# Hibernate Framework

Session-1

# What is Persistence

- **Persistence** is the ability of an object to survive beyond the lifetime of the program that creates it.

- Persistence is relevant for objects with an internal state. The state needs to be retained between object deactivation and object activation.
  - Storing object state on persistent storage before de-activation
  - Upon activation, load object state from persistent storage

- For an object-oriented language like Java, persistence ensures that the state of an object is accessible even after the application that created it has stopped executing.

- Persistent storage can be
  a. File system b. Relational Database c. Object-Database d. Flash-RAM

# JDBC & Persistence

Java applications traditionally used **JDBC (Java Database Connectivity) API** to persist data into **relational databases**.

The JDBC API uses SQL statements to perform create, read, update, and delete (CRUD) operations. JDBC code is embedded in Java classes -- in other words, it's tightly coupled to the business logic.

This code also relies heavily on SQL, which is not standardized across databases; that makes migrating from one database to another difficult.

*Relational database technology emphasizes data and its relationships, whereas the object-oriented paradigm used in Java concentrates not only on the data  but also on the operations performed on that data.*

# Object-Relational Mapping (ORM)

***Object-relational mapping* (ORM) is the solution.**

ORM is a technique that transparently persists *application objects to the tables in a relational database* where in *entities/classes are mapped to tables*, *instances are mapped to rows* and *attributes of instances are mapped to columns of table*.

- ORM behaves like a virtual database, hiding the underlying database architecture from the user.
- It provides functionality to perform complete CRUD operations and encourages object-oriented querying.
- Also supports metadata mapping and helps in the transaction management of the application.

# Hibernate Framework

- Hibernate is an open-source ORM solution for Java applications.

- **Hibernate** is an object-relational mapping (ORM) library for the Java language, providing a **framework** for mapping an object-oriented domain model to a traditional relational database.

- Hibernate was developed by a team headed by **Gavin King**. The development of Hibernate began in 2001 and the team was later acquired by JBoss, which now manages it.

- Hibernate was developed initially for Java; in 2005 a .Net version named **NHibernate** was introduced.

- The current version of Hibernate is 4.x, and it supports Java annotations.

# Hibernate Framework

Some of the benefits of using Hibernate as ORM tool are:

1.Hibernate supports mapping of java classes to database tables and vice versa. It provides features to perform CRUD operations across all the major relational databases.

2.Hibernate eliminates all the boiler-plate code that comes with JDBC and takes care of managing resources, so we can focus on business use cases rather than making sure that database operations are not causing resource leaks.

3.Hibernate supports transaction management and make sure there is no inconsistent data present in the system.

4.Since we use XML, property files or annotations for mapping java classes to database tables, it provides an abstraction layer between application and database.

5.Hibernate helps us in mapping joins, collections, inheritance objects and we can easily visualize how our model classes are representing database tables..

# Hibernate Framework

1.Hibernate provides a powerful query language (HQL) that is similar to SQL. However, HQL is fully object-oriented and understands concepts like inheritance, polymorphism and association.

2.Hibernate also offers integration with some external modules. For example Hibernate Validator is the reference implementation of Bean Validation (JSR 303).

