HIBERNATE QUICK GUIDE

Hibernate is an Object-Relational Mapping (ORM) solution for JAVA and it raised as an open source persistent framework created by Gavin King in 2001. It is a powerful, high performance Object-Relational Persistence and Query service for any Java Application.

Hibernate maps Java classes to database tables and from Java data types to SQL data types and relieve the developer from 95% of common data persistence related programming tasks.

Hibernate sits between traditional Java objects and database server to handle all the work in persisting those objects based on the appropriate O/R mechanisms and patterns.

Hibernate Advantages:

- Hibernate takes care of mapping Java classes to database tables using XML files and without writing any line of code.

- Provides simple APIs for storing and retrieving Java objects directly to and from the database.

- If there is change in Database or in any table then the only need to change XML file properties.

- Abstract away the unfamiliar SQL types and provide us to work around familiar Java Objects.

- Hibernate does not require an application server to operate.

- Manipulates Complex associations of objects of your database.

- Minimize database access with smart fetching strategies.

- Provides Simple querying of data.

Supported Databases:

Hibernate supports almost all the major RDBMS. Following is list of few of the database engines supported by Hibernate.

- HSQL Database Engine

- DB2/NT

- MySQL

- PostgreSQL

- FrontBase

- Oracle

- Microsoft SQL Server Database

- Sybase SQL Server

- Informix Dynamic Server

Hibernate Architecture:

The Hibernate architecture is layered to keep you isolated from having to know the underlying APIs. Hibernate makes use of the database and configuration data to provide persistence services (and persistent objects) to the application.

Following is a very high level view of the Hibernate Application Architecture.
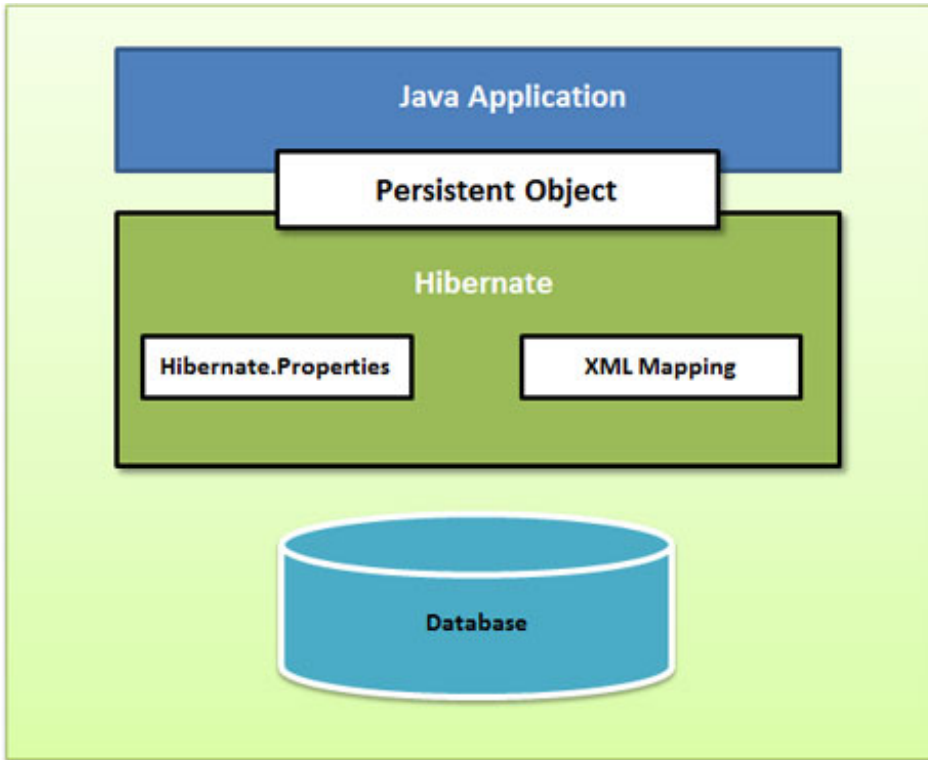
Following is a detailed view of the Hibernate Application Architecture with few important core classes.
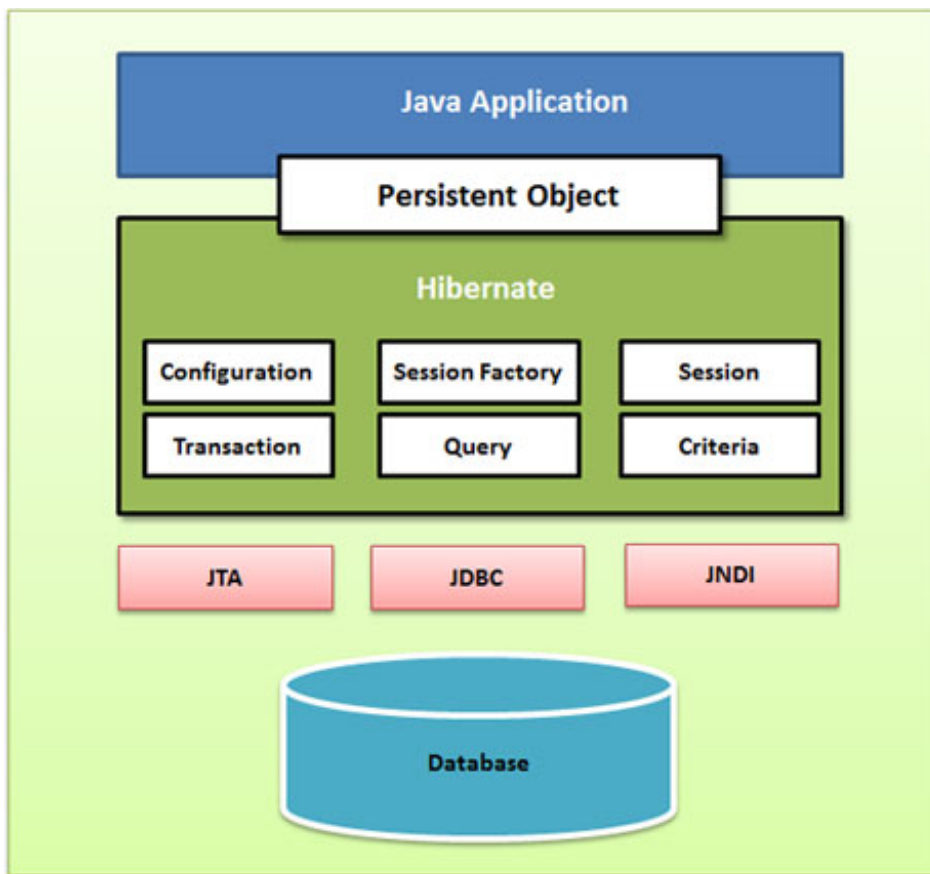
Hibernate uses various existing Java APIs, like JDBC, Java Transaction API(JTA), and Java Naming and Directory Interface (JNDI). JDBC provides a rudimentary level of abstraction of functionality common to relational databases, allowing almost any database with a JDBC driver to be supported by Hibernate. JNDI and JTA allow Hibernate to be integrated with J2EE application servers.

