Hibernate Framework

Hibernate is a Java framework that simplifies the development of Java application to interact with the database. It is an open source, lightweight, ORM (Object Relational Mapping) tool. Hibernate implements the specifications of JPA (Java Persistence API) for data persistence.

ORM Tool

An ORM tool simplifies the data creation, data manipulation and data access. It is a programming technique that maps the object to the data stored in the database.



The ORM tool internally uses the JDBC API to interact with the database.

53.8M

984

Java Try Catch

What is JPA?

Java Persistence API (JPA) is a Java specification that provides certain functionality and standard to ORM tools. The **javax.persistence** package contains the JPA classes and interfaces.

Advantages of Hibernate Framework

Following are the advantages of hibernate framework:

1) Open Source and Lightweight

Hibernate framework is open source under the LGPL license and lightweight.

2) Fast Performance

The performance of hibernate framework is fast because cache is internally used in hibernate framework. There are two types of cache in hibernate framework first level cache and second level cache. First level cache is enabled by default.

3) Database Independent Query

HQL (Hibernate Query Language) is the object-oriented version of SQL. It generates the database independent queries. So you don't need to write database specific queries. Before Hibernate, if database is changed for the project, we need to change the SQL query as well that leads to the maintenance problem.

4) Automatic Table Creation

Hibernate framework provides the facility to create the tables of the database automatically. So there is no need to create tables in the database manually.

5) Simplifies Complex Join

Fetching data from multiple tables is easy in hibernate framework.

# Hibernate Architecture

1. [Hibernate Architecture](https://www.javatpoint.com/hibernate-architecture)
2. [Elements of Hibernate Architecture](https://www.javatpoint.com/hibernate-architecture#elements)
   1. [SessionFactory](https://www.javatpoint.com/hibernate-architecture#e1)
   2. [Session](https://www.javatpoint.com/hibernate-architecture#e2)
   3. [Transaction](https://www.javatpoint.com/hibernate-architecture#e3)
   4. [ConnectionProvider](https://www.javatpoint.com/hibernate-architecture#e4)
   5. [TransactionFactory](https://www.javatpoint.com/hibernate-architecture#e5)

The Hibernate architecture includes many objects such as persistent object, session factory, transaction factory, connection factory, session, transaction etc.

The Hibernate architecture is categorized in four layers.

* Java application layer
* Hibernate framework layer
* Backhand api layer
* Database layer

Let's see the diagram of hibernate architecture:



## Elements of Hibernate Architecture

|  |
| --- |
| For creating the first hibernate application, we must know the elements of Hibernate architecture. They are as follows: |

#### SessionFactory

The SessionFactory is a factory of session and client of ConnectionProvider. It holds second level cache (optional) of data. The org.hibernate.SessionFactory interface provides factory method to get the object of Session.

#### Session

The session object provides an interface between the application and data stored in the database. It is a short-lived object and wraps the JDBC connection. It is factory of Transaction, Query and Criteria. It holds a first-level cache (mandatory) of data. The org.hibernate.Session interface provides methods to insert, update and delete the object. It also provides factory methods for Transaction, Query and Criteria.

#### Transaction

The transaction object specifies the atomic unit of work. It is optional. The org.hibernate.Transaction interface provides methods for transaction management.

# Hibernate Example using Annotation in Eclipse

The hibernate application can be created with annotation. There are many annotations that can be used to create hibernate application such as @Entity, @Id, @Table etc.

Hibernate Annotations are based on the JPA 2 specification and supports all the features.

All the JPA annotations are defined in the **javax.persistence** package. Hibernate EntityManager implements the interfaces and life cycle defined by the JPA specification.

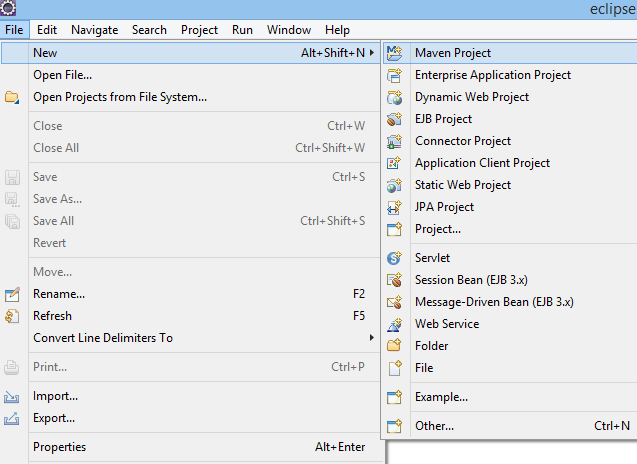
The core advantage of using hibernate annotation is that you don't need to create mapping (hbm) file. Here, hibernate annotations are used to provide the meta data.

### Example to create the hibernate application with Annotation

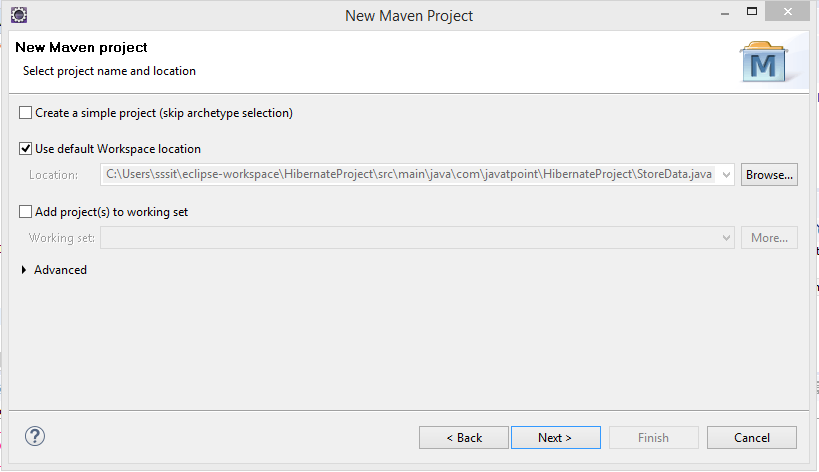
Here, we are going to create a maven based hibernate application using annotation in eclipse IDE. For creating the hibernate application in Eclipse IDE, we need to follow the below steps:

### 1) Create the Maven Project

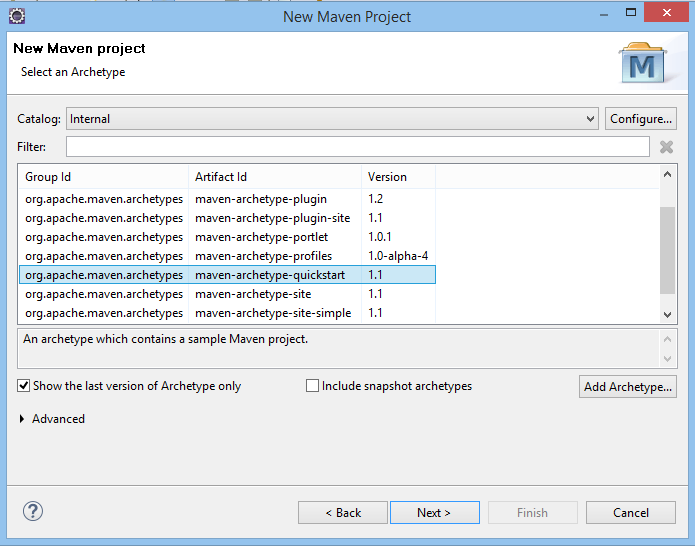
* To create the maven project left click on **File Menu** -**New**-**Maven Project**.



