

UNIT - IV

Hibernate Introduction, Hibernate Configuration, Hibernate Concepts, Hibernate O-R Mapping, Manipulating and Querying, Hibernate Query Language, Criteria Queries, Native SQL, Transaction and Concurrency

Introduction to Hibernate:

Hibernate is the Object-Relational Mapping (ORM) **framework in Java** created by Gavin King in 2001. It simplifies the interaction of a database and the Java application being developed. It is an ORM tool that is powerful and lightweight. Another important thing is that this is a high-performance open-source tool. Hibernate implements Java Persistence API specifications and is a powerful object-relational persistence and query service for applications developed in Java.

Hibernate is an ORM tool that maps database structures with Java objects dynamically at runtime. Using Hibernate, a persistent framework, allows the developers to focus on just business logic code writing despite writing accurately, as well as a good persistence layer that consists of writing the **SQL Queries**, connection management, and JDBC Code.

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

Hibernate sits between traditional Java objects and database server to handle all the works 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 the database or in any table, then you need to change the XML file properties only.
- Abstracts away the unfamiliar SQL types and provides a way to work around familiar Java Objects.
- Hibernate does not require an application server to operate.
- Manipulates Complex associations of objects of your database.
- Minimizes database access with smart fetching strategies.
- Provides simple querying of data.

Supported Databases

Hibernate supports almost all the major RDBMS. Following is a 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 Configuration: Hibernate Configuration Approaches,

Hibernate needs to know about the database configurations to connect to the database. There are three approaches with which we can do the configurations. These approaches are-

- a. Programmatic Configurations – Hibernate does provides a way to load and configure database and connection details programmatically.
- b. XML configurations – We can provide the database details in an XML file. By default hibernate loads the file with name hibernate.cfg.xml but we can load the file with custom name as well.
- c. Properties configurations- This approach uses a properties file for the configuration. By default hibernate loads the file with name hibernate.properties but we can load the file with custom name as well.

Programmatic Configurations

Hibernate does provide a Configuration class which provides certain methods to load the mapping files and database configurations programmatically .

a.Loading Mapping Files

To load the mapping files (also known as hbm files) there are three ways–

1. **addResource()** – We can call addResource() method and pass the path of mapping file available in a classpath. To load multiple mapping files, simply call addResources() method multiple times.

Below code snippet uses addResource() method to load user.hbm.xml and account.hbm.xml file available in com.hibernate.tutorial package.

```
Configuration configuration = new Configuration();  
configuration.addResource("com/hibernate/tutorial/user.hbm.xml");  
configuration.addResource("com/hibernate/tutorial/account.hbm.xml ");
```

2. **addClass()**- Alternatively we can call addClass() method and pass in the class name which needs to persist. To add multiple classes , we can call addClass() multiple times.

Below code snippet uses addClass() method to load user and account classes available in com.hibernate.tutorial package.

```
Configuration configuration = new Configuration();  
configuration.addClass("com.hibernate.tutorial.user.class");  
configuration.addClass("com.hibernate.tutorial.user.account.class ");
```

3. **addJar()** – We can call addJar() to specify the path of jar file containing all the mapping files. This approach provides a generic way and need not to add mapping every time we add a new mapping.

```
Configuration configuration = new Configuration();  
configuration.addJar(new File("mapping.jar"))
```

b. Loading Database Configurations

In earlier section we saw how to load the mapping files programmatically, but along with mapping files, Hibernate requires database configurations as well.

We can use the Configuration object to load the database configurations from an instance of properties files, System properties or even set the database properties directly. Lets discuss all approaches in details.

setProperty()- we can call setProperty() method on configuration object and pass the individual property as a key value pair. We can call setProperty() multiple times.

Below code specifies hibernate dialects , driver class , database connection url, database credentials using setProperty() method.

```
<strong><strong><strong><strong>Configuration configuration = new Configuration();  
configuration.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLDialect");  
configuration.setProperty("hibernate.connection.driver_class", "com.mysql.jdbc.Driver");  
configuration.setProperty("hibernate.connection.url", "jdbc:mysql://localhost:3306/tutorial");  
configuration.setProperty("hibernate.connection.username", "root");  
configuration.setProperty("hibernate.connection.password", "password");  
</strong></strong></strong></strong>
```

b.setProperty()- we can call setProperties() method on the configuration object to load the properties from system properties or can pass the property file instance explicitly.

Below code specifies load the database configurations from System properties. All the database configurations will be configured using “Java -Dproperty=value “

```
Configuration configuration = new Configuration();  
configuration.setProperties(System.getProperties());
```

Alternatively, we can create a properties file having database configurations and pass its instance in setProperties()

So we saw how we can load both mapping files and database configuration files programmatically so together the code snippet looks like below

```

<strong><strong><strong><strong>Configuration configuration = new
Configuration();
configuration.addResource("com/hibernate/tutorial/user.hbm.xml");
configuration.addResource("com/hibernate/tutorial/account.hbm.xml ");
configuration.setProperty("hibernate.dialect", "
org.hibernate.dialect.MySQLDialect");
configuration.setProperty("hibernate.connection.driver_class",
"com.mysql.jdbc.Driver");
configuration.setProperty("hibernate.connection.url", "
jdbc:mysql://localhost:3306/tutorial");
configuration.setProperty("hibernate.connection.username", "root");
configuration.setProperty("hibernate.connection.password", "password");
</strong></strong></strong></strong>

```

XML Configurations

The XML configuration approach is widely used and Hibernate loads the configurations from a file with name hibernate.cfg.xml from a class path. Alternatively, we can create an XML file with another name and pass the name of the file.

The sample XML file looks like below-

```

<strong><strong><strong><strong><?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.url">
            jdbc:mysql://localhost:3306/tutorial
        </property>
        <property name="hibernate.connection.username">
            root
        </property>
        <property name="hibernate.connection.password">
            password
        </property>
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">false</property>
        <property name="hibernate.connection.driver_class">
            com.mysql.jdbc.Driver
        </property>

        <mapping resource="user.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
</strong></strong></strong></strong>

```

To add mapping resources we can use <mapping resource> tag in hibernate.cfg.xml file .This is similar to addResource() method. Similarly, we can use <mapping jar> and <mapping class> tags for addJar() and addClass()

Just need to instantiate Configuration object like below

```
Configuration configuration = new Configuration().configure();
```

new Configuration() call will load the hibernate.properties file and calling configure() method on configuration object loads hibernate.cfg.xml. In case any property is defined in both hibernate.properties and hibernate.cfg.xml file then xml file will get precedence.

To load the custom files, we can call

```
Configuration configuration = new  
Configuration().configure("/configurations/myConfiguration.cfg.xml")
```

The Above code snippet will load myConfiguration.cfg.xml from the configuration subdirectory of class path.

Note: We can skip prefix “hibernate” from the hibernate properties like hibernate.connection.password is equivalent to connection.password

Properties file Configurations

Hibernate looks for a file named hibernate.properties file in the class path. Properties file provides the similar functionality as XML file provides with the difference of “ we cannot add a mapping resource in properties file)

Below is the sample content of hibernate.properties file

```
<strong><strong><strong><strong>hibernate.connection.url =  
jdbc:mysql://localhost:3306/tutorial hibernate.connection.username = root  
hibernate.connection.password = password  
hibernate.dialect = org.hibernate.dialect.MySQLDialect  
hibernate.show_sql = true</strong></strong></strong></strong>
```

Commonly Used Hibernate Properties

Hibernate supports many properties to configure database, connections, transactions, cache properties which can be referred at Hibernate Official Documentation but the most commonly used are-

Property Name	Description
hibernate.cache.provider_class	We can specify the implementation of org.hibernate.cache.CacheProvider implementation. This property is used to specify the cache implementation to use.
hibernate.cache.use_query_cache	To specify whether to use or not Query Cache
hibernate.cache.use_second_level_cache	To specify whether to use or not Second level Cache

Property Name	Description
hibernate.connection.autocommit	To set auto commit mode of connection
hibernate.connection.datasource	Specify the datasource to use
hibernate.connection.driver_class	JDBC Driver class
hibernate.connection.password	Database password
hibernate.connection.username	Database username
hibernate.connection.pool_size	To limit the number of connections
hibernate.connection.url	Specify the database connection URL
hibernate.dialect	Let Hibernate know which Dialect to use
hibernate.hbm2ddl.auto	<p>This property is used to specify if Hibernate to create , update , drop automatically when the application starts. There are five possible values –</p> <p>a) create-create the tables based on mapping files when application starts and removes if already exists.</p> <p>b) create-drop- drop the tables when session factory is closed and create when application starts based on mapping files.</p> <p>c) update – update the tables if already exists or create if not based on the mapping files.</p> <p>d) validate – does not create table validates the table against mapping files. Gives errors if mismatch</p> <p>e) none = does nothing.</p> <p>NOTE- This option works on Tables not on SCHEMA</p>
hibernate.show_sql	Logs the generated SQL on console. Possible values are true or false
hibernate.format_sql	Logs the formatted generated SQL on console. Possible values are true or

Property Name	Description
	false

Example: Hibernate Configuration: Web Application with Hibernate (using XML)

1. Web Application with Hibernate
2. Example to create web application using hibernate

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.

```
<form action="register.jsp" method="post">
Name:<input type="text" name="name"/> <br> <br>
Password:<input type="password" name="password"/> <br> <br>
Email ID:<input type="text" name="email"/> <br> <br>
<input type="submit" value="register"/>

</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.