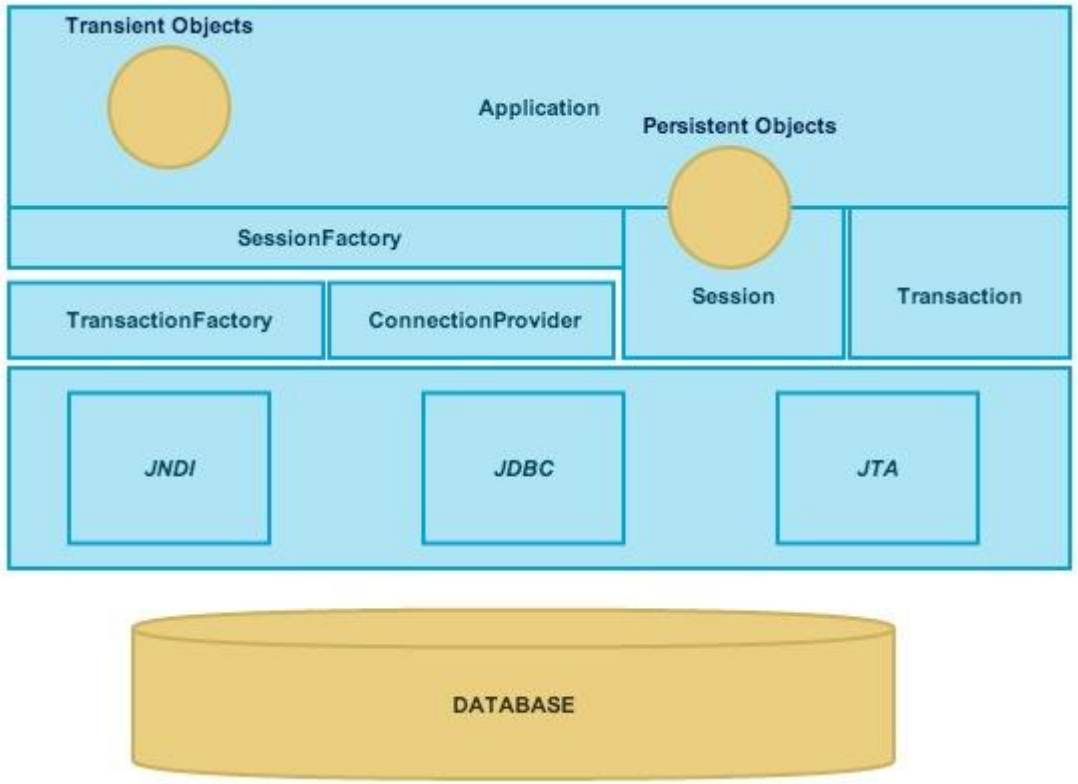
3.Hibernate is an open source project from Red Hat Community and used worldwide. This makes it a better choice than others because learning curve is small and there are tons of online documentations and help is easily available in forums.

4.Hibernate is easy to integrate with other Java EE frameworks, it's so popular that Spring Framework provides built-in support for integrating hibernate with Spring applications.

# Hibernate Architecture

***The following diagram shows minimal architecture of Hibernate:***

It works as an abstraction layer between application classes and JDBC/JTA APIs for database operations. It's clear that Hibernate is built on top of JDBC and JTA APIs.




It creates a layer between Database and the Application. It loads the configuration details like Database connection string, entity classes, mappings etc.

**hibernate.cfg.xml which is preferred** (alternatively, we can use **hibernate.properties**):

These two files are used to configure the hibernate service (connection driver class, connection URL, connection username, connection password, dialect etc).

*If both files are present in the classpath then* ***hibernate.cfg.xml*** *file overrides the settings found in the* ***hibernate.properties*** *file.*

# Core components of hibernate architecture

- **SessionFactory (org.hibernate.SessionFactory)**: SessionFactory is an immutable thread-safe cache of compiled mappings for a single database. We can get instance of org.hibernate.Session using SessionFactory.

- **Session (org.hibernate.Session)**: Session is a single-threaded, short-lived object representing a conversation between the application and the persistent store. It wraps JDBC java.sql.Connection and works as a factory for org.hibernate.Transaction.

- **Persistent objects**: Persistent objects are short-lived, single threaded objects that contains persistent state and business function. These can be ordinary JavaBeans/POJOs. They are associated with exactly one org.hibernate.Session.

- **Transient objects**: Transient objects are persistent classes instances that are not currently associated with a org.hibernate.Session. They may have been instantiated by the application and not yet persisted, or they may have been instantiated by a closed org.hibernate.Session.

# Core components of hibernate architecture

- **Transaction (org.hibernate.Transaction)**: Transaction is a single-threaded, short-lived object used by the application to specify atomic units of work. It abstracts the application from the underlying JDBC or JTA transaction. A org.hibernate.Session might span multiple org.hibernate.Transaction in some cases.

- **ConnectionProvider (org.hibernate.connection.ConnectionProvider)**: ConnectionProvider is a factory for JDBC connections. It provides abstraction between the application and underlying javax.sql.DataSource or java.sql.DriverManager. It is not exposed to application, but it can be extended by the developer.

- **TransactionFactory (org.hibernate.TransactionFactory)**: A factory for org.hibernate.Transaction instances.

# Hibernate Query Language(HQL)

The **HQL or Hibernate Query language** is powerful and can be easily used to develop complex query in Java program. HQL is then translated into SQL by the ORM component of Hibernate.

Hibernate generates JDBC code based on the underlying database chosen, and so saves you the trouble of writing JDBC code. It also supports connection pooling.

Although it is possible to use native SQL queries directly with a Hibernate-based persistence layer, it is more efficient to use HQL instead.

# Hibernate Query Language(HQL)

1. HQL is similar to SQL and is also case insensitive.

2. HQL and SQL both fire queries in a database. In the case of HQL, the queries are in the form of objects that are translated to SQL queries in the target database.