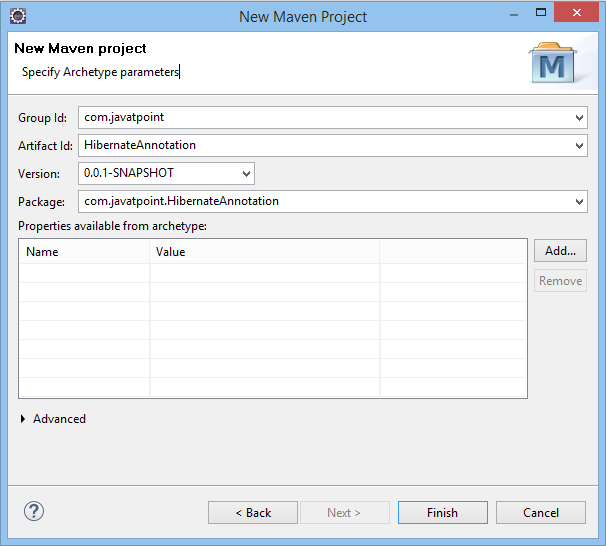
* The new maven project opens in your eclipse. **Click Next**.



* Now, select catalog type: internal and maven archetype - **quickstart** of 1.1 version. Then, **click next**.



* Now, specify the name of Group Id and Artifact Id. The Group Id contains package name (e.g. com.javatpoint) and Artifact Id contains project name (e.g. HibernateAnnotation). Then **click Finish**.



### 2) Add project information and configuration in pom.xml file.

Open pom.xml file and click source. Now, add the below dependencies between <dependencies>....</dependencies> tag. These dependencies are used to add the jar files in Maven project.

1. <dependency>
2. <groupId>org.hibernate</groupId>
3. <artifactId>hibernate-core</artifactId>
4. <version>5.3.1.Final</version>
5. </dependency>
7. <dependency>
8. <groupId>mysql</groupId>
9. <artifactId>mysql-connector-java</artifactId>
10. <version>8.0.30</version>
11. </dependency>

### 3) Create the Persistence class.

Here, we are creating the same persistent class which we have created in the previous topic. But here, we are using annotation.

**@Entity** annotation marks this class as an entity.

**@Table** annotation specifies the table name where data of this entity is to be persisted. If you don't use @Table annotation, hibernate will use the class name as the table name by default.

**@Id** annotation marks the identifier for this entity.

**@Column** annotation specifies the details of the column for this property or field. If @Column annotation is not specified, property name will be used as the column name by default.

To create the Persistence class, right click on **src/main/java - New - Class -** specify the class name with package - **finish**.

**Employee.java**

1. **package** com.javatpoint.mypackage;
3. **import** javax.persistence.Entity;
4. **import** javax.persistence.Id;
5. **import** javax.persistence.Table;
7. @Entity
8. @Table(name= "emp500")
9. **public** **class** Employee {
11. @Id
12. **private** **int** id;
13. **private** String firstName,lastName;
15. **public** **int** getId() {
16. **return** id;
17. }
18. **public** **void** setId(**int** id) {
19. **this**.id = id;
20. }
21. **public** String getFirstName() {
22. **return** firstName;
23. }
24. **public** **void** setFirstName(String firstName) {
25. **this**.firstName = firstName;
26. }
27. **public** String getLastName() {
28. **return** lastName;
29. }
30. **public** **void** setLastName(String lastName) {
31. **this**.lastName = lastName;
32. }
33. }

### 4) Create the Configuration file

To create the configuration file, right click on **src/main/java - new - file -** specify the file name (e.g. hibernate.cfg.xml) - **Finish**.

**hibernate.cfg.xml**

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<!-- Version 8 MySQL hiberante-cfg.xml example for Hibernate 5 -->

<hibernate-configuration>

<session-factory>

property name="connection.driver\_class">com.mysql.jdbc.Driver</property>

<property name="connection.url">jdbc:mysql://localhost:3036/database</property>

<property name="dialect">org.hibernate.dialect.MySQL8Dialect</property>

<property name="connection.username">root</property>

<property name="connection.password">password</property>

<property name="dialect">org.hibernate.dialect.MySQLDialect</property>

<property name="show\_sql">true</property>

<property name="hbm2ddl.auto">update</property>

<mapping class="com.mcnz.jpa.examples.Player"/>

</session-factory>

</hibernate-configuration>

### 5) Create the class that retrieves or stores the persistent object.

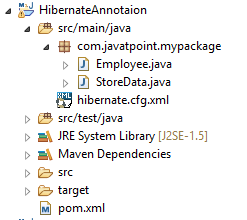
**StoreData.java**

1. **package** com.javatpoint.mypackage;
3. **import** org.hibernate.Session;
4. **import** org.hibernate.SessionFactory;
5. **import** org.hibernate.Transaction;
6. **import** org.hibernate.boot.Metadata;
7. **import** org.hibernate.boot.MetadataSources;
8. **import** org.hibernate.boot.registry.StandardServiceRegistry;
9. **import** org.hibernate.boot.registry.StandardServiceRegistryBuilder;

12. **public** **class** StoreData {
13. **public** **static** **void** main(String[] args) {
15. StandardServiceRegistry ssr = **new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
16. Metadata meta = **new** MetadataSources(ssr).getMetadataBuilder().build();
18. SessionFactory factory = meta.getSessionFactoryBuilder().build();
19. Session session = factory.openSession();
20. Transaction t = session.beginTransaction();
22. Employee e1=**new** Employee();
23. e1.setId(101);
24. e1.setFirstName("Gaurav");
25. e1.setLastName("Chawla");
27. session.save(e1);
28. t.commit();
29. System.out.println("successfully saved");
30. factory.close();
31. session.close();
33. }
34. }

### 6) Run the application

Before running the application, determine that the directory structure is like this.



To run the hibernate application, right click on the **StoreData - Run As - Java Application**.

# Web Application with Hibernate (using XML)

1. [Web Application with Hibernate](https://www.javatpoint.com/web-application-with-hibernate)
2. [Example to create web application using hibernate](https://www.javatpoint.com/web-application-with-hibernate#ex)

Here, we are going to create a web application with hibernate. For creating the web application, we are using JSP for presentation logic, Bean class for representing data and DAO class for database codes.

As we create the simple application in hibernate, we don't need to perform any extra operations in hibernate for creating web application. In such case, we are getting the value from the user using the JSP file.

### Example to create web application using hibernate

In this example, we are going to insert the record of the user in the database. It is simply a registration form.

#### index.jsp

This page gets input from the user and sends it to the register.jsp file using post method.

Play Videox[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. <form action="register.jsp" method="post">
2. Name:<input type="text" name="name"/><br><br/>
3. Password:<input type="password" name="password"/><br><br/>
4. Email ID:<input type="text" name="email"/><br><br/>
5. <input type="submit" value="register"/>"
7. </form>

#### register.jsp

This file gets all request parameters and stores this information into an object of User class. Further, it calls the register method of UserDao class passing the User class object.

1. <%@page **import**="com.javatpoint.mypack.UserDao"%>
2. <jsp:useBean id="obj" **class**="com.javatpoint.mypack.User">
3. </jsp:useBean>
4. <jsp:setProperty property="\*" name="obj"/>
6. <%
7. **int** i=UserDao.register(obj);
8. **if**(i>0)
9. out.print("You are successfully registered");
11. %>

#### User.java

It is the simple bean class representing the Persistent class in hibernate.

1. **package** com.javatpoint.mypack;
3. **public** **class** User {
4. **private** **int** id;
5. **private** String name,password,email;
7. //getters and setters
9. }

Here use entity class

#### UserDao.java

A Dao class, containing method to store the instance of User class.