Following section gives brief description of each of the class objects involved in Hibernate Application Architecture.

Configuration Object:

The Configuration object is the first Hibernate object you create in any Hibernate application and usually created only once during application initialization. It represents a configuration or properties file required by the Hibernate. The Configuration object provides two keys components:

- **Database Connection:** This is handled through one or more configuration files supported by Hibernate. These files are **hibernate.properties** and **hibernate.cfg.xml**.

- **Class Mapping Setup**
  This component creates the connection between the Java classes and database tables..

SessionFactory Object:

Configuration object is used to create a SessionFactory object which inturn configures Hibernate for the application using the supplied configuration file and allows for a Session object to be instantiated. The SessionFactory is a thread safe object and used by all the threads of an application.

The SessionFactory is heavyweight object so usually it is created during application start up and kept for later use. You would need one SessionFactory object per database using a separate configuration file. So if you are using multiple databases then you would have to create multiple SessionFactory objects.

Session Object:

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

The session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed them as needed.

Transaction Object:

A Transaction represents a unit of work with the database and most of the RDBMS supports transaction functionality. Transactions in Hibernate are handled by an underlying transaction manager and transaction (from JDBC or JTA).

This is an optional object and Hibernate applications may choose not to use this interface, instead managing transactions in their own application code.

Query Object:

Query objects use SQL or Hibernate Query Language (HQL) string to retrieve data from the database and create objects. A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

Criteria Object:

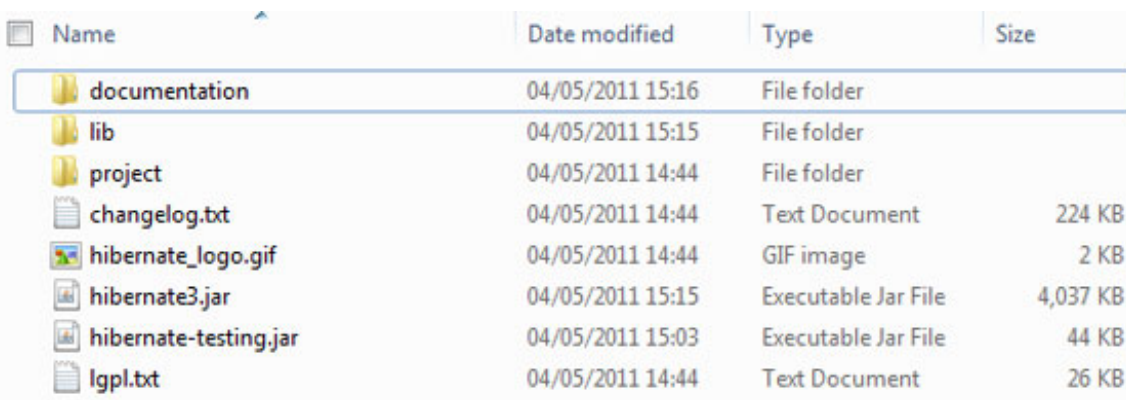Criteria object are used to create and execute object oriented criteria queries to retrieve objects.

Hibernate Environment Setup

This chapter will explain how to install Hibernate and other associated packages to prepare a develop environment for the Hibernate applications. We will work with MySQL database to experiment with Hibernate examples, so make sure you already have setup for MySQL database. For a more detail on MySQL you can check our MySQL Tutorial.

Downloading Hibernate:

It is assumed that you already have latest version of Java is installed on your machine. Following are the simple steps to download and install Hibernate on your machine.

- Make a choice whether you want to install Hibernate on Windows, or Unix and then proceed to the next step to download .zip file for windows and .tz file for Unix.

- Download the latest version of Hibernate from http://www.hibernate.org/downloads.

- At the time of writing this tutorial I downloaded **hibernate-distribution-3.6.4.Final** and when you unzip the downloaded file it will give you directory structure as follows.

Installing Hibernate:

Once you downloaded and unzipped the latest version of the Hibernate Installation file, you need to perform following two simple steps. Make sure you are setting your CLASSPATH variable properly otherwise you will face problem while compiling your application.

- Now copy all the library files from **/lib** into your CLASSPATH, and change your classpath variable to include all the JARs:

- Finally copy **hibernate3.jar** file into your CLASSPATH. This file lies in the root directory of the installation and is the primary JAR that Hibernate needs to do its work.

Hibernate Prerequisites:

Following is the list of the packages/libraries required by Hibernate and you should install them before starting with Hibernate. To install these packages you would have to copy library files from **/lib** into your CLASSPATH, and change your CLASSPATH variable accordingly.

S.N.	Packages/Libraries
1	**dom4j** - XML parsing www.dom4j.org/
2	**Xalan** - XSLT Processor http://xml.apache.org/xalan-j/
3	**Xerces** - The Xerces Java Parser http://xml.apache.org/xerces-j/
4	**cglib** - Appropriate changes to Java classes at runtime http://cglib.sourceforge.net/
5	**log4j** - Logging Faremwork http://logging.apache.org/log4j
6	**Commons** - Logging, Email etc. http://jakarta.apache.org/commons
7	**SLF4J** - Logging Facade for Java http://www.slf4j.org

Hibernate Configuration

Hibernate requires to know in advance where to find the mapping information that defines how your Java classes relate to the database tables. Hibernate also requires a set of configuration settings related to database and other related parameters. All such information is usually supplied as a standard Java properties file called **hibernate.properties**, or as an XML file named **hibernate.cfg.xml**.

I will consider XML formatted file **hibernate.cfg.xml** to specify required Hibernate properties in my examples. Most of the properties take their default values and it is not required to specify them in the property file unless it is really required. This file is kept in the root directory of your application's classpath.

Hibernate Properties:

Following is the list of important properties you would require to configure for a databases in a standalone situation:

S.N.	Properties and Description
1	**hibernate.dialect** This property makes Hibernate generate the appropriate SQL for the chosen database.
2	**hibernate.connection.driver_class** The JDBC driver class.
3	**hibernate.connection.url** The JDBC URL to the database instance.
4	**hibernate.connection.username** The database username.
5	**hibernate.connection.password** The database password.
6	**hibernate.connection.pool_size** Limits the number of connections waiting in the Hibernate database connection pool.

S.N.	Properties and Description
7	**hibernate.connection.autocommit** Allows autocommit mode to be used for the JDBC connection.