```
<%@page import="com.javatpoint.mypack.UserDao"%>
<jsp:useBean id="obj" class="com.javatpoint.mypack.User">
</jsp:useBean>
<jsp:setProperty property="*" name="obj"/>
<%
int i=UserDao.register(obj);
if(i>0)
out.print("You are successfully registered");
%>
```

User.java

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

```
package com.javatpoint.mypack;
public class User {
private int id;
private String name,password,email;
//getters and setters
}
```

user.hbm.xml

It maps the User class with the table of the database.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 5.3//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-5.3.dtd">

<hibernate-mapping>
<class name="com.javatpoint.mypack.User" table="u400">
<id name="id">
```

```

<generator class="increment"></generator>
</id>
<property name="name"></property>
<property name="password"></property>
<property name="email"></property>
</class>

</hibernate-mapping>

```

UserDao.java

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

```

package com.javatpoint.mypack;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class UserDao {

    public static int register(User u){
        int i=0;

        StandardServiceRegistry ssr = new StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
        Metadata meta = new MetadataSources(ssr).getMetadataBuilder().build();

        SessionFactory factory = meta.getSessionFactoryBuilder().build();
        Session session = factory.openSession();
        Transaction t = session.beginTransaction();

        i=(Integer)session.save(u);
    }
}

```

```
t.commit();
session.close();
```

```
return i;
```

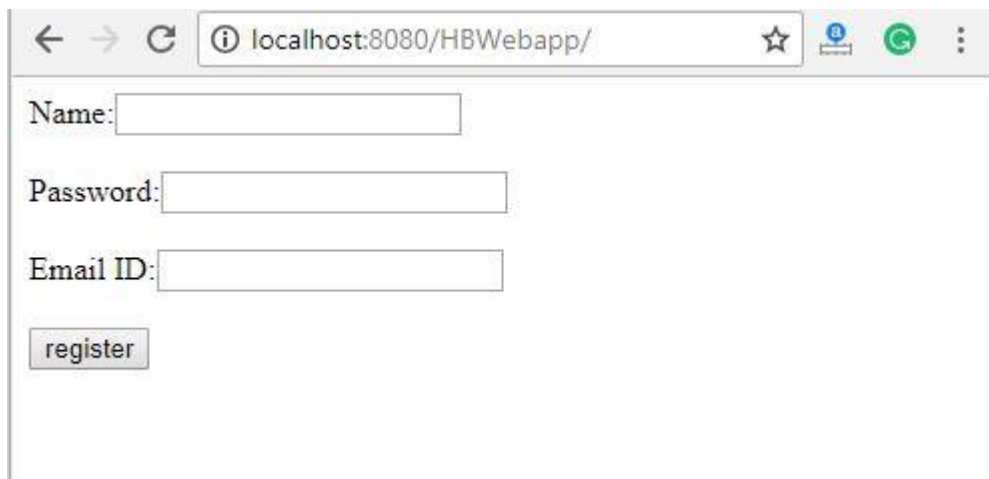
```
}
}
```

hibernate.cfg.xml

It is a configuration file, containing informations about the database and mapping file.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">
<hibernate-configuration>
<session-factory>
<property name="hbm2ddl.auto">create</property>
<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
<property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
<property name="connection.username">system</property>
<property name="connection.password">jtp</property>
<property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
    <mapping resource="user.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

Output

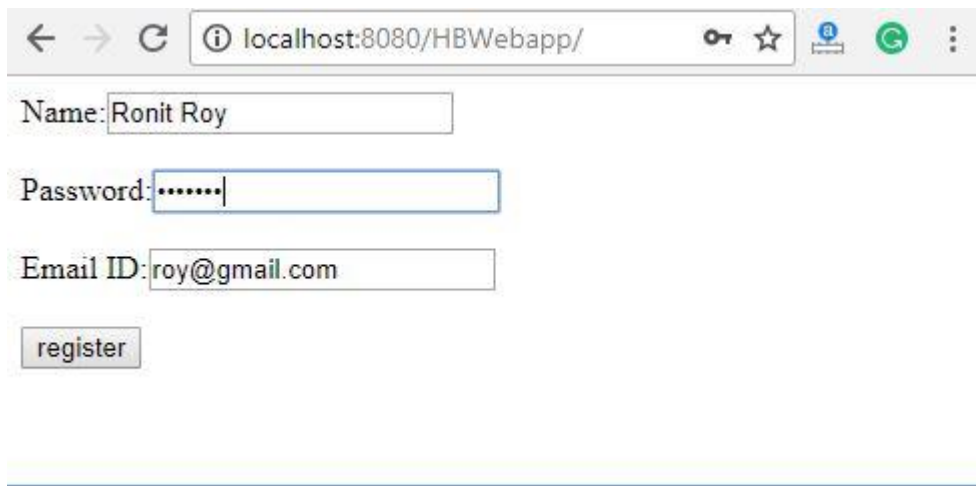


localhost:8080/HBWebapp/

Name:

Password:

Email ID:



localhost:8080/HBWebapp/

Name:

Password:

Email ID:

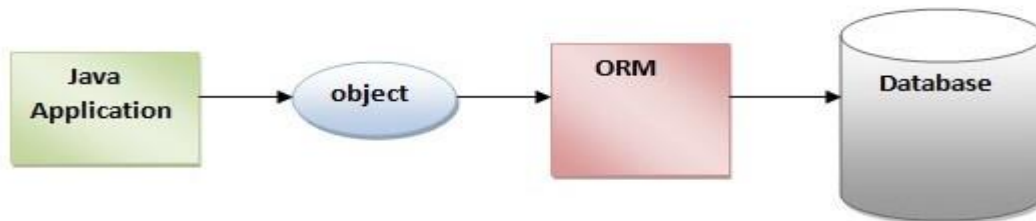
Output: You are successfully registered

Hibernate Concepts:

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.

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.

6) Provides Query Statistics and Database Status

Hibernate supports Query cache and provide statistics about query and database status.

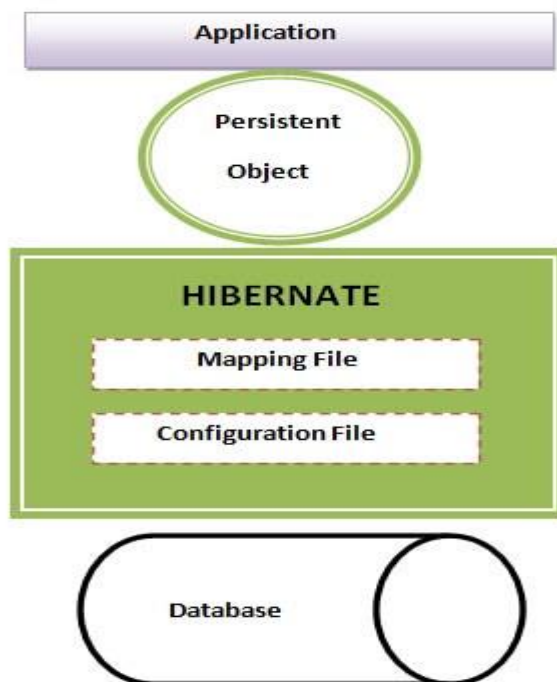
Hibernate Architecture:

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

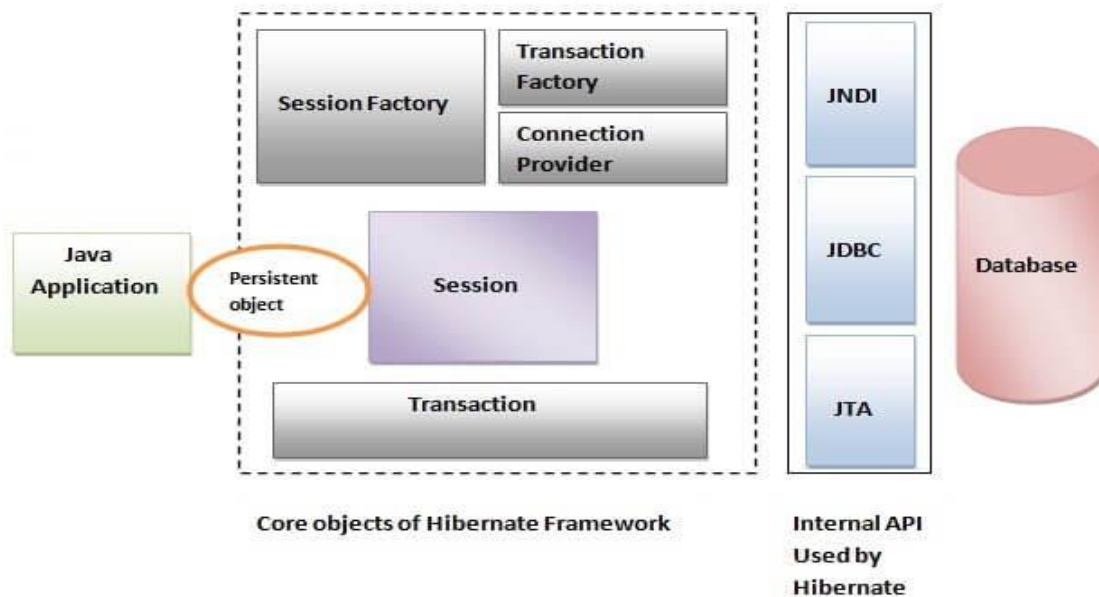
The Hibernate architecture is categorized in four layers.

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

Let's see the diagram of hibernate architecture:



This is the high level architecture of Hibernate with mapping file and configuration file.



Hibernate framework uses many objects such as session factory, session, transaction etc. along with existing Java API such as JDBC (Java Database Connectivity), JTA (Java Transaction API) and JNDI (Java Naming Directory Interface).

Elements of Hibernate Architecture

For creating the first hibernate application, we must know the elements of Hibernate architecture. 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.

ConnectionProvider

It is a factory of JDBC connections. It abstracts the application from DriverManager or DataSource. It is optional.

TransactionFactory

It is a factory of Transaction. It is optional.

Hibernate O-R Mapping:

Hibernate is a free and open-source object-relational mapper for Java. This simple approach gets rid of all the problems with JDBC. Hibernate develops persistence logic, which arranges and prepares the data for subsequent use. Its advantages over other frameworks include being open-source, portable, and an ORM tool. Hibernate mappings are one of the key features of hibernate and they establish the relationship between two database tables as attributes in your model that allows you to easily navigate the associations in your model and criteria queries.