1. **package** com.javatpoint.mypack;
3. **import** org.hibernate.Session;
4. **import** org.hibernate.SessionFactory;
5. **import** org.hibernate.Transaction;
6. **import** org.hibernate.boot.Metadata;
7. **import** org.hibernate.boot.MetadataSources;
8. **import** org.hibernate.boot.registry.StandardServiceRegistry;
9. **import** org.hibernate.boot.registry.StandardServiceRegistryBuilder;
11. **public** **class** UserDao {
13. **public** **static** **int** register(User u){
14. **int** i=0;
16. StandardServiceRegistry ssr = **new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
17. Metadata meta = **new** MetadataSources(ssr).getMetadataBuilder().build();
19. SessionFactory factory = meta.getSessionFactoryBuilder().build();
20. Session session = factory.openSession();
21. Transaction t = session.beginTransaction();
23. i=(Integer)session.save(u);
25. t.commit();
26. session.close();
28. **return** i;
30. }
31. }

**hibernate.cfg.xml**

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<!-- Version 8 MySQL hiberante-cfg.xml example for Hibernate 5 -->

<hibernate-configuration>

<session-factory>

property name="connection.driver\_class">com.mysql.jdbc.Driver</property>

<property name="connection.url">jdbc:mysql://localhost/database</property>

<property name="dialect">org.hibernate.dialect.MySQL8Dialect</property>

<property name="connection.username">root</property>

<property name="connection.password">password</property>

<property name="dialect">org.hibernate.dialect.MySQLDialect</property>

<property name="show\_sql">true</property>

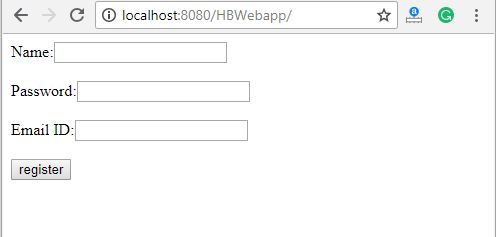
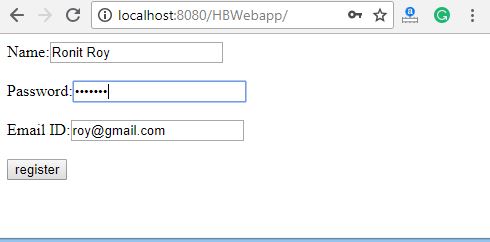
<property name="hbm2ddl.auto">update</property>

<mapping class="com.mcnz.jpa.examples.Player"/>

</session-factory>

</hibernate-configuration>

### Output

# SQL Dialects in Hibernate

The dialect specifies the type of database used in hibernate so that hibernate generate appropriate type of SQL statements. For connecting any hibernate application with the database, it is required to provide the configuration of SQL dialect.

### Syntax of SQL Dialect

1. <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>

### List of SQL Dialects

There are many Dialects classes defined for RDBMS in the **org.hibernate.dialect** package. They are as follows:

|  |  |
| --- | --- |
| **RDBMS** | **Dialect** |
| Oracle (any version) | org.hibernate.dialect.OracleDialect |
| Oracle9i | org.hibernate.dialect.Oracle9iDialect |
| Oracle10g | org.hibernate.dialect.Oracle10gDialect |
| MySQL | org.hibernate.dialect.MySQLDialect |
| MySQL with InnoDB | org.hibernate.dialect.MySQLInnoDBDialect |
| MySQL with MyISAM | org.hibernate.dialect.MySQLMyISAMDialect |
| DB2 | org.hibernate.dialect.DB2Dialect |
| DB2 AS/400 | org.hibernate.dialect.DB2400Dialect |
| DB2 OS390 | org.hibernate.dialect.DB2390Dialect |
| Microsoft SQL Server | org.hibernate.dialect.SQLServerDialect |
| Sybase | org.hibernate.dialect.SybaseDialect |
| Sybase Anywhere | org.hibernate.dialect.SybaseAnywhereDialect |
| PostgreSQL | org.hibernate.dialect.PostgreSQLDialect |
| SAP DB | org.hibernate.dialect.SAPDBDialect |
| Informix | org.hibernate.dialect.InformixDialect |
| HypersonicSQL | org.hibernate.dialect.HSQLDialect |
| Ingres | org.hibernate.dialect.IngresDialect |
| Progress | org.hibernate.dialect.ProgressDialect |
| Mckoi SQL | org.hibernate.dialect.MckoiDialect |
| Interbase | org.hibernate.dialect.InterbaseDialect |
| Pointbase | org.hibernate.dialect.PointbaseDialect |
| FrontBase | org.hibernate.dialect.FrontbaseDialect |
| Firebird | org.hibernate.dialect.FirebirdDialect |

# Hibernate Logging by Log4j using properties file

1. By log4j.properties file

Here, we are going to enable logging using log4j through properties file.

## Steps to perform Hibernate Logging by Log4j using properties file

There are two ways to perform logging using log4j using properties file:

1. Load the log4j jar files with hibernate
2. Create the log4j.properties file inside the src folder (parallel with hibernate.cfg.xml file)

### Example of Hibernate Logging by Log4j using properties file

You can enable logging in hibernate by following only two steps in any hibernate example. This is the first example of hibernate application with logging support using log4j.

Play Videox[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

### Load the required jar files

You need to load the slf4j.jar and log4j.jar files with hibernate jar files.

[download all the required jar files](https://www.javatpoint.com/src/hb/hibernatejar.zip)

### Create log4j.properties file

Now you need to create log4j.properties file. In this example, all the log details will be written in the C:\\javatpointhibernate.log file.

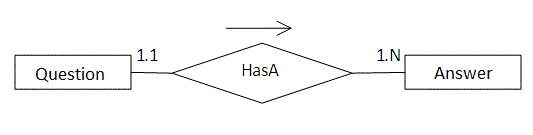
**log4j.properties**

1. # Direct log messages to a log file
2. log4j.appender.file=org.apache.log4j.RollingFileAppender
3. log4j.appender.file.File=C:\\javatpointhibernate.log
4. log4j.appender.file.MaxFileSize=1MB
5. log4j.appender.file.MaxBackupIndex=1
6. log4j.appender.file.layout=org.apache.log4j.PatternLayout
7. log4j.appender.file.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n
9. # Direct log messages to stdout
10. log4j.appender.stdout=org.apache.log4j.ConsoleAppender
11. log4j.appender.stdout.Target=System.out
12. log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
13. log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n
15. # Root logger option
16. log4j.rootLogger=INFO, file, stdout
18. # Log everything. Good **for** troubleshooting
19. log4j.logger.org.hibernate=INFO
21. # Log all JDBC parameters
22. log4j.logger.org.hibernate.type=ALL

# Hibernate One to Many Example using Annotation

In this section, we will perform one-to-many association to map the list object of persistent class using annotation.

Here, we are using the scenario of Forum where one question has multiple answers.



In such case, there can be many answers for a question and each answer may have its own information that is why we have used list in the persistent class (containing the reference of Answer class) to represent a collection of answers.

## Example of One to Many mapping using annotation

### 1) Create the Persistent class

This persistent class defines properties of the class including List.

Play Videox[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

**Question.java**

1. **package** com.javatpoint;
2. **import** javax.persistence.\*;
3. **import** java.util.List;
5. @Entity
6. @Table(name="q5991")
7. **public** **class** Question {
9. @Id
10. @GeneratedValue(strategy=GenerationType.TABLE)
11. **private** **int** id;
12. **private** String qname;
14. @OneToMany(cascade = CascadeType.ALL)
15. @JoinColumn(name="qid")
16. @OrderColumn(name="type")
17. **private** List<Answer> answers;
18. **public** **int** getId() {
19. **return** id;
20. }
21. **public** **void** setId(**int** id) {
22. **this**.id = id;
23. }
24. **public** String getQname() {
25. **return** qname;
26. }
27. **public** **void** setQname(String qname) {
28. **this**.qname = qname;
29. }
30. **public** List<Answer> getAnswers() {
31. **return** answers;
32. }
33. **public** **void** setAnswers(List<Answer> answers) {
34. **this**.answers = answers;
35. }
36. }

**Answer.java**

1. **package** com.javatpoint;
3. **import** javax.persistence.\*;
5. @Entity
6. @Table(name="ans5991")
7. **public** **class** Answer {
8. @Id
9. @GeneratedValue(strategy=GenerationType.TABLE)
11. **private** **int** id;
12. **private** String answername;
13. **private** String postedBy;
14. **public** **int** getId() {
15. **return** id;
16. }
17. **public** **void** setId(**int** id) {
18. **this**.id = id;
19. }
20. **public** String getAnswername() {
21. **return** answername;
22. }
23. **public** **void** setAnswername(String answername) {
24. **this**.answername = answername;
25. }
26. **public** String getPostedBy() {
27. **return** postedBy;
28. }
29. **public** **void** setPostedBy(String postedBy) {
30. **this**.postedBy = postedBy;
31. }
32. }

### 2) Add project information and configuration in pom.xml file.

Open pom.xml file and click source. Now, add the below dependencies between <dependencies> .... </dependencies> tag.

