can save multiple associations between readers and subscriptions.



## 2. Owner entity

Owner entity is the entity which is **responsible make making the association and maintaining it**. In our case, I am making ReaderEntity the owner entity. **@JoinTable** annotation has been used to make this association.

|  |
| --- |
| ReaderEntity.java |
| package hibernate.test.manyToMany.joinTable;    import java.io.Serializable;  import java.util.Set;    import javax.persistence.CascadeType;  import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.JoinColumn;  import javax.persistence.JoinTable;  import javax.persistence.ManyToMany;  import javax.persistence.Table;  import javax.persistence.UniqueConstraint;    @Entity(name = "ReaderEntity")  @Table(name = "READER", uniqueConstraints = {          @UniqueConstraint(columnNames = "ID"),          @UniqueConstraint(columnNames = "EMAIL") })    public class ReaderEntity implements Serializable  {      private static final long serialVersionUID = -1798070786993154676L;        @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)      @Column(name = "ID", unique = true, nullable = false)      private Integer readerId;        @Column(name = "EMAIL", unique = true, nullable = false, length = 100)      private String email;        @Column(name = "FIRST\_NAME", unique = false, nullable = false, length = 100)      private String firstName;        @Column(name = "LAST\_NAME", unique = false, nullable = false, length = 100)      private String lastName;        @ManyToMany(cascade=CascadeType.ALL)      @JoinTable(name="READER\_SUBSCRIPTIONS", joinColumns={@JoinColumn(referencedColumnName="ID")}                                          , inverseJoinColumns={@JoinColumn(referencedColumnName="ID")})      private Set<SubscriptionEntity> subscriptions;        //Getters and setters  } |

## 3. Mapped entity

Our mapped entity is SubscriptionEntity which is mapped to ReaderEntity using “**mappedBy**” attribute.

|  |
| --- |
| SubscriptionEntity.java |
| package hibernate.test.manyToMany.joinTable;    import java.io.Serializable;  import java.util.Set;    import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.ManyToMany;  import javax.persistence.Table;  import javax.persistence.UniqueConstraint;    @Entity(name = "SubscriptionEntity")  @Table(name = "SUBSCRIPTION", uniqueConstraints = {          @UniqueConstraint(columnNames = "ID")})    public class SubscriptionEntity implements Serializable  {      private static final long serialVersionUID = -6790693372846798580L;        @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)      @Column(name = "ID", unique = true, nullable = false)      private Integer subscriptionId;        @Column(name = "SUBS\_NAME", unique = true, nullable = false, length = 100)      private String subscriptionName;        @ManyToMany(mappedBy="subscriptions")      private Set<ReaderEntity> readers;        //Getters and setters  } |

## 4. Configuring entities in hibernate config file

We have make available both entities to runtime. To do so, we have to add them in **hibernate.cfg.xml** file.

|  |
| --- |
| hibernate.cfg.xml |
| <?xml version="1.0" encoding="utf-8"?>  <!DOCTYPE hibernate-configuration PUBLIC  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">  <hibernate-configuration>      <session-factory>          <property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>          <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/hibernatetest</property>          <property name="hibernate.connection.password">XXXXXX</property>          <property name="hibernate.connection.username">root</property>          <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>          <property name="show\_sql">true</property>          <property name="hbm2ddl.auto">create</property>          <mapping class="hibernate.test.manyToMany.joinTable.ReaderEntity"/>          <mapping class="hibernate.test.manyToMany.joinTable.SubscriptionEntity"/>      </session-factory>  </hibernate-configuration> |

## 5. Hibernate many to many annotation mapping example

Now, its time to test the code. I have written following code to test above entities and their many to many relationship.

|  |
| --- |
| TestJoinTable.java |
| package hibernate.test.manyToMany;    import hibernate.test.HibernateUtil;  import hibernate.test.manyToMany.joinTable.\*;    import java.util.HashSet;  import java.util.Set;    import org.hibernate.Session;    public class TestJoinTable  {      public static void main(String[] args)      {          Session session = HibernateUtil.getSessionFactory().openSession();          session.beginTransaction();            //Add subscription          SubscriptionEntity subOne = new SubscriptionEntity();          subOne.setSubscriptionName("Entertainment");            SubscriptionEntity subTwo = new SubscriptionEntity();          subTwo.setSubscriptionName("Horror");            Set<SubscriptionEntity> subs = new HashSet<SubscriptionEntity>();          subs.add(subOne);          subs.add(subTwo);            //Add readers          ReaderEntity readerOne = new ReaderEntity();          readerOne.setEmail("demo-user1@mail.com");          readerOne.setFirstName("demo");          readerOne.setLastName("user");            ReaderEntity readerTwo = new ReaderEntity();          readerTwo.setEmail("demo-user2@mail.com");          readerTwo.setFirstName("demo");          readerTwo.setLastName("user");            Set<ReaderEntity> readers = new HashSet<ReaderEntity>();          readers.add(readerOne);          readers.add(readerTwo);            readerOne.setSubscriptions(subs);          readerTwo.setSubscriptions(subs);            session.save(readerOne);          session.save(readerTwo);            session.getTransaction().commit();          HibernateUtil.shutdown();      }  } |