O/RM itself can be defined as a software or product that represents and/or convert the data between the application (written in Object-Oriented Language) and the database. In other words we can say O/RM maps an object to the relational database table. Hibernate is also an O/RM and is available as open source project. To use Hibernate with Java you will be required to create a file with .hbm.xml extension into which the mapping information will be provided. This file contains the mapping of Object with the relational table that provides the information to the Hibernate at the time of persisting the data. ORM abstracts the application from the process related to database such as saving, updating, deleting of objects from the relational database table.

Example :

Employee.java

```
package srinivas;
```

```
public class Employee
```

```
{
```

```
    private long empId;
```

```
    private String empName;
```

```
    public Employee() {
```

```
    }
```

```
    public long getEmpId() {
```

```
        return this.empId;
```

```
    }
```



```

public void setEmpId(long empId) {

    this.empId = empId;

}

public String getEmpName() {

    return this.empName;

}

public void setEmpName(String empName) {

    this.empName = empName;

}

}

```

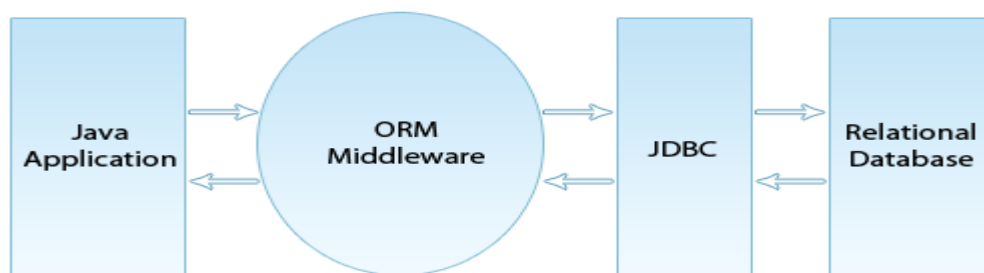
employee.hbm.xml

```

<?xml version="1.0"?>
<!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="srinivas.Employee" table="employee">
<id name="empId" type="long" column="Id" >
<generator class="assigned"/>
</id>
<property name="empName">
<column name="Name" />
</property>
</class>
</hibernate-mapping>

```

The figure given below demonstrate you how ORM abstracts the application to the database related process and vice-versa :



O – R Mapping process

The above figure demonstrates that an ORM abstracts the both in respect to database how an application tries to persist the data and in respect to an application that how objects are stored in database.

Hibernate Mapping Types:

The Java data types are mapped into RDBMS data types when you create a Hibernate mapping document. Java or SQL databases do not use the types stated and used in the mapping files. They are known as Hibernate mapping types and can convert Java data types to SQL data types and vice versa.

Primary Types of Hibernate Mapping

There are mainly three different kinds of mapping. As follows:

1. One to one: One property is mapped to another attribute in this form of relationship in a way that only one-to-one mapping is maintained. An illustration will help you comprehend this better. Suppose, for instance, that only one employee works for one department.

When a worker cannot work for another department at the same time, the mapping is known as one-to-one.

2. One to many: One property is mapped to many other characteristics in this form of relationship by being mapped to another attribute in a specific way. An illustration will help you comprehend this better. as in the case of a student who belongs to multiple groups, like a simultaneous cultural organization, sports team, and robotics team.

The student and group relationship in this scenario is referred to be a many-to-one relationship.

3. Many to many: One attribute is mapped to another attribute in this type of connection so that any number of attributes can be linked to other attributes without a limit on the quantity. An example will help you better comprehend this. For instance, in the library, one individual may check out a number of books, and one book may be issued to a number of other books.

Many to many partnerships are what this form of relationship is known as. Before implementation, there must be a thorough knowledge of the business use case for this complex relationship.

Types of Mapping

The primary fundamental forms of Hibernate mapping types are:

1. Primitive Types: Data types such as "integer," "character," "float," "string," "double," "Boolean," "short," "long," etc., are defined for this type of mapping. These can be found in the hibernate framework to map java data types to RDBMS data types.

2. Binary and Large Object Types: They include "clob," "blob," "binary," "text," etc. To maintain the data type mapping of big things like images and videos, blob and clob data types are present.

[3. Date and Time Types](#): "Date," "time," "calendar," "timestamp," etc. are some examples. We have data type mappings for dates and times that are similar to primitive.

[4. JDK-related Types](#): This category includes some mappings for things outside the scope of the previous kind of mappings. "Class," "locale," "currency," and "timezone" are among them.

Example: 1

Primitive Types

Some of the primitive forms of Hibernate mapping types:

Mapping type	Java type	ANSI SQL Type
integer	int or java.lang.Integer	INTEGER
long	long or java.lang.Long	BIGINT
short	short or java.lang.Short	SMALLINT
float	float or java.lang.Float	FLOAT
double	double or java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
character	java.lang.String	CHAR(1)
string	java.lang.String	VARCHAR
byte	byte or java.lang.Byte	TINYINT
boolean	boolean or java.lang.Boolean	BIT
yes/no	boolean or java.lang.Boolean	CHAR(1) ('Y' or 'N')
true/false	boolean or java.lang.Boolean	CHAR(1) ('T' or 'F')

Example:2

Binary and Large Object Types

Binary and Large object types are one of the hibernate mapping types.

Mapping type	Java type	ANSI SQL Type
binary	byte[]	VARBINARY (or BLOB)
text	java.lang.String	CLOB
serializable	any Java class that implements java.io.Serializable	VARBINARY (or BLOB)
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB

Example -3

Date and Time Types: Hibernate mapping types also contain the Date and Time types.

Mapping type	Java type	ANSI SQL Type
date	java.util.Date or java.sql.Date	DATE
time	java.util.Date or java.sql.Time	TIME
timestamp	java.util.Date or java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE

Example - 4

JDK-related Types : The fourth category of Hibernate mapping types is JDK-related types.

Mapping type	Java type	ANSI SQL Type
class	java.lang.Class	VARCHAR
locale	java.util.Locale	VARCHAR
timezone	java.util.TimeZone	VARCHAR
currency	java.util.Currency	VARCHAR

Manipulating and Querying

Hibernate Query Language (HQL) is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties. HQL stands for Hibernate Query Language. HQL syntax is quite similar to SQL syntax but it performs operations on objects and properties of persistent classes instead of tables and columns. Hibernate framework translate HQL queries into database specific queries to perform action, Query interface provides the methods and functionality to represent and manipulate a HQL query in the object oriented way.

Note: We can directly use SQL statements in hibernate.

//This class represents a persistent class for Student.

Example: Student.java

```
public class Student {
    //data members
    private int studentId;
    private String firstName;
    private String lastName;
    private String className;
    private String rollNo;
    private int age;

    //no-argument constructor
    public Student() {

    }

    //getter and setter methods
    public int getStudentId() {
        return studentId;
    }
    public void setStudentId(int studentId) {
        this.studentId = studentId;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getClassName() {
        return className;
    }
    public void setClassName(String className) {
        this.className = className;
    }
    public String getRollNo() {
        return rollNo;
    }
}
```

```

    }
    public void setRollNo(String rollNo) {
        this.rollNo = rollNo;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}

```

hibernate.cfg.xml

```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>
        <property name="dialect">
            org.hibernate.dialect.OracleDialect
        </property>
        <property name="connection.url">
            jdbc:oracle:thin:@localhost:1521:XE
        </property>
        <property name="connection.username">
            system
        </property>
        <property name="connection.password">
            oracle
        </property>
        <property name="connection.driver_class">
            oracle.jdbc.driver.OracleDriver
        </property>
        <property name="hbm2ddl.auto">
            update
        </property>
        <property name="show_sql">
            true
        </property>

        <mapping resource="student.hbm.xml"/>

    </session-factory>

</hibernate-configuration>

```

student.hbm.xml

```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping SYSTEM
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

```

```

<class name="com.w3spoint.business.Student" table="Student">
  <id name="studentId" type="int" column="Student_Id">
    <generator class="native"></generator>
  </id>

  <property name="firstName" column="First_Name" type="string"/>
  <property name="lastName" column="Last_Name" type="string"/>
  <property name="className" column="Class" type="string"/>
  <property name="rollNo" column="RollNo" type="string"/>
  <property name="age" column="Age" type="int"/>

</class>

</hibernate-mapping>

```

HibernateUtil.java

```

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

/**
 * This is a utility class for getting the hibernate session object.
 * @author w3spoint
 */
public class HibernateUtil {
    private static final SessionFactory sessionFactory =
        buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        SessionFactory sessionFactory = null;
        try {
            //Create the configuration object.
            Configuration configuration = new Configuration();
            //Initialize the configuration object
            //with the configuration file data
            configuration.configure("hibernate.cfg.xml");
            // Get the SessionFactory object from configuration.
            sessionFactory = configuration.buildSessionFactory();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return sessionFactory;
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}

```

HibernateTest.java

```

import org.hibernate.Query;
import org.hibernate.Session;
import com.w3spoint.persistence.HibernateUtil;

/**
 * This class is used for the hibernate operations.
 * @author w3spoint