1. <dependency>
2. <groupId>org.hibernate</groupId>
3. <artifactId>hibernate-core</artifactId>
4. <version>5.3.1.Final</version>
5. </dependency>
7. <dependency>
8. <groupId>com.oracle</groupId>
9. <artifactId>ojdbc14</artifactId>
10. <version>10.2.0.4.0</version>
11. </dependency>

### 3) Create the configuration file

This file contains information about the database and mapping file.

**hibernate.cfg.xml**

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<!-- Version 8 MySQL hiberante-cfg.xml example for Hibernate 5 -->

<hibernate-configuration>

<session-factory>

property name="connection.driver\_class">com.mysql.jdbc.Driver</property>

<property name="connection.url">jdbc:mysql://localhost/database</property>

<property name="dialect">org.hibernate.dialect.MySQL8Dialect</property>

<property name="connection.username">root</property>

<property name="connection.password">password</property>

<property name="dialect">org.hibernate.dialect.MySQLDialect</property>

<property name="show\_sql">true</property>

<property name="hbm2ddl.auto">update</property>

<mapping class="com.mcnz.jpa.examples.Player"/>

</session-factory>

</hibernate-configuration>

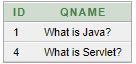
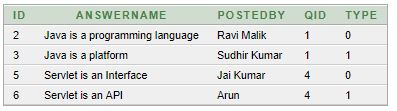
### 4) Create the class to store the data

In this class we are storing the data of the question class.

1. **package** com.javatpoint;
3. **import** java.util.ArrayList;
5. **import** org.hibernate.Session;
6. **import** org.hibernate.SessionFactory;
7. **import** org.hibernate.Transaction;
8. **import** org.hibernate.boot.Metadata;
9. **import** org.hibernate.boot.MetadataSources;
10. **import** org.hibernate.boot.registry.StandardServiceRegistry;
11. **import** org.hibernate.boot.registry.StandardServiceRegistryBuilder;
12. **public** **class** StoreData {
13. **public** **static** **void** main(String[] args) {
15. StandardServiceRegistry ssr=**new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
16. Metadata meta=**new** MetadataSources(ssr).getMetadataBuilder().build();
18. SessionFactory factory=meta.getSessionFactoryBuilder().build();
19. Session session=factory.openSession();
21. Transaction t=session.beginTransaction();
23. Answer ans1=**new** Answer();
24. ans1.setAnswername("Java is a programming language");
25. ans1.setPostedBy("Ravi Malik");
27. Answer ans2=**new** Answer();
28. ans2.setAnswername("Java is a platform");
29. ans2.setPostedBy("Sudhir Kumar");
31. Answer ans3=**new** Answer();
32. ans3.setAnswername("Servlet is an Interface");
33. ans3.setPostedBy("Jai Kumar");
35. Answer ans4=**new** Answer();
36. ans4.setAnswername("Servlet is an API");
37. ans4.setPostedBy("Arun");
39. ArrayList<Answer> list1=**new** ArrayList<Answer>();
40. list1.add(ans1);
41. list1.add(ans2);
43. ArrayList<Answer> list2=**new** ArrayList<Answer>();
44. list2.add(ans3);
45. list2.add(ans4);
47. Question question1=**new** Question();
48. question1.setQname("What is Java?");
49. question1.setAnswers(list1);
51. Question question2=**new** Question();
52. question2.setQname("What is Servlet?");
53. question2.setAnswers(list2);
55. session.persist(question1);
56. session.persist(question2);
58. t.commit();
59. session.close();
60. System.out.println("success");
61. }
62. }

#### ****Note -**** Using these annotations in a similar way, we can also perform one-to-many association for set, map and bag objects.

#### Output

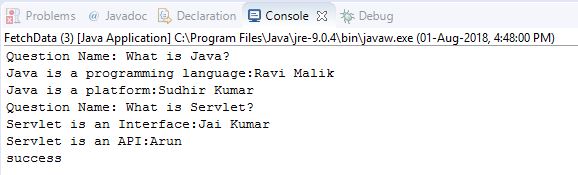
 

## How to fetch the data of List

Here, we have used HQL to fetch all the records of Question class including answers. In such case, it fetches the data from two tables that are functional dependent.

1. **package** com.javatpoint;
2. **import** java.util.\*;
3. **import** javax.persistence.TypedQuery;
4. **import** org.hibernate.\*;
5. **import** org.hibernate.boot.Metadata;
6. **import** org.hibernate.boot.MetadataSources;
7. **import** org.hibernate.boot.registry.StandardServiceRegistry;
8. **import** org.hibernate.boot.registry.StandardServiceRegistryBuilder;
10. **public** **class** FetchData {
11. **public** **static** **void** main(String[] args) {
13. StandardServiceRegistry ssr=**new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
14. Metadata meta=**new** MetadataSources(ssr).getMetadataBuilder().build();
16. SessionFactory factory=meta.getSessionFactoryBuilder().build();
17. Session session=factory.openSession();
19. TypedQuery query=session.createQuery("from Question");
20. List<Question> list=query.getResultList();
22. Iterator<Question> itr=list.iterator();
23. **while**(itr.hasNext()){
24. Question q=itr.next();
25. System.out.println("Question Name: "+q.getQname());
27. //printing answers
28. List<Answer> list2=q.getAnswers();
29. Iterator<Answer> itr2=list2.iterator();
30. **while**(itr2.hasNext())
31. {
32. Answer a=itr2.next();
33. System.out.println(a.getAnswername()+":"+a.getPostedBy());
34. }
35. }
36. session.close();
37. System.out.println("success");
38. }
39. }

#### Output



# Hibernate Many to Many Example using Annotation

In the previous section, we have performed many to many mapping using XML file. Here, we are going to perform this task using annotation.

We can map many to many relation either using list, set, bag, map etc. Here, we are going to use list for many-to-many mapping. In such case, three tables will be created.

## Example of Many to Many Mapping

In this example, we will generate a many to many relation between questions and answers by list.