3. SQL works with tables and columns to manipulate the data stored in it.

4. HQL works with classes and their properties to finally be mapped to a table structure in a database.

5. HQL supports concepts like polymorphism, inheritance, association, etc. It is a powerful and easy-to-learn language that makes SQL object oriented.

6. HQL lets us modify the data through insert, update, and delete queries. You can add tables, procedures, or views to your database. The permissions on these added objects can be changed.

# Hibernate Configuration

**The configuration file aids in establishing a connection to a particular relational database.**

- *Hibernate provides following types of configurations*
  - **hibernate.cfg.xml** – A standard XML file which contains hibernate configuration and which resides in root of application's CLASSPATH
  - **hibernate.properties** – A Java compliant property file which holds key value pair for different hibernate configuration strings.
  - **Programmatic configuration** – By invoking the methods of org.hibernate.cfg.Configuration instance
  - Note: **hibernate.cfg.xml is preferred method.**

# 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">
<hibernate-configuration>
<session-factory>
   <!-- hibernate dialect -->
   <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
   <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
   <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/tutorial</property>
   <property name="hibernate.connection.username">root</property>
   <property name="hibernate.connection.password">root</property>
   <property name="transaction.factory_class">org.hibernate.transaction.JDBCTransactionFactory</property>
    <property name="hibernate.current_session_context_class">thread</property>
   <!-- Automatic schema creation (begin) === -->
   <!-- <property name="hibernate.hbm2ddl.auto">create</property>  -->
    <!-- # mapping files with external dependencies # -->
   <mapping resource="org/asr/hibernate//Person.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

*For the latest hibernate mappings:*
http://www.jboss.org/dtd/hibernate/

**SessionFactory sessionFactory = new Configuration().configure().buildSessionFacto**

The above statement will load default **hibernate** and all the configuration mentioned in it.

Note: The Configuration class buildSessionFacto is **deprecated** and is recommended to use **buildSessionFactory(ServiceRegistry)**

https://community.jboss.org/wiki/SessionsAndTransactions

# hibernate.properties

**Below is the sample hibernate.properties file:**

```
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url= jdbc:mysql://localhost:3306/tutorial
hibernate.connection.username=root
hibernate.connection.password=root
hibernate.connection.pool_size=1
hibernate.transaction.factory_class = \org.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \org.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = org.hibernate.dialect.MySQLDialect
```

# Building a SessionFactory

```
import org.hibernate.SessionFactory;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.service.ServiceRegistry;

public class HibernateUtils {

    private static final SessionFactory sessionFactory = buildSessionFactory();
    // Hibernate 5:
    private static SessionFactory buildSessionFactory() {
        try {
            // Create the ServiceRegistry from hibernate.cfg.xml
            ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()//
                    .configure("hibernate.cfg.xml").build();
            // Create a metadata sources using the specified service registry.
            Metadata metadata = new
MetadataSources(serviceRegistry).getMetadataBuilder().build();
            return metadata.getSessionFactoryBuilder().build();
        }
```

Contd..

# Building a SessionFactory

```
catch (Throwable ex) {

            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    public static void shutdown() {
        // Close caches and connection pools
        getSessionFactory().close();
    }

}
```

# Getting Session instance

session represents a communication channel between database and application. Each session represents a set of transactions.

Hibernate SessionFactory provides three methods through which we can acquire Session object –

- **getCurrentSession(),**
- **openSession() and**
- **openStatelessSession().**

**Ex.**

```
Session session = sessionFactory.openSession(); // get a new Session
```

The differences between these methods will be covered later.

# getCurrentSession()

## Hibernate getCurrentSession

Hibernate SessionFactory getCurrentSession() method returns the session bound to the context. But for this to work, we need to configure it in hibernate configuration file like below.

```
<property name="hibernate.current_session_context_class">thread</property>
```

If not configured to thread, exception is thrown.

Since this session object belongs to the hibernate context, we don't need to close it. Once the session factory is closed, this session object gets closed.

Hibernate Session objects are not thread safe, so we should not use it in multi-threaded environment.

We can use it in single threaded environment because it's relatively faster than opening a new session.

# openSession()

**Hibernate openSession**

Hibernate SessionFactory openSession() method always opens a new session.

We should close this session object once we are done with all the database operations.

We should open a new session for each request in multi-threaded environment.

For web application frameworks, we can choose to open a new session for each request or for each session based on the requirement.

# Mapping File - Outdated

Mapping files specify how classes and objects are mapped to  he database table and columns.