```

```

*/
public class HibernateTest {
    public static void main(String args[]){
        //Create the student object.
        Student student = new Student();

        //Setting the object properties.
        student.setFirstName("Vivek");
        student.setLastName("Solenki");
        student.setClassName("MCA ");
        student.setRollNo("MCA/07/70");
        student.setAge(27);

        //Get the session object.
        Session session =
            HibernateUtil.getSessionFactory().openSession();

        //Start hibernate transaction.
        session.beginTransaction();

        //Persist the student object.
        session.save(student);

        //Update the student object.
        Query query1 = session.createQuery("update Student" +
            " set className = 'MCA final'" +
            " where rollNo = 'MCA/07/70'");
        query1.executeUpdate();

        //select a student record
        Query query2 = session.
            createQuery("FROM Student where rollNo = 'MCA/07/70'");
        Student stu1 = (Student) query2.uniqueResult();
        System.out.println("First Name: " + stu1.getFirstName());
        System.out.println("Last Name: " + stu1.getLastName());
        System.out.println("Class: " + stu1.getClassName());
        System.out.println("RollNo: " + stu1.getRollNo());
        System.out.println("Age: " + stu1.getAge());

        //select query using named parameters
        Query query3 = session.
            createQuery("FROM Student where rollNo = :rollNo");
        query3.setParameter("rollNo", "MCA/07/70");
        Student stu2 = (Student) query3.uniqueResult();
        System.out.println("First Name: " + stu2.getFirstName());
        System.out.println("Last Name: " + stu2.getLastName());
        System.out.println("Class: " + stu2.getClassName());
        System.out.println("RollNo: " + stu2.getRollNo());
        System.out.println("Age: " + stu2.getAge());

        //select query using positional parameters
        Query query4 = session.
            createQuery("FROM Student where rollNo = ?");
        query4.setString(0, "MCA/07/70");
        Student stu3 = (Student) query4.uniqueResult();
        System.out.println("First Name: " + stu3.getFirstName());
        System.out.println("Last Name: " + stu3.getLastName());
        System.out.println("Class: " + stu3.getClassName());
        System.out.println("RollNo: " + stu3.getRollNo());
        System.out.println("Age: " + stu3.getAge());
    }
}

```



```

//delete a student record
Query query5 = session.createQuery("delete Student" +
                                   " where rollNo = 'MCA/07/70'");
query5.executeUpdate();

//Commit hibernate transaction.
session.getTransaction().commit();

//Close the hibernate session.
session.close();
}
}

```

Output:

Hibernate: select hibernate_sequence.nextval from dual

Hibernate: insert into Student (First_Name, Last_Name, Class, RollNo, Age, Student_Id) values (?, ?, ?, ?, ?, ?)

Hibernate: update Student set Class='MCA final' where RollNo='MCA/07/70'

Hibernate: select student0_.Student_Id as Student1_0_, student0_.First_Name as First2_0_, student0_.Last_Name as Last3_0_, student0_.Class as Class0_, student0_.RollNo as RollNo0_, student0_.Age as Age0_ from Student student0_ where student0_.RollNo='MCA/07/70'

First Name: Vivek

Last Name: Solenki

Class: MCA

RollNo: MCA/07/70

Age: 27

Hibernate: select student0_.Student_Id as Student1_0_, student0_.First_Name as First2_0_, student0_.Last_Name as Last3_0_, student0_.Class as Class0_, student0_.RollNo as RollNo0_, student0_.Age as Age0_ from Student student0_ where student0_.RollNo=?

First Name: Vivek

Last Name: Solenki

Class: MCA

RollNo: MCA/07/70

Age: 27

Hibernate: select student0_.Student_Id
as Student1_0_, student0_.First_Name as First2_0_,
student0_.Last_Name as Last3_0_, student0_.Class
as Class0_, student0_.RollNo as RollNo0_, student0_.Age
as Age0_ from Student student0_ where student0_.RollNo=?

First Name: Vivek

Last Name: Solenki

Class: MCA

RollNo: MCA/07/70

Age: 27

Hibernate: delete from Student where RollNo='MCA/07/70'

Hibernate Query Language:

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:

1. **`public int executeUpdate()`** is used to execute the update or delete query.
2. **`public List list()`** returns the result of the relation 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

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

Example of HQL to get records with pagination

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

Example of HQL update query

```
Transaction tx=session.beginTransaction();
Query q=session.createQuery("update User set name=:n where id=:i");
q.setParameter("n","Udit Kumar");
q.setParameter("i",111);

int status=q.executeUpdate();
```

```
System.out.println(status);  
tx.commit();
```

Example of HQL delete query

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

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

Example to get maximum salary of employee

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

Example to get minimum salary of employee

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

Example to count total number of employee ID

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

Example to get average salary of each employees

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

Hibernate Criteria Query Language (HCQL)

HCQL: HCQL stands for Hibernate Criteria Query Language. As we discussed HQL provides a way of manipulating data using objects instead of database tables. Hibernate also provides more object oriented alternative ways of HQL. Hibernate Criteria API provides one of these alternatives. HCQL is mainly used in search operations and works on filtration rules and logical conditions.

The Criteria interface, Restrictions class and Order class provides the methods and functionality to perform HCQL operations.

The Criteria interface object can be created by createCriteria() method of Session interface.

Syntax:

Criteria cr = session.createCriteria(persistentClassName.class);

Example:

Student.java

```
/**
 * This class represents a persistent class for Student.
 * @author w3spoint
 */
public class Student {
    //data members
    private int studentId;
    private String firstName;
    private String lastName;
    private String className;
    private String rollNo;
    private int age;

    //no-argument constructor
    public Student() {

    }

    //getter and setter methods
    public int getStudentId() {
        return studentId;
    }
    public void setStudentId(int studentId) {
        this.studentId = studentId;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
}
```

```

public String getLastName() {
    return lastName;
}
public void setLastName(String lastName) {
    this.lastName = lastName;
}
public String getClassName() {
    return className;
}
public void setClassName(String className) {
    this.className = className;
}
public String getRollNo() {
    return rollNo;
}
public void setRollNo(String rollNo) {
    this.rollNo = rollNo;
}
public int getAge() {
    return age;
}
public void setAge(int age) {
    this.age = age;
}
}

```

hibernate.cfg.xml

```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>
        <property name="dialect">
            org.hibernate.dialect.OracleDialect
        </property>
        <property name="connection.url">
            jdbc:oracle:thin:@localhost:1521:XE
        </property>
        <property name="connection.username">
            system
        </property>
        <property name="connection.password">
            oracle
        </property>
        <property name="connection.driver_class">
            oracle.jdbc.driver.OracleDriver
        </property>
        <property name="hbm2ddl.auto">
            update
        </property>
        <property name="show_sql">
            true
        </property>

        <mapping resource="student.hbm.xml"/>
    </session-factory>

```

```
</hibernate-configuration>
```

student.hbm.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping SYSTEM
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

    <class name="com.w3spoint.business.Student" table="Student">
        <id name="studentId" type="int" column="Student_Id">
            <generator class="native"></generator>
        </id>

        <property name="firstName" column="First_Name" type="string"/>
        <property name="lastName" column="Last_Name" type="string"/>
        <property name="className" column="Class" type="string"/>
        <property name="rollNo" column="RollNo" type="string"/>
        <property name="age" column="Age" type="int"/>

    </class>

</hibernate-mapping>
```

HibernateUtil.java

```
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

/**
 * This is a utility class for getting the hibernate session object.
 * @author w3spoint
 */
public class HibernateUtil {
    private static final SessionFactory sessionFactory =
        buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        SessionFactory sessionFactory = null;
        try {
            //Create the configuration object.
            Configuration configuration = new Configuration();
            //Initialize the configuration object
            //with the configuration file data
            configuration.configure("hibernate.cfg.xml");
            // Get the SessionFactory object from configuration.
            sessionFactory = configuration.buildSessionFactory();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return sessionFactory;
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

```

    }

}

HibernateTest.java

import java.util.List;
import org.hibernate.Criteria;
import org.hibernate.Session;
import com.w3spoint.persistence.HibernateUtil;

/**
 * This class is used for the hibernate operations.
 * @author w3spoint
 */
public class HibernateTest {
    public static void main(String args[]){
        //Create the student object.
        Student student = new Student();

        //Setting the object properties.
        student.setFirstName("Sunil");
        student.setLastName("Kunar");
        student.setClassName("MCA final");
        student.setRollNo("MCA/07/15");
        student.setAge(27);

        //Get the session object.
        Session session =
            HibernateUtil.getSessionFactory().openSession();

        //Start hibernate transaction.
        session.beginTransaction();

        //Persist the student object.
        session.save(student);

        //Commit hibernate transaction.
        session.getTransaction().commit();

        //select a student record using Criteria
        Criteria criteria = session.createCriteria(Student.class);
        List<Student> stuList = (List<Student>) criteria.list();
        for(Student stu : stuList){
            System.out.println("First Name: "
                               + stu.getFirstName());
            System.out.println("Last Name: "
                               + stu.getLastName());
            System.out.println("Class: "
                               + stu.getClassName());
            System.out.println("RollNo: "
                               + stu.getRollNo());
            System.out.println("Age: " + stu.getAge());
        }

        //Close the hibernate session.
        session.close();
    }
}

```


Output:

Hibernate: select hibernate_sequence.nextval from dual

Hibernate: insert into Student (First_Name, Last_Name, Class,
RollNo, Age, Student_Id) values (?, ?, ?, ?, ?, ?)

Hibernate: select this_.Student_Id as Student1_0_0_,
this_.First_Name as First2_0_0_, this_.Last_Name as
Last3_0_0_, this_.Class as Class0_0_, this_.RollNo as
RollNo0_0_, this_.Age as Age0_0_ from Student this_

First Name: Sunil

Last Name: Kunar

Class: MCA final

RollNo: MCA/07/15

Age: 27

Example: Different methods Criteria Query

- [Hibernate criteria restrictions query example.](#)
- [Hibernate criteria ordering query example.](#)
- [Hibernate criteria pagination query example.](#)
- [Hibernate criteria projections query example.](#)

Hibernate criteria restrictions query example.

Restrictions Query Hibernate

Restrictions class provides the methods to restrict the search result based on the restriction provided.