### 1) Create the Persistent class

**Question.java**

1. **package** com.javatpoint;
3. **import** java.util.List;
4. **import** javax.persistence.\*;
6. @Entity
7. @Table(name="ques1123")
8. **public** **class** Question {
9. @Id
10. @GeneratedValue(strategy=GenerationType.AUTO)
11. **private** **int** id;
12. **private** String qname;
14. @ManyToMany(targetEntity = Answer.**class**, cascade = { CascadeType.ALL })
15. @JoinTable(name = "q\_ans1123",
16. joinColumns = { @JoinColumn(name = "q\_id") },
17. inverseJoinColumns = { @JoinColumn(name = "ans\_id") })
18. **private** List<Answer> answers;
20. **public** **int** getId() {
21. **return** id;
22. }
23. **public** **void** setId(**int** id) {
24. **this**.id = id;
25. }
26. **public** String getQname() {
27. **return** qname;
28. }
29. **public** **void** setQname(String qname) {
30. **this**.qname = qname;
31. }
32. **public** List<Answer> getAnswers() {
33. **return** answers;
34. }
35. **public** **void** setAnswers(List<Answer> answers) {
36. **this**.answers = answers;
37. }
38. }

**Answer.java**

1. **package** com.javatpoint;
3. **import** javax.persistence.\*;
5. @Entity
6. @Table(name="ans1123")
7. **public** **class** Answer {
9. @Id
10. @GeneratedValue(strategy=GenerationType.AUTO)
11. **private** **int** id;
12. **private** String answername;
13. **private** String postedBy;
14. **public** **int** getId() {
15. **return** id;
16. }
17. **public** **void** setId(**int** id) {
18. **this**.id = id;
19. }
20. **public** String getAnswername() {
21. **return** answername;
22. }
23. **public** **void** setAnswername(String answername) {
24. **this**.answername = answername;
25. }
26. **public** String getPostedBy() {
27. **return** postedBy;
28. }
29. **public** **void** setPostedBy(String postedBy) {
30. **this**.postedBy = postedBy;
31. }
33. }

### 2) Add project information and configuration in pom.xml file.

Open pom.xml file and click source. Now, add the below dependencies between <dependencies>....</dependencies> tag.

1. <dependency>
2. <groupId>org.hibernate</groupId>
3. <artifactId>hibernate-core</artifactId>
4. <version>5.3.1.Final</version>
5. </dependency>
6. <dependency>
7. <groupId>com.oracle</groupId>
8. <artifactId>ojdbc14</artifactId>
9. <version>10.2.0.4.0</version>
10. </dependency>
11. **hibernate.cfg.xml**
12. <?xml version="1.0" encoding="UTF-8"?>
13. <!DOCTYPE hibernate-configuration PUBLIC
14. "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
15. "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
16. <!-- Version 8 MySQL hiberante-cfg.xml example for Hibernate 5 -->
17. <hibernate-configuration>
18. <session-factory>
19. property name="connection.driver\_class">com.mysql.jdbc.Driver</property>
20. <property name="connection.url">jdbc:mysql://localhost/database</property>
21. <property name="dialect">org.hibernate.dialect.MySQL8Dialect</property>
22. <property name="connection.username">root</property>
23. <property name="connection.password">password</property>
24. <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
25. <property name="show\_sql">true</property>
26. <property name="hbm2ddl.auto">update</property>
27. <mapping class="com.mcnz.jpa.examples.Player"/>
28. </session-factory>
29. </hibernate-configuration>

### 4) Create the class to store the data

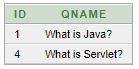
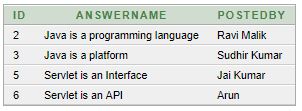
**StoreData.java**

1. **package** com.javatpoint;
2. **import** java.util.ArrayList;
3. **import** org.hibernate.\*;
4. **import** org.hibernate.boot.Metadata;
5. **import** org.hibernate.boot.MetadataSources;
6. **import** org.hibernate.boot.registry.StandardServiceRegistry;
7. **import** org.hibernate.boot.registry.StandardServiceRegistryBuilder;
9. **public** **class** StoreData {
10. **public** **static** **void** main(String[] args) {
12. StandardServiceRegistry ssr=**new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
13. Metadata meta=**new** MetadataSources(ssr).getMetadataBuilder().build();
14. SessionFactory factory=meta.getSessionFactoryBuilder().build();
15. Session session=factory.openSession();
16. Transaction t=session.beginTransaction();
18. Answer an1=**new** Answer();
19. an1.setAnswername("Java is a programming language");
20. an1.setPostedBy("Ravi Malik");
22. Answer an2=**new** Answer();
23. an2.setAnswername("Java is a platform");
24. an2.setPostedBy("Sudhir Kumar");
26. Question q1=**new** Question();
27. q1.setQname("What is Java?");
28. ArrayList<Answer> l1=**new** ArrayList<Answer>();
29. l1.add(an1);
30. l1.add(an2);
31. q1.setAnswers(l1);

34. Answer ans3=**new** Answer();
35. ans3.setAnswername("Servlet is an Interface");
36. ans3.setPostedBy("Jai Kumar");
38. Answer ans4=**new** Answer();
39. ans4.setAnswername("Servlet is an API");
40. ans4.setPostedBy("Arun");
42. Question q2=**new** Question();
43. q2.setQname("What is Servlet?");
44. ArrayList<Answer> l2=**new** ArrayList<Answer>();
45. l2.add(ans3);
46. l2.add(ans4);
47. q2.setAnswers(l2);
49. session.persist(q1);
50. session.persist(q2);
52. t.commit();
53. session.close();
54. System.out.println("success");

57. }
58. }

### Output

# Hibernate One to One Example using Annotation

Here, we are going to perform one to one mapping by one-to-one element using annotation. In such case, no foreign key is created in the primary table.

In this example, one employee can have one address and one address belongs to one employee only. Here, we are using bidirectional association. Let's look at the persistent classes.

### 1) Persistent classes for one to one mapping

There are two persistent classes Employee.java and Address.java. Employee class contains Address class reference and vice versa.

**Employee.java**

67.6M

1.1K

Hello Java Program for Beginners

1. **package** com.javatpoint;
2. **import** javax.persistence.\*;
4. @Entity
5. @Table(name="emp220")
6. **public** **class** Employee {
8. @Id
9. @GeneratedValue(strategy=GenerationType.AUTO)
10. @PrimaryKeyJoinColumn
11. **private** **int** employeeId;
12. **private** String name,email;
13. @OneToOne(targetEntity=Address.**class**,cascade=CascadeType.ALL)
14. **private** Address address;
15. **public** **int** getEmployeeId() {
16. **return** employeeId;
17. }
18. **public** **void** setEmployeeId(**int** employeeId) {
19. **this**.employeeId = employeeId;
20. }
21. **public** String getName() {
22. **return** name;
23. }
24. **public** **void** setName(String name) {
25. **this**.name = name;
26. }
27. **public** String getEmail() {
28. **return** email;
29. }
30. **public** **void** setEmail(String email) {
31. **this**.email = email;
32. }
33. **public** Address getAddress() {
34. **return** address;
35. }
36. **public** **void** setAddress(Address address) {
37. **this**.address = address;
38. }
40. }

**Address.java**

1. **package** com.javatpoint;
2. **import** javax.persistence.\*;
4. @Entity
5. @Table(name="address220")
6. **public** **class** Address {
8. @Id
9. @GeneratedValue(strategy=GenerationType.AUTO)
10. **private** **int** addressId;
11. **private** String addressLine1,city,state,country;
12. **private** **int** pincode;
14. @OneToOne(targetEntity=Employee.**class**)
15. **private** Employee employee;
16. **public** **int** getAddressId() {
17. **return** addressId;
18. }
19. **public** **void** setAddressId(**int** addressId) {
20. **this**.addressId = addressId;
21. }
22. **public** String getAddressLine1() {
23. **return** addressLine1;
24. }
25. **public** **void** setAddressLine1(String addressLine1) {
26. **this**.addressLine1 = addressLine1;
27. }
28. **public** String getCity() {
29. **return** city;
30. }
31. **public** **void** setCity(String city) {
32. **this**.city = city;
33. }
34. **public** String getState() {
35. **return** state;
36. }
37. **public** **void** setState(String state) {
38. **this**.state = state;
39. }
40. **public** String getCountry() {
41. **return** country;
42. }
43. **public** **void** setCountry(String country) {
44. **this**.country = country;
45. }
46. **public** **int** getPincode() {
47. **return** pincode;
48. }
49. **public** **void** setPincode(**int** pincode) {
50. **this**.pincode = pincode;
51. }
52. **public** Employee getEmployee() {
53. **return** employee;
54. }
55. **public** **void** setEmployee(Employee employee) {
56. **this**.employee = employee;
57. }
58. }

### 2) Add project information and configuration in pom.xml file.

Open pom.xml file and click source. Now, add the below dependencies between <dependencies>....</dependencies> tag. These dependencies are used to add the jar files in Maven project.