If you are using a database along with an application server and JNDI then you would have to configure the following properties:

S.N.	Properties and Description
1	**hibernate.connection.datasource** The JNDI name defined in the application server context you□re using for the application.
2	**hibernate.jndi.class** The InitialContext class for JNDI.
3	**hibernate.jndi.<JNDIpropertyname>** Passes any JNDI property you like to the JNDI *InitialContext*.
4	**hibernate.jndi.url** Provides the URL for JNDI.
5	**hibernate.connection.username** The database username.
6	**hibernate.connection.password** The database password.

Hibernate with MySQL Database:

MySQL is one of the most popular open-source database systems available today. Let us create **hibernate.cfg.xml** configuration file and place it in the root of your application's classpath. You would have to make sure that you have **testdb** database available in your MySQL database and you have a user **test** available to access the database.

The XML configuration file must conform to the Hibernate 3 Configuration DTD, which is available from http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
   <session-factory>
   <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
   </property>
   <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
   </property>

   <!-- Assume test is the database name -->
   <property name="hibernate.connection.url">
      jdbc:mysql://localhost/test
   </property>
   <property name="hibernate.connection.username">
      root
   </property>
   <property name="hibernate.connection.password">
      root123
   </property>

   <!-- List of XML mapping files -->
   <mapping resource="Employee.hbm.xml"/>
```

```
</session-factory>
</hibernate-configuration>
```

The above configuration file includes **<mapping>** tags which are related to hibernate-mapping file and we will see in next chapter what exactly is a hibernate mapping file and how and why do we use it. Following is the list of various important databases dialect property type:

Database	Dialect Property
DB2	org.hibernate.dialect.DB2Dialect
HSQLDB	org.hibernate.dialect.HSQLDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Informix	org.hibernate.dialect.InformixDialect
Ingres	org.hibernate.dialect.IngresDialect
Interbase	org.hibernate.dialect.InterbaseDialect
Microsoft SQL Server 2000	org.hibernate.dialect.SQLServerDialect
Microsoft SQL Server 2005	org.hibernate.dialect.SQLServer2005Dialect
Microsoft SQL Server 2008	org.hibernate.dialect.SQLServer2008Dialect
MySQL	org.hibernate.dialect.MySQLDialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle 11g	org.hibernate.dialect.Oracle10gDialect
Oracle 10g	org.hibernate.dialect.Oracle10gDialect
Oracle 9i	org.hibernate.dialect.Oracle9iDialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
Progress	org.hibernate.dialect.ProgressDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Sybase	org.hibernate.dialect.SybaseDialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect

Hibernate Examples

Let us try an example of using Hibernate to provide Java persistence in a standalone application. We will go through different steps involved in creating Java Application using Hibernate technology.

Create POJO Classes:

The first step in creating an application is to build the Java POJO class or classes, depending on the application that will be persisted to the database. Let us consider our Employee class with getXXX and setXXX methods to make it JavaBeans compliant class.

A POJO (Plain Old Java Object) is a Java object that doesn't extend or implement some specialized classes and interfaces respectively required

by the EJB framework. All normal Java objects are POJO.

When you design a classs to be persisted by Hibernate, it's important to provide JavaBeans compliant code as well as one attribute which would work as index like id attribute in the Employee class.

```java
public class Employee {
   private int id;
   private String firstName;
   private String lastName;
   private int salary;

   public Employee() {}
   public Employee(String fname, String lname, int salary) {
      this.firstName = fname;
      this.lastName = lname;
      this.salary = salary;
   }
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName( String first_name ) {
      this.firstName = first_name;
   }
   public String getLastName() {
      return lastName;
   }
   public void setLastName( String last_name ) {
      this.lastName = last_name;
   }
   public int getSalary() {
      return salary;
   }
   public void setSalary( int salary ) {
      this.salary = salary;
   }
}
```

Create Database Tables:

Second step would be creating tables in your database. There would be one table corresponding to each object you are willing to provide persistence. Consider above objects need to be stored and retrieved into the following RDBMS table:

```sql
create table EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

Create Mapping Configuration File:

This step is to create a mapping file that instructs Hibernate how to map the defined class or classes to the database tables.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
```