Commonly used Restriction class methods:

1. Restrictions.eq: Make a restriction that the property value must be equal to the specified value.

Syntax:

```
Restrictions.eq("property", specifiedValue)
```

2. Restrictions.lt: Make a restriction that the property value must be less than the specified value.

Syntax:

```
Restrictions.lt("property", specifiedValue)
```

3. Restrictions.le: Make a restriction that the property value must be less than or equal to the specified value.

Syntax:

```
Restrictions.le("property", specifiedValue)
```

4. Restrictions.gt: Make a restriction that the property value must be greater than the specified value.

Syntax:

```
Restrictions.gt("property", specifiedValue)
```

5. Restrictions.ge: Make a restriction that the property value must be greater than or equal to the specified value.

Syntax:

```
Restrictions.ge("property", specifiedValue)
```

6. Restrictions.like: Make a restriction that the property value follows the specified like pattern.

Syntax:

```
Restrictions.like("property", "likePattern")
```

7. Restrictions.between: Make a restriction that the property value must be between the start and end limit values.

Syntax:

```
Restrictions.between("property", startValue, endValue)
```

8. Restrictions.isNull: Make a restriction that the property value must be null.

Syntax:

Restrictions.isNull("property")

9. Restrictions.isNotNull: Make a restriction that the property value must not be null.

Syntax:

Restrictions.isNotNull("property")

Example: Restrictions query

Student.java

```
/**
 * This class represents a persistent class for Student.
 * @author w3spoint
 */
public class Student {
    //data members
    private int studentId;
    private String firstName;
    private String lastName;
    private String className;
    private String rollNo;
    private int age;

    //no-argument constructor
    public Student() {

    }

    //getter and setter methods
    public int getStudentId() {
        return studentId;
    }
    public void setStudentId(int studentId) {
        this.studentId = studentId;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getClassName() {
        return className;
    }
    public void setClassName(String className) {
        this.className = className;
    }
}
```

```

    }
    public String getRollNo() {
        return rollNo;
    }
    public void setRollNo(String rollNo) {
        this.rollNo = rollNo;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}

```

```

}

```

hibernate.cfg.xml

```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>
        <property name="dialect">
            org.hibernate.dialect.OracleDialect
        </property>
        <property name="connection.url">
            jdbc:oracle:thin:@localhost:1521:XE
        </property>
        <property name="connection.username">
            system
        </property>
        <property name="connection.password">
            oracle
        </property>
        <property name="connection.driver_class">
            oracle.jdbc.driver.OracleDriver
        </property>
        <property name="hbm2ddl.auto">
            update
        </property>
        <property name="show_sql">
            true
        </property>

        <mapping resource="student.hbm.xml"/>

    </session-factory>

</hibernate-configuration>

```

student.hbm.xml

```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping SYSTEM
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

```

```

<hibernate-mapping>

<class name="com.w3spoint.business.Student" table="Student">
  <id name="studentId" type="int" column="Student_Id">
    <generator class="native"></generator>
  </id>

  <property name="firstName" column="First_Name" type="string"/>
  <property name="lastName" column="Last_Name" type="string"/>
  <property name="className" column="Class" type="string"/>
  <property name="rollNo" column="RollNo" type="string"/>
  <property name="age" column="Age" type="int"/>

</class>

</hibernate-mapping>

```

HibernateUtil.java

```

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

/**
 * This is a utility class for getting the hibernate session object.
 * @author w3spoint
 */
public class HibernateUtil {
    private static final SessionFactory sessionFactory =
        buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        SessionFactory sessionFactory = null;
        try {
            //Create the configuration object.
            Configuration configuration = new Configuration();
            //Initialize the configuration object
            //with the configuration file data
            configuration.configure("hibernate.cfg.xml");
            // Get the SessionFactory object from configuration.
            sessionFactory = configuration.buildSessionFactory();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return sessionFactory;
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}

```

HibernateTest.java

```

import org.hibernate.Criteria;
import org.hibernate.Session;
import org.hibernate.criterion.Restrictions;
import com.w3spoint.persistence.HibernateUtil;

```

```

/**
 * This class is used for the hibernate operations.
 * @author w3spoint
 */
public class HibernateTest {
    public static void main(String args[]){
        //Create the student object.
        Student student = new Student();

        //Setting the object properties.
        student.setFirstName("Ashish");
        student.setLastName("Malik");
        student.setClassName("MCA final");
        student.setRollNo("MCA/07/21");
        student.setAge(27);

        //Get the session object.
        Session session =
            HibernateUtil.getSessionFactory().openSession();

        //Start hibernate transaction.
        session.beginTransaction();

        //Persist the student object.
        session.save(student);

        //Commit hibernate transaction.
        session.getTransaction().commit();

        //select a student record using Criteria restriction query
        Criteria criteria = session.createCriteria(Student.class);
        criteria.add(Restrictions.eq("rollNo", "MCA/07/21"));
        Student stu = (Student) criteria.uniqueResult();
        System.out.println("First Name: " + stu.getFirstName());
        System.out.println("Last Name: " + stu.getLastName());
        System.out.println("Class: " + stu.getClassName());
        System.out.println("RollNo: " + stu.getRollNo());
        System.out.println("Age: " + stu.getAge());

        //Close hibernate session.
        session.close();
    }
}

```

Output:

Hibernate: select hibernate_sequence.nextval from dual

Hibernate: insert into Student (First_Name, Last_Name, Class,
RollNo, Age, Student_Id) values (?, ?, ?, ?, ?, ?)

Hibernate: select this_.Student_Id as Student1_0_0_,
this_.First_Name as First2_0_0_, this_.Last_Name as

Last3_0_0_, this_.Class as Class0_0_, this_.RollNo as

RollNo0_0_, this_.Age as Age0_0_ from

Student this_ where this_.RollNo=?

First Name: Ashish

Last Name: Malik

Class: MCA final

RollNo: MCA/07/21

Age: 27

Hibernate criteria ordering query example.

Order By Hibernate, Order class provides the methods for performing ordering operations.

Methods of Order class:

1. Order.asc: To sort the records in ascending order based on the specified property.

Syntax:

Order.asc("property")

2. Order.desc: To sort the records in descending order based on the specified property.

Syntax:

Order.desc("property")

Example: Order By Hibernate

Student.java

```
/**
 * This class represents a persistent class for Student.
 * @author w3spoint
 */
public class Student {
    //data members
    private int studentId;
```

```

private String firstName;
private String lastName;
private String className;
private String rollNo;
private int age;

//no-argument constructor
public Student() {

}

//getter and setter methods
public int getStudentId() {
    return studentId;
}
public void setStudentId(int studentId) {
    this.studentId = studentId;
}
public String getFirstName() {
    return firstName;
}
public void setFirstName(String firstName) {
    this.firstName = firstName;
}
public String getLastName() {
    return lastName;
}
public void setLastName(String lastName) {
    this.lastName = lastName;
}
public String getClassName() {
    return className;
}
public void setClassName(String className) {
    this.className = className;
}
public String getRollNo() {
    return rollNo;
}
public void setRollNo(String rollNo) {
    this.rollNo = rollNo;
}
public int getAge() {
    return age;
}
public void setAge(int age) {
    this.age = age;
}

}

```

hibernate.cfg.xml

```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>
        <property name="dialect">
            org.hibernate.dialect.OracleDialect

```



```

        </property>
        <property name="connection.url">
            jdbc:oracle:thin:@localhost:1521:XE
        </property>
        <property name="connection.username">
            system
        </property>
        <property name="connection.password">
            oracle
        </property>
        <property name="connection.driver_class">
            oracle.jdbc.driver.OracleDriver
        </property>
        <property name="hbm2ddl.auto">
            update
        </property>
        <property name="show_sql">
            true
        </property>

        <mapping resource="student.hbm.xml"/>

    </session-factory>
</hibernate-configuration>
student.hbm.xml

```

```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping SYSTEM
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

    <class name="com.w3spoint.business.Student" table="Student">
        <id name="studentId" type="int" column="Student_Id">
            <generator class="native"></generator>
        </id>

        <property name="firstName" column="First_Name" type="string"/>
        <property name="lastName" column="Last_Name" type="string"/>
        <property name="className" column="Class" type="string"/>
        <property name="rollNo" column="RollNo" type="string"/>
        <property name="age" column="Age" type="int"/>

    </class>

</hibernate-mapping>

```

HibernateUtil.java

```

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

/**
 * This is a utility class for getting the hibernate session object.
 * @author w3spoint
 */
public class HibernateUtil {
    private static final SessionFactory sessionFactory =
        buildSessionFactory();

```

```

private static SessionFactory buildSessionFactory() {
    SessionFactory sessionFactory = null;
    try {
        //Create the configuration object.
        Configuration configuration = new Configuration();
        //Initialize the configuration object
        //with the configuration file data
        configuration.configure("hibernate.cfg.xml");
        // Get the SessionFactory object from configuration.
        sessionFactory = configuration.buildSessionFactory();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    return sessionFactory;
}

public static SessionFactory getSessionFactory() {
    return sessionFactory;
}
}

```

HibernateTest.java

```

import java.util.List;
import org.hibernate.Criteria;
import org.hibernate.Session;
import org.hibernate.criterion.Order;
import com.w3spoint.persistence.HibernateUtil;

/**
 * This class is used for the hibernate operations.
 * @author w3spoint
 */
public class HibernateTest {
    public static void main(String args[]){
        //Create the student object.
        Student student1 = new Student();
        Student student2 = new Student();
        Student student3 = new Student();

        //Setting the object properties.
        student1.setFirstName("Sunil");
        student1.setLastName("Kunar");
        student1.setClassName("MCA final");
        student1.setRollNo("MCA/07/15");
        student1.setAge(27);

        student2.setFirstName("Pardeep");
        student2.setLastName("Janra");
        student2.setClassName("MCA final");
        student2.setRollNo("MCA/07/35");
        student2.setAge(27);

        student3.setFirstName("Sandy");
        student3.setLastName("Sethi");
        student3.setClassName("MCA final");
        student3.setRollNo("MCA/07/19");
        student3.setAge(27);

        //Get the session object.
    }
}