1. <dependency>
2. <groupId>org.hibernate</groupId>
3. <artifactId>hibernate-core</artifactId>
4. <version>5.3.1.Final</version>
5. </dependency>
6. <dependency>
7. <groupId>com.oracle</groupId>
8. <artifactId>ojdbc14</artifactId>
9. <version>10.2.0.4.0</version>
10. </dependency>
11. **hibernate.cfg.xml**
12. <?xml version="1.0" encoding="UTF-8"?>
13. <!DOCTYPE hibernate-configuration PUBLIC
14. "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
15. "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
16. <!-- Version 8 MySQL hiberante-cfg.xml example for Hibernate 5 -->
17. <hibernate-configuration>
18. <session-factory>
19. property name="connection.driver\_class">com.mysql.jdbc.Driver</property>
20. <property name="connection.url">jdbc:mysql://localhost/database</property>
21. <property name="dialect">org.hibernate.dialect.MySQL8Dialect</property>
22. <property name="connection.username">root</property>
23. <property name="connection.password">password</property>
24. <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
25. <property name="show\_sql">true</property>
26. <property name="hbm2ddl.auto">update</property>
27. <mapping class="com.mcnz.jpa.examples.Player"/>
28. </session-factory>
29. </hibernate-configuration>

### 4) User classes to store and fetch the data

**Store.java**

1. **package** com.javatpoint;
3. **import** org.hibernate.\*;
4. **import** org.hibernate.boot.Metadata;
5. **import** org.hibernate.boot.MetadataSources;
6. **import** org.hibernate.boot.registry.StandardServiceRegistry;
7. **import** org.hibernate.boot.registry.StandardServiceRegistryBuilder;
9. **public** **class** Store {
10. **public** **static** **void** main(String[] args) {
12. StandardServiceRegistry ssr=**new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
13. Metadata meta=**new** MetadataSources(ssr).getMetadataBuilder().build();
15. SessionFactory factory=meta.getSessionFactoryBuilder().build();
16. Session session=factory.openSession();
18. Transaction t=session.beginTransaction();
20. Employee e1=**new** Employee();
21. e1.setName("Ravi Malik");
22. e1.setEmail("ravi@gmail.com");
24. Address address1=**new** Address();
25. address1.setAddressLine1("G-21,Lohia nagar");
26. address1.setCity("Ghaziabad");
27. address1.setState("UP");
28. address1.setCountry("India");
29. address1.setPincode(201301);
31. e1.setAddress(address1);
32. address1.setEmployee(e1);
34. session.persist(e1);
35. t.commit();
37. session.close();
38. System.out.println("success");
39. }
40. }

### Output

Hibernate One to One Example using Annotation 1 Hibernate One to One Example using Annotation 2

**Fetch.java**

1. **package** com.javatpoint;
2. **import** java.util.Iterator;
3. **import** java.util.List;
5. **import** javax.persistence.TypedQuery;
6. **import** org.hibernate.Session;
7. **import** org.hibernate.SessionFactory;
8. **import** org.hibernate.boot.Metadata;
9. **import** org.hibernate.boot.MetadataSources;
10. **import** org.hibernate.boot.registry.StandardServiceRegistry;
11. **import** org.hibernate.boot.registry.StandardServiceRegistryBuilder;
13. **public** **class** Fetch {
14. **public** **static** **void** main(String[] args) {
15. StandardServiceRegistry ssr=**new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
16. Metadata meta=**new** MetadataSources(ssr).getMetadataBuilder().build();
18. SessionFactory factory=meta.getSessionFactoryBuilder().build();
19. Session session=factory.openSession();
21. TypedQuery query=session.createQuery("from Employee");
22. List<Employee> list=query.getResultList();
24. Iterator<Employee> itr=list.iterator();
25. **while**(itr.hasNext()){
26. Employee emp=itr.next();
27. System.out.println(emp.getEmployeeId()+" "+emp.getName()+" "+emp.getEmail());
28. Address address=emp.getAddress();
29. System.out.println(address.getAddressLine1()+" "+address.getCity()+" "+
30. address.getState()+" "+address.getCountry()+" "+address.getPincode());
31. }
33. session.close();
34. System.out.println("success");
35. }
36. }

# Hibernate Many to One Mapping using Annotation

In many to one mapping, various attributes can be referred to one attribute only.

In this example, every employee has one company address only and one address belongs to many employees. Here, we are going to perform many to one mapping using annotation.

Let's look at the persistent classes

## 1) Persistent classes for one to one mapping

There are two persistent classes Employee.java and Address.java. Employee class contains Address class reference and vice versa.

Play Videox[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

### Employee.java

1. **package** com.javatpoint;
2. **import** javax.persistence.\*;
4. @Entity
5. @Table(name="emp107")
6. **public** **class** Employee {
7. @Id
8. @GeneratedValue(strategy=GenerationType.AUTO)
9. **private** **int** employeeId;
10. **private** String name,email;
11. @ManyToOne(cascade=CascadeType.ALL)
12. **private** Address address;
13. **public** **int** getEmployeeId() {
14. **return** employeeId;
15. }
16. **public** **void** setEmployeeId(**int** employeeId) {
17. **this**.employeeId = employeeId;
18. }
19. **public** String getName() {
20. **return** name;
21. }
22. **public** **void** setName(String name) {
23. **this**.name = name;
24. }
25. **public** String getEmail() {
26. **return** email;
27. }
28. **public** **void** setEmail(String email) {
29. **this**.email = email;
30. }
31. **public** Address getAddress() {
32. **return** address;
33. }
34. **public** **void** setAddress(Address address) {
35. **this**.address = address;
36. }
37. }

### Address.java

1. **package** com.javatpoint;
3. **import** javax.persistence.\*;
5. @Entity
6. @Table(name="address107")
7. **public** **class** Address {
8. @Id
9. @GeneratedValue(strategy=GenerationType.AUTO)
10. **private** **int** addressId;
11. **private** String addressLine1,city,state,country;
12. **private** **int** pincode;
13. @OneToOne(cascade=CascadeType.ALL)
14. **private** Employee employee;
15. **public** **int** getAddressId() {
16. **return** addressId;
17. }
18. **public** **void** setAddressId(**int** addressId) {
19. **this**.addressId = addressId;
20. }
21. **public** String getAddressLine1() {
22. **return** addressLine1;
23. }
24. **public** **void** setAddressLine1(String addressLine1) {
25. **this**.addressLine1 = addressLine1;
26. }
27. **public** String getCity() {
28. **return** city;
29. }
30. **public** **void** setCity(String city) {
31. **this**.city = city;
32. }
33. **public** String getState() {
34. **return** state;
35. }
36. **public** **void** setState(String state) {
37. **this**.state = state;
38. }
39. **public** String getCountry() {
40. **return** country;
41. }
42. **public** **void** setCountry(String country) {
43. **this**.country = country;
44. }
45. **public** **int** getPincode() {
46. **return** pincode;
47. }
48. **public** **void** setPincode(**int** pincode) {
49. **this**.pincode = pincode;
50. }
51. **public** Employee getEmployee() {
52. **return** employee;
53. }
54. **public** **void** setEmployee(Employee employee) {
55. **this**.employee = employee;
56. }
57. }

## 2) Add project information and configuration in pom.xml file.

Open pom.xml file and click source. Now, add the below dependencies between <dependencies>....</dependencies> tag. These dependencies are used to add the jar files in Maven project.