- Traditionally, there is one mapping file per class.
- The name of the mapping file conventionally takes the form **Classname.hbm.xml**.
- For example, the *Customer* class has a *Customer.hbm.xml* mapping file.
- The mapping file is conventionally stored in the same package with the domain class it maps to the database.
- The mapping file must follow Hibernate document type definition (DTD) (currently version 3.0).

The mapping file that the configuration file identifies maps a particular persistent class to the database table.

It maps specific columns to specific fields, and has associations, collections, primary key mapping, and ID key generation mechanisms.

# The Mapping File Example: Person.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd>
<hibernate-mapping>
      <class name= " org.asr.Person" table="Person">
            <id name="id" column="ID">
                  <generator class="native" />
            </id>
            <property name="name">
                  <column name="NAME" length="16" not-null="true" />
            </property>
            <property name="surname">
                  <column name="SURNAME" length="16" not-null="true" />
            </property>
            <property name="address">
                  <column name="ADDRESS" length="16" not-null="true" />
            </property>
      </class>
</hibernate-mapping>
```

**Outdated**

# Hibernate Generator

**1. Increment**

The increment generator is probably the most familiar creator of IDs. Each time the generator needs to generate an ID, It performs a select on current database, determines the current largest ID value, and increment of next value.

**Syntax:**

```
<generator class="increment"/>
```

**2. Identity**

If the database has identity column associated with it.

**Syntax:**

```
<generator class="identity" >
        <param name="identity">identity_name</param>
</generator>
```

**3. Sequence**

If the database has sequence column associated with it.

**Syntax**

```
<generator class="sequence">
        <param name="sequence">seq_name</param>
</generator>
```

# Hibernate Generator

## 4. Hilo

It generates unique IDs for database table. The IDs won't necessarily be sequential. The generator must have access to secondary table. The default table is **hibernate_unique_key** and default column is **next_hi**. We need to insert one row in the table as below example.

**Example**

**•Create table hibernate_unique_key(next_hi  int);**

**•insert into demo.`hibernate_unique_key`values(100);**

**•Syntax**

```
<generator class="hilo"/>    //it will use hibernate-unique-key table
    <param name="table">TABLE_NAME</param>
    <param name="column">COLUMN_NAME</param>
    <param name="max to">500</param>
</generator>
```

## 5. Native

It picks **identity, sequence or hilo** depending on the database.

**Syntax**

```
<generator class="native"/>
```

## 6. Assigned

If you want to assign the identifier manually in the application, then use the assigned generator.

**Syntax**

```
<generator class="assigned"/>
```

# About Dialects

```
.....
<hibernate-configuration>
<session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    ..........
</session-factory>
</hibernate-configuration>
```

Databases have subtle differences in the SQL they use. Things such as data types for example vary across databases (e.g. in Oracle we put an integer value in a **number** field and in SQL Server use an **int** field) Or database specific functionality - selecting the top n rows is different depending on the database.

**The dialect abstracts this so we don't have to worry about it.**

List of Hibernate SQL Dialects

| RDBMS | Dialect |
|---|---|
| DB2 | org.hibernate.dialect.DB2Dialect |
| DB2 AS/400 | org.hibernate.dialect.DB2400Dialect |
| DB2 OS390 | org.hibernate.dialect.DB2390Dialect |
| PostgreSQL | org.hibernate.dialect.PostgreSQLDialect |
| MySQL | org.hibernate.dialect.MySQLDialect |
| MySQL with InnoDB | org.hibernate.dialect.MySQLInnoDBDialect |
| MySQL with MyISAM | org.hibernate.dialect.MySQLMyISAMDialect |
| Oracle (any version) | org.hibernate.dialect.OracleDialect |
| Oracle 9i/10g | org.hibernate.dialect.Oracle9Dialect |
| Sybase | org.hibernate.dialect.SybaseDialect |
| Sybase Anywhere | org.hibernate.dialect.SybaseAnywhereDialect |
| Microsoft SQL Server | org.hibernate.dialect.SQLServerDialect |
| 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 |

# About transaction.factory_class

```
…….
<hibernate-configuration>
<session-factory>
    <property name="transaction.factory_class">org.hibernate.transaction.JDBCTransactionFactory</property>
    …….
</session-factory>
</hibernate-configuration>
```

- The transaction creation follows ***a factory pattern***. For example, in an environment where application servers (like JBOSS, WebSphere etc.) manage the transaction creation via JTA APIs, we will use a JTA Transaction factory to create transactions and that transactions will be further used by hibernate.
- Hibernate transaction factory creates database transactions on underlying DB.
- A session may be spanning over multiple transactions - A hibernate session abstracts a database connection.
- Over the same connection, multiple begin transaction, commit transaction cycles are possible.
- Transactions are required for any writes, deletes, protected reads. So the session holds an implicit transaction.
- In short, we need transactions when we want other processes to be stopped from interfering with our data between SQL statements.

# About current_session_context_class

…….