```

```

Session session =
    HibernateUtil.getSessionFactory().openSession();

//Start hibernate transaction.
session.beginTransaction();

//Persist the student object.
session.save(student1);
session.save(student2);
session.save(student3);

//Commit hibernate transaction.
session.getTransaction().commit();

//select a student record using Criteria ordering query
Criteria criteria = session.createCriteria(Student.class);
criteria.addOrder(Order.asc("firstName"));
List<Student> stuList = (List<Student>) criteria.list();
for(Student stu : stuList){
    System.out.println("First Name: " + stu.getFirstName());
    System.out.println("Last Name: " + stu.getLastName());
    System.out.println("Class: " + stu.getClassName());
    System.out.println("RollNo: " + stu.getRollNo());
    System.out.println("Age: " + stu.getAge());
}

//Close hibernate session.
session.close();
}
}

```

Output:

```

Hibernate: select hibernate_sequence.nextval from dual
Hibernate: select hibernate_sequence.nextval from dual
Hibernate: select hibernate_sequence.nextval from dual
Hibernate: insert into Student (First_Name, Last_Name, Class,
RollNo, Age, Student_Id) values (?, ?, ?, ?, ?, ?)
Hibernate: insert into Student (First_Name, Last_Name, Class,
RollNo, Age, Student_Id) values (?, ?, ?, ?, ?, ?)
Hibernate: insert into Student (First_Name, Last_Name, Class,
RollNo, Age, Student_Id) values (?, ?, ?, ?, ?, ?)
Hibernate: select this_.Student_Id as Student1_0_0_,
this_.First_Name as First2_0_0_, this_.Last_Name as Last3_0_0_,
this_.Class as Class0_0_, this_.RollNo as RollNo0_0_, this_.Age
as Age0_0_ from Student this_ order by this_.First_Name asc
First Name: Pardeep
Last Name: Janra
Class: MCA final
RollNo: MCA/07/35
Age: 27
First Name: Sandy
Last Name: Sethi
Class: MCA final
RollNo: MCA/07/19
Age: 27
First Name: Sunil
Last Name: Kunar
Class: MCA final
RollNo: MCA/07/15

```

Age: 27

Hibernate criteria pagination query example.

Pagination In Hibernate

Criteria interface provides the methods for performing pagination operations.

Methods of Criteria interface used for pagination query:

1. setFirstResult(int firstResult): Set the first row of your result based on the firstResult value. Row index starts from 0.

Syntax:

```
public Criteria setFirstResult(int firstResult)
```

2. setMaxResults(int maxResults): Set the maximum number of records in your result.

Syntax:

```
public Criteria setMaxResults(int maxResults)
```

Example:Pagination in hibernate

Student.java

```
/**
 * This class represents a persistent class for Student.
 * @author w3spoint
 */
public class Student {
    //data members
    private int studentId;
    private String firstName;
    private String lastName;
    private String className;
    private String rollNo;
    private int age;

    //no-argument constructor
    public Student() {

    }

    //getter and setter methods
    public int getStudentId() {
        return studentId;
    }
    public void setStudentId(int studentId) {
        this.studentId = studentId;
    }
}
```

```

    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getClassName() {
        return className;
    }
    public void setClassName(String className) {
        this.className = className;
    }
    public String getRollNo() {
        return rollNo;
    }
    public void setRollNo(String rollNo) {
        this.rollNo = rollNo;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}

```

hibernate.cfg.xml

```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>
        <property name="dialect">
            org.hibernate.dialect.OracleDialect
        </property>
        <property name="connection.url">
            jdbc:oracle:thin:@localhost:1521:XE
        </property>
        <property name="connection.username">
            system
        </property>
        <property name="connection.password">
            oracle
        </property>
        <property name="connection.driver_class">
            oracle.jdbc.driver.OracleDriver
        </property>
        <property name="hbm2ddl.auto">

```

```

        update
    </property>
    <property name="show_sql">
        true
    </property>

    <mapping resource="student.hbm.xml"/>

</session-factory>

</hibernate-configuration>

```

student.hbm.xml

```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping SYSTEM
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

    <class name="com.w3spoint.business.Student" table="Student">
        <id name="studentId" type="int" column="Student_Id">
            <generator class="native"></generator>
        </id>

        <property name="firstName" column="First_Name" type="string"/>
        <property name="lastName" column="Last_Name" type="string"/>
        <property name="className" column="Class" type="string"/>
        <property name="rollNo" column="RollNo" type="string"/>
        <property name="age" column="Age" type="int"/>

    </class>

</hibernate-mapping>

```

HibernateUtil.java

```

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

/**
 * This is a utility class for getting the hibernate session object.
 * @author w3spoint
 */
public class HibernateUtil {
    private static final SessionFactory sessionFactory =
        buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        SessionFactory sessionFactory = null;
        try {
            //Create the configuration object.
            Configuration configuration = new Configuration();
            //Initialize the configuration object
            //with the configuration file data
            configuration.configure("hibernate.cfg.xml");
            // Get the SessionFactory object from configuration.
            sessionFactory = configuration.buildSessionFactory();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

```

        return sessionFactory;
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}

```

HibernateTest.java

```

import java.util.List;
import org.hibernate.Criteria;
import org.hibernate.Session;
import com.w3spoint.persistence.HibernateUtil;

/**
 * This class is used for the hibernate operations.
 * @author w3spoint
 */
public class HibernateTest {
    public static void main(String args[]){
        //Create the student object.
        Student student1 = new Student();
        Student student2 = new Student();
        Student student3 = new Student();

        //Setting the object properties.
        student1.setFirstName("Bharat");
        student1.setLastName("Jaiswal");
        student1.setClassName("MCA final");
        student1.setRollNo("MCA/07/15");
        student1.setAge(27);

        student2.setFirstName("Richi");
        student2.setLastName("Gora");
        student2.setClassName("MCA final");
        student2.setRollNo("MCA/07/35");
        student2.setAge(27);

        student3.setFirstName("Rajesh");
        student3.setLastName("Garg");
        student3.setClassName("MCA final");
        student3.setRollNo("MCA/07/19");
        student3.setAge(27);

        //Get the session object.
        Session session =
            HibernateUtil.getSessionFactory().openSession();

        //Start hibernate transaction.
        session.beginTransaction();

        //Persist the student object.
        session.save(student1);
        session.save(student2);
        session.save(student3);

        //Commit hibernate transaction.
    }
}

```

```

        session.getTransaction().commit();

        //select a student record using Criteria pagination query
        Criteria criteria = session.createCriteria(Student.class);
        criteria.setFirstResult(1);
        criteria.setMaxResults(2);
        List<Student> stuList = (List<Student>) criteria.list();
        for(Student stu : stuList){
            System.out.println("First Name: " + stu.getFirstName());
            System.out.println("Last Name: " + stu.getLastName());
            System.out.println("Class: " + stu.getClassName());
            System.out.println("RollNo: " + stu.getRollNo());
            System.out.println("Age: " + stu.getAge());
        }

        //Close hibernate session.
        session.close();
    }
}

```

Output:

```

Hibernate: select hibernate_sequence.nextval from dual
Hibernate: select hibernate_sequence.nextval from dual
Hibernate: select hibernate_sequence.nextval from dual
Hibernate: insert into Student (First_Name, Last_Name, Class,
RollNo, Age, Student_Id) values (?, ?, ?, ?, ?, ?)
Hibernate: insert into Student (First_Name, Last_Name, Class,
RollNo, Age, Student_Id) values (?, ?, ?, ?, ?, ?)
Hibernate: insert into Student (First_Name, Last_Name, Class,
RollNo, Age, Student_Id) values (?, ?, ?, ?, ?, ?)
Hibernate: select * from ( select row_.*, rownum
rownum_ from ( select this_.Student_Id as Student1_0_0_,
this_.First_Name as First2_0_0_, this_.Last_Name as
Last3_0_0_, this_.Class as Class0_0_, this_.RollNo as
RollNo0_0_, this_.Age as Age0_0_ from Student this_ )
row_ ) where rownum_ <= ? and rownum_ > ?
First Name: Pardeep
Last Name: Janra
Class: MCA final
RollNo: MCA/07/35
Age: 27
First Name: Sandy
Last Name: Sethi
Class: MCA final
RollNo: MCA/07/19
Age: 27

```

[Hibernate criteria projections query example.](#)

Projections Hibernate

Projections class provides the methods to perform the operation on a particular column.

Commonly used methods of Projections class:

1. Projections.rowCount: Return the total no. of rows.

Syntax:

`Projections.rowCount()`

2. Projections.avg: Return the average of a specified property.

Syntax:

`Projections.avg("property")`

3. Projections.countDistinct: Return the distinct count of a property.

Syntax:

`Projections.countDistinct("property")`

4. Projections.max: Return the maximum value of a property.

Syntax:

`Projections.max("property")`

5. Projections.min: Return the minimum value of a property.

Syntax:

`Projections.min("property")`

6. Projections.sum: Return the sum of a property.

Syntax:

`Projections.sum("property")`

Example: Projections Hibernate

Student.java

```
/**
 * This class represents a persistent class for Student.
 * @author w3spoint
 */
public class Student {
    //data members
    private int studentId;
    private String firstName;
    private String lastName;
    private String className;
    private String rollNo;
    private int age;

    //no-argument constructor
    public Student() {

    }

    //getter and setter methods
```

```

    public int getStudentId() {
        return studentId;
    }
    public void setStudentId(int studentId) {
        this.studentId = studentId;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getClassName() {
        return className;
    }
    public void setClassName(String className) {
        this.className = className;
    }
    public String getRollNo() {
        return rollNo;
    }
    public void setRollNo(String rollNo) {
        this.rollNo = rollNo;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}

```

hibernate.cfg.xml

```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>
        <property name="dialect">
            org.hibernate.dialect.OracleDialect
        </property>
        <property name="connection.url">
            jdbc:oracle:thin:@localhost:1521:XE
        </property>
        <property name="connection.username">
            system
        </property>
        <property name="connection.password">
            oracle

```