1. <dependency>
2. <groupId>org.hibernate</groupId>
3. <artifactId>hibernate-core</artifactId>
4. <version>5.3.1.Final</version>
5. </dependency>
6. <dependency>
7. <groupId>com.oracle</groupId>
8. <artifactId>ojdbc14</artifactId>
9. <version>10.2.0.4.0</version>
10. </dependency>
11. **hibernate.cfg.xml**
12. <?xml version="1.0" encoding="UTF-8"?>
13. <!DOCTYPE hibernate-configuration PUBLIC
14. "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
15. "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
16. <!-- Version 8 MySQL hiberante-cfg.xml example for Hibernate 5 -->
17. <hibernate-configuration>
18. <session-factory>
19. property name="connection.driver\_class">com.mysql.jdbc.Driver</property>
20. <property name="connection.url">jdbc:mysql://localhost/database</property>
21. <property name="dialect">org.hibernate.dialect.MySQL8Dialect</property>
22. <property name="connection.username">root</property>
23. <property name="connection.password">password</property>
24. <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
25. <property name="show\_sql">true</property>
26. <property name="hbm2ddl.auto">update</property>
27. <mapping class="com.mcnz.jpa.examples.Player"/>
28. </session-factory>
29. </hibernate-configuration>

## 4) User classes to store and fetch the data

### Store.java

1. **package** com.javatpoint;
3. **import** org.hibernate.\*;
4. **import** org.hibernate.boot.Metadata;
5. **import** org.hibernate.boot.MetadataSources;
6. **import** org.hibernate.boot.registry.StandardServiceRegistry;
7. **import** org.hibernate.boot.registry.StandardServiceRegistryBuilder;
9. **public** **class** Store {
10. **public** **static** **void** main(String[] args) {
12. StandardServiceRegistry ssr=**new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
13. Metadata meta=**new** MetadataSources(ssr).getMetadataBuilder().build();
15. SessionFactory factory=meta.getSessionFactoryBuilder().build();
16. Session session=factory.openSession();
18. Transaction t=session.beginTransaction();
20. Employee e1=**new** Employee();
21. e1.setName("Ravi Malik");
22. e1.setEmail("ravi@gmail.com");
24. Employee e2=**new** Employee();
25. e2.setName("Anuj Verma");
26. e2.setEmail("anuj@gmail.com");
28. Address address1=**new** Address();
29. address1.setAddressLine1("G-13,Sector 3");
30. address1.setCity("Noida");
31. address1.setState("UP");
32. address1.setCountry("India");
33. address1.setPincode(201301);
35. e1.setAddress(address1);
36. e2.setAddress(address1);
38. session.persist(e1);
39. session.persist(e2);
40. t.commit();
42. session.close();
43. System.out.println("success");
44. }
45. }

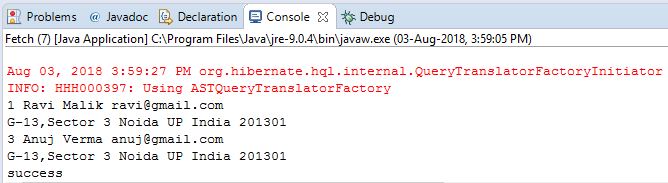
### OUTPUT

 Hibernate Many to One Example using Annotation 2

### Fetch.java

1. **package** com.javatpoint;
2. **import** java.util.Iterator;
3. **import** java.util.List;
5. **import** javax.persistence.TypedQuery;
6. **import** org.hibernate.Session;
7. **import** org.hibernate.SessionFactory;
8. **import** org.hibernate.boot.Metadata;
9. **import** org.hibernate.boot.MetadataSources;
10. **import** org.hibernate.boot.registry.StandardServiceRegistry;
11. **import** org.hibernate.boot.registry.StandardServiceRegistryBuilder;
13. **public** **class** Fetch {
14. **public** **static** **void** main(String[] args) {
15. StandardServiceRegistry ssr=**new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
16. Metadata meta=**new** MetadataSources(ssr).getMetadataBuilder().build();
18. SessionFactory factory=meta.getSessionFactoryBuilder().build();
19. Session session=factory.openSession();
21. TypedQuery query=session.createQuery("from Employee e");
22. List<Employee> list=query.getResultList();
24. Iterator<Employee> itr=list.iterator();
25. **while**(itr.hasNext()){
26. Employee emp=itr.next();
27. System.out.println(emp.getEmployeeId()+" "+emp.getName()+" "+emp.getEmail());
28. Address address=emp.getAddress();
29. System.out.println(address.getAddressLine1()+" "+address.getCity()+" "+
30. address.getState()+" "+address.getCountry()+" "+address.getPincode());
31. }
33. session.close();
34. System.out.println("success");
35. }
36. }

### Output

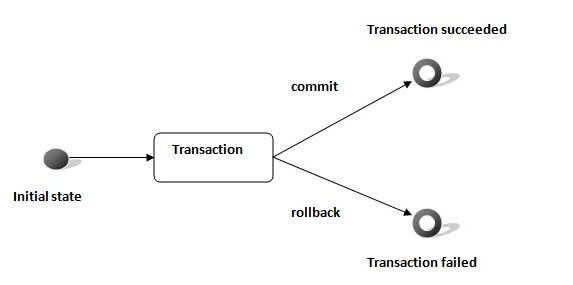


### Output

# Hibernate Transaction Management Example

1. [Understanding Transaction](https://www.javatpoint.com/hibernate-transaction-management-example)
2. [ACID Property](https://www.javatpoint.com/hibernate-transaction-management-example)
3. [Hibernate Transaction Management](https://www.javatpoint.com/hibernate-transaction-management-example)

A **transaction** simply represents a unit of work. In such case, if one step fails, the whole transaction fails (which is termed as atomicity). A transaction can be described by ACID properties (Atomicity, Consistency, Isolation and Durability).



### Transaction Interface in Hibernate

In hibernate framework, we have **Transaction** interface that defines the unit of work. It maintains abstraction from the transaction implementation (JTA,JDBC).

A transaction is associated with Session and instantiated by calling **session.beginTransaction()**.

The methods of Transaction interface are as follows:

1. **void begin()** starts a new transaction.
2. **void commit()** ends the unit of work unless we are in FlushMode.NEVER.
3. **void rollback()** forces this transaction to rollback.
4. **void setTimeout(int seconds)** it sets a transaction timeout for any transaction started by a subsequent call to begin on this instance.
5. **boolean isAlive()** checks if the transaction is still alive.
6. **void registerSynchronization(Synchronization s)** registers a user synchronization callback for this transaction.
7. **boolean wasCommited()** checks if the transaction is commited successfully.
8. **boolean wasRolledBack()** checks if the transaction is rolledback successfully.

### Example of Transaction Management in Hibernate

In hibernate, it is better to rollback the transaction if any exception occurs, so that resources can be free. Let's see the example of transaction management in hibernate.

1. Session session = **null**;
2. Transaction tx = **null**;
4. **try** {
5. session = sessionFactory.openSession();
6. tx = session.beginTransaction();
7. //some action
9. tx.commit();
11. }**catch** (Exception ex) {
12. ex.printStackTrace();
13. tx.rollback();
14. }
15. **finally** {session.close();}

# Hibernate Query Language (HQL)

1. [Hibernate Query Language](https://www.javatpoint.com/hql)
2. [Advantage of HQL](https://www.javatpoint.com/hql)
3. [Query Interface](https://www.javatpoint.com/hql)

Hibernate Query Language (HQL) is same as SQL (Structured Query Language) but it doesn't depends on the table of the database. Instead of table name, we use class name in HQL. So it is database independent query language.

### Advantage of HQL

There are many advantages of HQL. They are as follows:

* database independent
* supports polymorphic queries
* easy to learn for Java Programmer

### Query Interface

It is an object oriented representation of Hibernate Query. The object of Query can be obtained by calling the createQuery() method Session interface.