```
<hibernate-configuration>
<session-factory>
    <property name="hibernate.current_session_context_class">thread</property>  …….
</session-factory>
</hibernate-configuration>
```

Contextual Sessions in Hibernate is basically about context in which we want Hibernate to manage the scope of your transactions. This context drives the mechanics of sessionFactory.getCurrentSession() method.

*Three possible configs for hibernate.current_session_context_class:*

- "jta" context = an already existing jta transaction
- "thread" context = current thread (think ThreadLocal)
- "managed" context = custom to your domain

*Also:*

- "jta" and "thread" are supported by hibernate out of box
- "thread" context is used in most stand alone hibernate apps or those based on light weight frameworks like spring
- "jta" is used in Java EE environments

*https://community.jboss.org/wiki/SessionsAndTransactions*

# About hibernate.hbm2ddl.auto

```
.....
<hibernate-configuration>
<session-factory>
 <property name= "show_sql">true</property>
 <property name="hibernate.hbm2ddl.auto">update</property>
</session-factory>
</hibernate-configuration>
```

| | |
|---|---|
| hibernate.show_sql | Write all SQL statements to console. This is an alternative to setting the log category org.hibernate.SQL to debug.**e.g.** true \| false |

Automatically validates or exports schema DDL to the database when the SessionFactory is created.

With **create-drop**, the database schema will be dropped when the SessionFactory is closed explicitly.

**e.g.** validate | update | create | create-drop

# CustomerTester.java

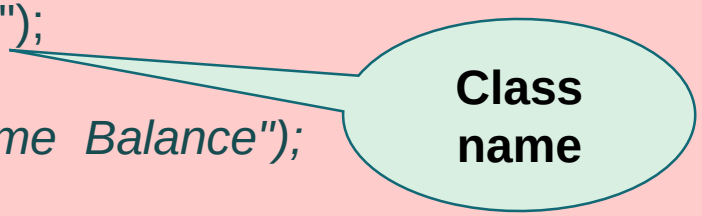
```
public class CustomerTester {
public static void main(String[] args) {
Session session = SessionFactoryUtility.getSessionFactory().getCurrentSession();

session.beginTransaction();
createCustomer(session);
System.out.println("1 row(s) inserted");

System.out.println("--------------------------------");
listCustomers(session);
System.out.println("--------------------------------");
session.getTransaction().commit();   }
```

```
public static void createCustomer(Session session) {
        Customer customer = new Customer();
        customer.setFirstname("Srinivas");
        customer.setLastname("Reddy");
        customer.setBalance(6450.50);
        session.save(customer);
}
```

```
private static void queryCustomer(Session session) {
        //In session.createQuery("query") method, give the class name, not table name, because it is HQL not SQL
         Query query=session.createQuery("from Customer");
        List<Customer> list = query.list();
        System.out.println("Customer ID  Firstname  Lastname  Balance");
        Iterator<Customer> iterator = list.iterator();
        while(iterator.hasNext()){
                System.out.println(iterator.next());
        }
}
```



# 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">

<hibernate-configuration>
 <session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect</property>
    <property name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
    <property name="hibernate.connection.url">jdbc:oracle:thin:@localhost:1521:myoracle</property>
    <property name="hibernate.connection.username">scott</property>
    <property name="hibernate.connection.password">tiger</property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.hbm2ddl.auto">update</property>
    <property name="hibernate.connection.release_mode">auto</property>
    <property name="current_session_context_class">thread</property>
    <property name="hibernate.connection.autoReconnect">true</property>

    <mapping class="com.capgemini.hibernate.entity.User" />
 </session-factory>

</hibernate-configuration>
```

**Create *hibernate.cfg.xml* file in the folder src/main/resources.**

**Create the folder if it doesn't exist.**

# Managing Dependency

**Update pom.xml file**

```
<groupId>org.asr</groupId>
<artifactId>HibbernateProject</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

<name>HibbernateProject</name>
<url>http://maven.apache.org</url>

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>4.3.6.Final</version>
</dependency>
```

```
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>5.1.30</version>
</dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>


</dependencies>
```

# Thank You!