```

        </property>
        <property name="connection.driver_class">
            oracle.jdbc.driver.OracleDriver
        </property>
        <property name="hbm2ddl.auto">
            update
        </property>
        <property name="show_sql">
            true
        </property>

        <mapping resource="student.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
student.hbm.xml

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping SYSTEM
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

    <class name="com.w3spoint.business.Student" table="Student">
        <id name="studentId" type="int" column="Student_Id">
            <generator class="native"></generator>
        </id>

        <property name="firstName" column="First_Name" type="string"/>
        <property name="lastName" column="Last_Name" type="string"/>
        <property name="className" column="Class" type="string"/>
        <property name="rollNo" column="RollNo" type="string"/>
        <property name="age" column="Age" type="int"/>

    </class>

</hibernate-mapping>

```

HibernateUtil.java

```

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

/**
 * This is a utility class for getting the hibernate session object.
 * @author w3spoint
 */
public class HibernateUtil {
    private static final SessionFactory sessionFactory =
        buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        SessionFactory sessionFactory = null;
        try {
            //Create the configuration object.
            Configuration configuration = new Configuration();
            //Initialize the configuration object
            //with the configuration file data

```

```

        configuration.configure("hibernate.cfg.xml");
        // Get the SessionFactory object from configuration.
        sessionFactory = configuration.buildSessionFactory();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    return sessionFactory;
}

public static SessionFactory getSessionFactory() {
    return sessionFactory;
}
}

```

HibernateTest.java

```

import org.hibernate.Criteria;
import org.hibernate.Session;
import org.hibernate.criterion.Projections;

import com.w3spoint.persistence.HibernateUtil;

/**
 * This class is used for the hibernate operations.
 * @author w3spoint
 */
public class HibernateTest {
    public static void main(String args[]){
        //Create the student object.
        Student student1 = new Student();
        Student student2 = new Student();
        Student student3 = new Student();

        //Setting the object properties.
        student1.setFirstName("Roxy");
        student1.setLastName("Malik");
        student1.setClassName("MCA final");
        student1.setRollNo("MCA/07/32");
        student1.setAge(28);

        student2.setFirstName("Neeraj");
        student2.setLastName("Chechi");
        student2.setClassName("MCA final");
        student2.setRollNo("MCA/07/33");
        student2.setAge(29);

        student3.setFirstName("Sahdev");
        student3.setLastName("Gorila");
        student3.setClassName("MCA final");
        student3.setRollNo("MCA/07/19");
        student3.setAge(27);

        //Get the session object.
        Session session =
            HibernateUtil.getSessionFactory().openSession();

        //Start hibernate transaction.
        session.beginTransaction();

        //Persist the student object.
    }
}

```

```

        session.save(student1);
        session.save(student2);
        session.save(student3);

        //Commit hibernate transaction.
        session.getTransaction().commit();

        //select a student record using Criteria pagination query
        Criteria criteria = session.createCriteria(Student.class);

        // give total row count.
        criteria.setProjection(Projections.rowCount());
        Integer count = (Integer) criteria.uniqueResult();
        System.out.println("No. of records: " + count);

        // give maximum age.
        criteria.setProjection(Projections.max("age"));
        Integer maxAge = (Integer) criteria.uniqueResult();
        System.out.println("Max age: " + maxAge);

        //Close hibernate session.
        session.close();
    }
}

```

Output:

```

Hibernate: select hibernate_sequence.nextval from dual
Hibernate: select hibernate_sequence.nextval from dual
Hibernate: select hibernate_sequence.nextval from dual
Hibernate: insert into Student (First_Name, Last_Name, Class,
RollNo, Age, Student_Id) values (?, ?, ?, ?, ?, ?)
Hibernate: insert into Student (First_Name, Last_Name, Class,
RollNo, Age, Student_Id) values (?, ?, ?, ?, ?, ?)
Hibernate: insert into Student (First_Name, Last_Name, Class,
RollNo, Age, Student_Id) values (?, ?, ?, ?, ?, ?)
Hibernate: select count(*) as y0_ from Student this_
No. of records: 3
Hibernate: select max(this_.Age) as y0_ from Student this_
Max age: 29

```

Native SQL

Hibernate is a popular object-relational mapping (ORM) tool used in Java applications. It allows developers to map Java objects to database tables and perform CRUD (create, read, update, delete) operations on the database without writing SQL queries manually. Native SQL queries are useful when you need to perform complex queries that cannot be expressed using Hibernate's Query Language (HQL) or Criteria API. Native SQL queries can be used to perform complex joins, aggregate functions, and subqueries. To execute a native SQL query in Hibernate, create an SQLQuery object and set the SQL statement to be executed.

Advantages of using native SQL queries in Hibernate

While Hibernate provides a powerful and easy-to-use ORM framework for accessing relational databases, there are some scenarios where native SQL queries can offer advantages over using the Hibernate Query Language (HQL) or Criteria API. Here are some advantages of using native SQL queries in Hibernate:

1. **Performance:** Native SQL queries can be optimized for performance and may outperform HQL queries or Criteria queries in some cases. This is because native SQL queries are executed directly by the database, bypassing the Hibernate framework and any overhead that it may introduce.
2. **Complex queries:** Native SQL queries can handle complex queries that are difficult or impossible to express using HQL or the Criteria API. For example, queries involving complex joins or subqueries can be expressed more easily and efficiently using native SQL.
3. **Integration with legacy systems:** In some cases, legacy systems may require the use of native SQL queries to access the database. By providing support for native SQL queries, Hibernate can integrate more easily with legacy systems and allow them to be modernized more gradually.
4. **Access to database-specific features:** Native SQL queries allow access to database-specific features that may not be available through Hibernate or JPA. This can include advanced features such as stored procedures, triggers, or custom data types.
5. **Familiarity with SQL:** Many developers are familiar with SQL and may prefer to use native SQL queries to express their queries in a more familiar syntax. This can make the code easier to read and maintain, especially for developers who are new to Hibernate or JPA.

JPA Native Query vs Hibernate Native Query

- Both JPA and Hibernate provide support for executing native SQL queries. However, there are some differences between JPA native queries and Hibernate native queries.
- JPA (Java Persistence API) is a specification for ORM (Object-Relational Mapping) frameworks in Java, while Hibernate is an implementation of the JPA specification. Both JPA and Hibernate provide support for executing native SQL queries, but there are some differences between the two.

S.NO	JPA Native Query	Hibernate Native Query
1	The JPA specification defines a set of interfaces for executing native SQL queries, including the EntityManager interface and the Query interface.	Hibernate provides its own implementation of native SQL queries, which includes additional features not found in the JPA specification.
2	Native queries in JPA are created using the createNativeQuery() method of the EntityManager interface, which returns an instance of the Query interface.	Hibernate Native Query is created using the createSQLQuery() method of the Session interface, which returns an instance of the SQLQuery interface.
3	JPA Native Query allows developers to map the result set of the query to entities using the addEntity() method or to non-entity classes using the setResultTransformer() method.	Hibernate Native Query provides support for additional mapping options, including the ability to map the result set of a query to a non-entity class using the addScalar() method.
4	JPA Native Query does not provide support for Hibernate-specific features, such as the ability to map the result set of a query to a non-entity class using the addScalar() method.	Hibernate Native Query also provides additional methods for controlling the caching behavior of the query, including setCacheable(), setCacheRegion(), and setCacheMode().

Writing and Executing native SQL queries in Hibernate

Example:

1. Create a Hibernate Session

The first step is to obtain a Hibernate Session object. This can be done using the SessionFactory, as shown in the following code snippet:

```
SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
Session session = sessionFactory.openSession();
```

2. Create a Native Query

Once you have obtained a Hibernate Session, you can create a native SQL query using the createSQLQuery() method, as shown in the following code snippet:

```
String sqlQuery = "SELECT * FROM customers WHERE last_name = :lastName";
SQLQuery query = session.createSQLQuery(sqlQuery);
```

3. Set Parameters

If your query includes parameters, you can set their values using the `setParameter()` method, as shown in the following code snippet:

```
query.setParameter("lastName", "Smith");
```

4. Execute the Query

Once you have created and configured your native SQL query, you can execute it using the `list()` or `uniqueResult()` methods, as shown in the following code snippet:

```
List<Customer> customers = query.list();
```

5. Close the Session

Finally, once you have finished using the Hibernate Session and the results of your query, you should close the Session to release any resources it is holding, as shown in the following code snippet:

```
session.close();
```

Debugging Native SQL Queries in Hibernate

Debugging native SQL queries in Hibernate can be done in several ways. Here are some tips to help you debug native SQL queries in Hibernate:

1. Print the generated SQL query

Hibernate generates the SQL query based on the HQL or native SQL query that you write. You can print the generated SQL query to the console to see how Hibernate has transformed your query. You can do this by enabling the Hibernate `show_sql` property in your configuration file, as shown below:

```
<property name="hibernate.show_sql">true</property>
```

2. Use logging to track SQL queries

Hibernate provides a logging facility that you can use to track SQL queries. You can enable logging for SQL queries by setting the Hibernate logging level to debug or trace, as shown below:

```
<logger name="org.hibernate.SQL" level="debug"/>
```

3. Check for syntax errors

Native SQL queries are prone to syntax errors, just like any other SQL query. If you encounter errors when executing your native SQL query, make sure to check for syntax errors in the query.

4. Check for mapping errors

If you are mapping the results of a native SQL query to entities or non-entity classes, make sure that the mapping is correct. Any errors in the mapping can cause the query to fail or return unexpected results.

5. Use a debugger

If you are still having trouble debugging your native SQL query, you can use a debugger to step through the code and see where the query is failing. Set breakpoints in your code and step through it to see where the query is failing or returning unexpected results.

Example: Native SQL

Native SQL to express database queries if you want to utilize database-specific features such as query hints or the CONNECT keyword in Oracle. Hibernate 3.x allows you to specify handwritten SQL, including stored procedures, for all create, update, delete, and load operations and Your application will create a native SQL query from the session with the `createSQLQuery()` method on the Session interface.