The query interface provides many methods. There is given commonly used methods:

Play Videox[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. **public int executeUpdate()** is used to execute the update or delete query.
2. **public List list()** returns the result of the ralation as a list.
3. **public Query setFirstResult(int rowno)** specifies the row number from where record will be retrieved.
4. **public Query setMaxResult(int rowno)** specifies the no. of records to be retrieved from the relation (table).
5. **public Query setParameter(int position, Object value)** it sets the value to the JDBC style query parameter.
6. **public Query setParameter(String name, Object value)** it sets the value to a named query parameter.

### Example of HQL to get all the records

1. Query query=session.createQuery("from Emp");//here persistent class name is Emp
2. List list=query.list();

### Example of HQL to get records with pagination

1. Query query=session.createQuery("from Emp");
2. query.setFirstResult(5);
3. query.setMaxResult(10);
4. List list=query.list();//will return the records from 5 to 10th number

### Example of HQL update query

1. Transaction tx=session.beginTransaction();
2. Query q=session.createQuery("update User set name=:n where id=:i");
3. q.setParameter("n","Udit Kumar");
4. q.setParameter("i",111);
6. **int** status=q.executeUpdate();
7. System.out.println(status);
8. tx.commit();

### Example of HQL delete query

1. Query query=session.createQuery("delete from Emp where id=100");
2. //specifying class name (Emp) not tablename
3. query.executeUpdate();

### HQL with Aggregate functions

You may call avg(), min(), max() etc. aggregate functions by HQL. Let's see some common examples:

### Example to get total salary of all the employees

1. Query q=session.createQuery("select sum(salary) from Emp");
2. List<Integer> list=q.list();
3. System.out.println(list.get(0));

### Example to get maximum salary of employee

1. Query q=session.createQuery("select max(salary) from Emp");

### Example to get minimum salary of employee

1. Query q=session.createQuery("select min(salary) from Emp");

### Example to count total number of employee ID

1. Query q=session.createQuery("select count(id) from Emp");

### Example to get average salary of each employees

1. Query q=session.createQuery("select avg(salary) from Emp");

# HCQL (Hibernate Criteria Query Language)

1. [Hibernate Criteria Query Language](https://www.javatpoint.com/hcql)
2. [Criteria Interface](https://www.javatpoint.com/hcql#criteria)
3. [Restrictions class](https://www.javatpoint.com/hcql#restrictions)
4. [Examples of HCQL](https://www.javatpoint.com/hcql#ex)

The Hibernate Criteria Query Language (HCQL) is used to fetch the records based on the specific criteria. The Criteria interface provides methods to apply criteria such as retreiving all the records of table whose salary is greater than 50000 etc.

### Advantage of HCQL

The HCQL provides methods to add criteria, so it is **easy** for the java programmer to add criteria. The java programmer is able to add many criteria on a query.

### Criteria Interface

The Criteria interface provides many methods to specify criteria. The object of Criteria can be obtained by calling the **createCriteria()** method of Session interface.

#### Syntax of createCriteria() method of Session interface

1. **public** Criteria createCriteria(Class c)

The commonly used methods of Criteria interface are as follows:

1. **public Criteria add(Criterion c)** is used to add restrictions.
2. **public Criteria addOrder(Order o)** specifies ordering.
3. **public Criteria setFirstResult(int firstResult)** specifies the first number of record to be retreived.
4. **public Criteria setMaxResult(int totalResult)** specifies the total number of records to be retreived.
5. **public List list()** returns list containing object.
6. **public Criteria setProjection(Projection projection)** specifies the projection.

### Restrictions class

Restrictions class provides methods that can be used as Criterion. The commonly used methods of Restrictions class are as follows:

1. **public static SimpleExpression lt(String propertyName,Object value)** sets the **less than** constraint to the given property.
2. **public static SimpleExpression le(String propertyName,Object value)** sets the **less than or equal** constraint to the given property.
3. **public static SimpleExpression gt(String propertyName,Object value)** sets the **greater than** constraint to the given property.
4. **public static SimpleExpression ge(String propertyName,Object value)** sets the **greater than or equal** than constraint to the given property.
5. **public static SimpleExpression ne(String propertyName,Object value)** sets the **not equal** constraint to the given property.
6. **public static SimpleExpression eq(String propertyName,Object value)** sets the **equal** constraint to the given property.
7. **public static Criterion between(String propertyName, Object low, Object high)** sets the **between** constraint.
8. **public static SimpleExpression like(String propertyName, Object value)** sets the **like** constraint to the given property.

### Order class

The Order class represents an order. The commonly used methods of Restrictions class are as follows:

1. **public static Order asc(String propertyName)** applies the ascending order on the basis of given property.
2. **public static Order desc(String propertyName)** applies the descending order on the basis of given property.

### Examples of Hibernate Criteria Query Language

There are given a lot of examples of HCQL.

### Example of HCQL to get all the records

1. Crietria c=session.createCriteria(Emp.**class**);//passing Class class argument
2. List list=c.list();

### Example of HCQL to get the 10th to 20th record

1. Crietria c=session.createCriteria(Emp.**class**);
2. c.setFirstResult(10);
3. c.setMaxResult(20);
4. List list=c.list();

### Example of HCQL to get the records whose salary is greater than 10000

1. Crietria c=session.createCriteria(Emp.**class**);
2. c.add(Restrictions.gt("salary",10000));//salary is the propertyname
3. List list=c.list();

### Example of HCQL to get the records in ascending order on the basis of salary

1. Crietria c=session.createCriteria(Emp.**class**);
2. c.addOrder(Order.asc("salary"));
3. List list=c.list();

### HCQL with Projection

We can fetch data of a particular column by projection such as name etc. Let's see the simple example of projection that prints data of NAME column of the table only.

1. Criteria c=session.createCriteria(Emp.**class**);
2. c.setProjection(Projections.property("name"));
3. List list=c.list();

# JPA vs. Hibernate

## What is JPA?

A JPA (Java Persistence API) is a specification of Java which is used to access, manage, and persist data between Java object and relational database. It is considered as a standard approach for Object Relational Mapping.

JPA can be seen as a bridge between object-oriented domain models and relational database systems. Being a specification, JPA doesn't perform any operation by itself. Thus, it requires implementation. So, ORM tools like Hibernate, TopLink, and iBatis implements JPA specifications for data persistence.

## What is Hibernate?

A Hibernate is a Java framework which is used to store the Java objects in the relational database system. It is an open-source, lightweight, ORM (Object Relational Mapping) tool.

Hibernate is an implementation of JPA. So, it follows the common standards provided by the JPA.

Play Videox[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

## Need of JPA

As we have seen so far, JPA is a specification. It provides common prototype and functionality to ORM tools. By implementing the same specification, all ORM tools (like Hibernate, TopLink, iBatis) follows the common standards. In the future, if we want to switch our application from one ORM tool to another, we can do it easily.

## JPA vs. Hibernate

|  |  |
| --- | --- |
| **JPA** | **Hibernate** |
| Java Persistence API (JPA) defines the management of relational data in the Java applications. | Hibernate is an Object-Relational Mapping (ORM) tool which is used to save the state of Java object into the database. |
| It is just a specification. Various ORM tools implement it for data persistence. | It is one of the most frequently used JPA implementation. |
| It is defined in **javax.persistence** package. | It is defined in **org.hibernate** package. |
| The **EntityManagerFactory** interface is used to interact with the entity manager factory for the persistence unit. Thus, it provides an entity manager. | It uses **SessionFactory** interface to create Session instances. |
| It uses **EntityManager** interface to create, read, and delete operations for instances of mapped entity classes. This interface interacts with the persistence context. | It uses **Session** interface to create, read, and delete operations for instances of mapped entity classes. It behaves as a runtime interface between a Java application and Hibernate. |
| It uses **Java Persistence Query Language** (JPQL) as an object-oriented query language to perform database operations. |  |