```xml
<hibernate-mapping>
    <class name="Employee" table="EMPLOYEE">
        <meta attribute="class-description">
            This class contains the employee detail.
        </meta>
        <id name="id" type="int" column="id">
            <generator />
        </id>
        <property name="firstName" column="first_name" type="string"/>
        <property name="lastName" column="last_name" type="string"/>
        <property name="salary" column="salary" type="int"/>
    </class>
</hibernate-mapping>
```

You should save the mapping document in a file with the format <classname>.hbm.xml. We saved our mapping document in the file Employee.hbm.xml. Let us see little detail about the mapping document:

- The mapping document is an XML document having <hibernate-mapping > as the root element which contains all the <class> elements.

- The **<class>** elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database table name is specified using the **table** attribute.

- The **<meta>** element is optional element and can be used to create the class description.

- The **<id>** element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<generator>** element within the id element is used to automatically generate the primary key values. Set the **class** attribute of the generator element is set to **native** to let hibernate pick up either **identity, sequence** or **hilo** algorithm to create primary key depending upon the capabilities of the underlying database.

- The **<property>** element is used to map a Java class property to a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

There are other attributes and elements available which will be used in a mapping document and I would try to cover as many as possible while discussing other Hibernate related topics.

Create Application Class:

Finally, we will create our application class with the main() method to run the application. We will use this application to save few Employee's records and then we will apply CRUD operations on those records.

```java
import java.util.List;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class ManageEmployee {
   private static SessionFactory factory;
   public static void main(String[] args) {
      try{
         factory = new Configuration().configure().buildSessionFactory();
      }catch (Throwable ex) {
         System.err.println("Failed to create sessionFactory object." + ex);
         throw new ExceptionInInitializerError(ex);
      }
```

```java
      ManageEmployee ME = new ManageEmployee();

      /* Add few employee records in database */
      Integer empID1 = ME.addEmployee("Zara", "Ali", 1000);
      Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);
      Integer empID3 = ME.addEmployee("John", "Paul", 10000);

      /* List down all the employees */
      ME.listEmployees();

      /* Update employee's records */
      ME.updateEmployee(empID1, 5000);

      /* Delete an employee from the database */
      ME.deleteEmployee(empID2);

      /* List down new list of the employees */
      ME.listEmployees();
   }
   /* Method to CREATE an employee in the database */
   public Integer addEmployee(String fname, String lname, int salary){
      Session session = factory.openSession();
      Transaction tx = null;
      Integer employeeID = null;
      try{
         tx = session.beginTransaction();
         Employee employee = new Employee(fname, lname, salary);
         employeeID = (Integer) session.save(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
      return employeeID;
   }
   /* Method to  READ all the employees */
   public void listEmployees( ){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         List employees = session.createQuery("FROM Employee").list();
         for (Iterator iterator =
                          employees.iterator(); iterator.hasNext();){
            Employee employee = (Employee) iterator.next();
            System.out.print("First Name: " + employee.getFirstName());
            System.out.print("  Last Name: " + employee.getLastName());
            System.out.println("  Salary: " + employee.getSalary());
         }
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
   /* Method to UPDATE salary for an employee */
   public void updateEmployee(Integer EmployeeID, int salary ){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         Employee employee =
                     (Employee)session.get(Employee.class, EmployeeID);
         employee.setSalary( salary );
   session.update(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
```

```
      }finally {
         session.close();
      }
   }
   /* Method to DELETE an employee from the records */
   public void deleteEmployee(Integer EmployeeID){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         Employee employee =
               (Employee)session.get(Employee.class, EmployeeID);
         session.delete(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
}
```

Compilation and Execution:

Here are the steps to compile and run the above mentioned application. Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create hibernate.cfg.xml configuration file as explained in configuration chapter.

- Create Employee.hbm.xml mapping file as shown above.

- Create Employee.java source file as shown above and compile it.

- Create ManageEmployee.java source file as shown above and compile it.

- Execute ManageEmployee binary to run the program.

You would get following result, and records would be created in EMPLOYEE table.

```
$java ManageEmployee
.......VARIOUS LOG MESSAGES WILL DISPLAY HERE........

First Name: Zara  Last Name: Ali  Salary: 1000
First Name: Daisy  Last Name: Das  Salary: 5000
First Name: John  Last Name: Paul  Salary: 10000
First Name: Zara  Last Name: Ali  Salary: 5000
First Name: John  Last Name: Paul  Salary: 10000
```

If you check your EMPLOYEE table, it should have following records:

```
mysql> select * from EMPLOYEE;
+----+------------+-----------+--------+
| id | first_name | last_name | salary |
+----+------------+-----------+--------+
| 29 | Zara       | Ali       |   5000 |
| 31 | John       | Paul      |  10000 |
+----+------------+-----------+--------+
2 rows in set (0.00 sec

mysql>
```