```
public SQLQuery createSQLQuery(String sqlString) throws
HibernateException
```

After you pass a string containing the SQL query to the `createSQLQuery()` method, you can associate the SQL result with either an existing Hibernate entity, a join, or a scalar result using `addEntity()`, `addJoin()`, and `addScalar()` methods respectively.

Scalar Queries

The most basic SQL query is to get a list of scalars (values) from one or more tables. Following is the syntax for using native SQL for scalar values –

```
String sql = "SELECT first_name, salary FROM EMPLOYEE";
SQLQuery query = session.createSQLQuery(sql);
query.setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);
List results = query.list();
```

Entity Queries

The above queries were all about returning scalar values, basically returning the "raw" values from the result set. Following is the syntax to get entity objects as a whole from a native sql query via addEntity().

```
String sql = "SELECT * FROM EMPLOYEE";
SQLQuery query = session.createSQLQuery(sql);
query.addEntity(Employee.class);
List results = query.list();
```

Named SQL Queries

Following is the syntax to get entity objects from a native sql query via addEntity() and using named SQL query.

```
String sql = "SELECT * FROM EMPLOYEE WHERE id = :employee_id";
SQLQuery query = session.createSQLQuery(sql);
query.addEntity(Employee.class);
query.setParameter("employee_id", 10);
List results = query.list();
```

Native SQL Example

Consider the following POJO class –

```
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;
    }
}

```

Let us create the following EMPLOYEE table to store Employee objects

```

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)
);

```

Following will be mapping file –

```

<?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">

<hibernate-mapping>
    <class name = "Employee" table = "EMPLOYEE">

        <meta attribute = "class-description">
            This class contains the employee detail.
        </meta>
    </class>
</hibernate-mapping>

```

```

<id name = "id" type = "int" column = "id">
    <generator class="native"/>
</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>

```

Finally, we will create our application class with the main() method to run the application where we will use **Native SQL** queries

```

import java.util.*;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.SessionFactory;
import org.hibernate.SQLQuery;
import org.hibernate.Criteria;
import org.hibernate.Hibernate;
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);
        }

        ManageEmployee ME = new ManageEmployee();

        /* Add few employee records in database */
        Integer empID1 = ME.addEmployee("Zara", "Ali", 2000);
        Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);
        Integer empID3 = ME.addEmployee("John", "Paul", 5000);
        Integer empID4 = ME.addEmployee("Mohd", "Yasee", 3000);

        /* List down employees and their salary using Scalar Query */
        ME.listEmployeesScalar();

        /* List down complete employees information using Entity Query */
        ME.listEmployeesEntity();
    }
}

```

```

/* 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 using Scalar Query */
public void listEmployeesScalar( ){
    Session session = factory.openSession();
    Transaction tx = null;

    try {
        tx = session.beginTransaction();
        String sql = "SELECT first_name, salary FROM EMPLOYEE";
        SQLQuery query = session.createSQLQuery(sql);
        query.setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);
        List data = query.list();

        for(Object object : data) {
            Map row = (Map)object;
            System.out.print("First Name: " + row.get("first_name"));
            System.out.println(", Salary: " + row.get("salary"));
        }
        tx.commit();
    } catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    } finally {
        session.close();
    }
}

```

```

/* Method to READ all the employees using Entity Query */

```

```

public void listEmployeesEntity( ){
    Session session = factory.openSession();
    Transaction tx = null;

    try {
        tx = session.beginTransaction();
        String sql = "SELECT * FROM EMPLOYEE";
        SQLQuery query = session.createSQLQuery(sql);
        query.addEntity(Employee.class);
        List employees = query.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();
    }
}
}

```

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 the following result, and records would be created in the EMPLOYEE table

Output:

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

First Name: Zara, Salary: 2000
First Name: Daisy, Salary: 5000
First Name: John, Salary: 5000
First Name: Mohd, Salary: 3000
First Name: Zara Last Name: Ali Salary: 2000
First Name: Daisy Last Name: Das Salary: 5000
First Name: John Last Name: Paul Salary: 5000
First Name: Mohd Last Name: Yasee Salary: 3000
```

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

```
mysql> select * from EMPLOYEE;
+----+-----+-----+-----+
| id | first_name | last_name | salary |
+----+-----+-----+-----+
| 26 | Zara      | Ali      | 2000   |
| 27 | Daisy     | Das      | 5000   |
| 28 | John      | Paul     | 5000   |
| 29 | Mohd      | Yasee    | 3000   |
+----+-----+-----+-----+
4 rows in set (0.00 sec)
mysql>
```

Transactions and Concurrency in Hibernate

A transaction is a unit of work in which either all operations must execute or none of them. To understand the importance of transaction, think of an example which applies on all of us. “Transferring Amount from one account to another “ – this operation includes below at least two steps

- a. Deduct the balance from sender's account
- b. Add the amount to the receiver's account.

Now think of the situation where amount is deducted from sender's account but not gets delivered to receiver account due to some errors. Such issues are managed by transaction management in which both the steps are performed in a single unit of work where either both steps are performed successfully or in case anyone gets failed, it should be roll backed.

There are four important terms which are very important to understand.

- a. **Atomic** - As described above , atomicity makes sure that either all operations within a transaction must be successful or none of them.
- b. **Consistent**- This property makes sure that data should be in consistent state once the transaction is completed.
- c. **Isolated**- this property allows multiple users to access the same set of data and each user's processing should be isolated from others.
- d. **Durable** – Result of the transaction should be permanent once transaction is completed to avoid any loss of data.

It is very important to understand the difference between transaction boundaries and transaction demarcation. Starting and end point of a transaction are known as transaction boundaries and technique to identify transaction boundaries are known as transaction demarcation.

Concurrency

Database operations can be performed simultaneously (concurrently) by multiple users and this can end up with serious impacts. In order to control the concurrency , there are two approaches

- a. Optimistic - Versioning is used in this approach.
- b. Pessimistic – Acquiring Lock mechanism is used in this approach.

Locking refers to actions taken to prevent data in a relational database from changing between the time it is read and the time that it is used.

Your locking strategy can be either *optimistic* or *pessimistic*.

Locking strategies

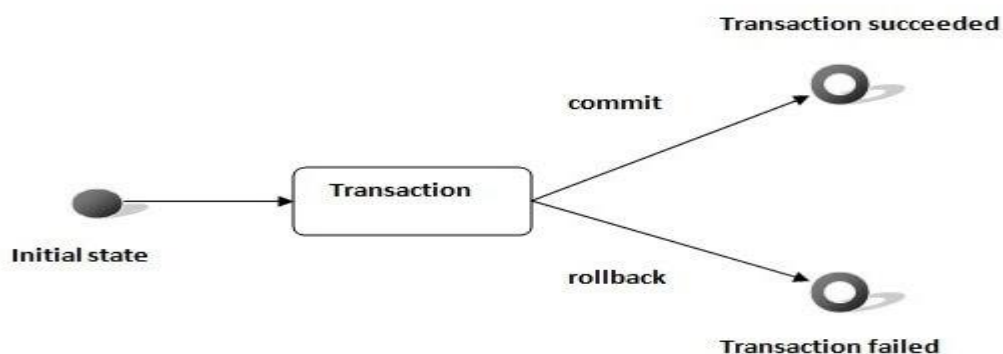
Optimistic

Optimistic locking assumes that multiple transactions can complete without affecting each other, and that therefore transactions can proceed without locking the data resources that they affect. Before committing, each transaction verifies that no other transaction has modified its data. If the check reveals conflicting modifications, the committing transaction rolls back^[1].

Pessimistic

Pessimistic locking assumes that concurrent transactions will conflict with each other, and requires resources to be locked after they are read and only unlocked after the application has finished using the data. Hibernate provides mechanisms for implementing both types of locking in your applications.

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 wasCommitted()** checks if the transaction is committed 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.

```
Session session = null;
Transaction tx = null;
try {
    session = sessionFactory.openSession();
    tx = session.beginTransaction();
    //some action
    tx.commit();
} catch (Exception ex) {
    ex.printStackTrace();
    tx.rollback();
}
finally { session.close();
}
```

Hibernate Transactions Interface

In Hibernate framework, we have Transaction interface that defines the unit of work. It maintains the abstraction from the transaction implementation (JTA, JDBC). A Transaction is associated with Hibernate Session and instantiated by calling the `sessionObj.beginTransaction()`.

The methods of Transaction interface are as follows:

Name	Description	Syntax
<code>begin()</code>	It starts a new transaction.	<code>public void begin() throws HibernateException</code>
<code>commit()</code>	It ends the transaction and flushes the associated session.	<code>public void commit() throws HibernateException</code>
<code>rollback()</code>	It rolls back the current transaction.	<code>public void rollback() throws HibernateException</code>
<code>setTimeout(int seconds)</code>	It set the transaction timeout for any transaction started by a subsequent call to <code>begin()</code> on this instance.	<code>public void setTimeout(int seconds) throws HibernateException</code>
<code>isActive()</code>	It checks if this transaction is still active or not.	<code>public boolean isActive() throws HibernateException</code>
<code>wasRolledBack()</code>	It checks if this transaction roll backed successfully or not.	<code>public boolean wasRolledBack() throws HibernateException</code>
<code>wasCommitted()</code>	It checks if this transaction committed successfully or not.	<code>public boolean wasCommitted() throws HibernateException</code>
<code>registerSynchronization(Synchronization synchronization)</code>	It registers a user synchronization callback for this transaction.	<code>public boolean registerSynchronization(Synchronization synchronization) throws HibernateException</code>

Hibernate Transaction Management Basic Structure

This is the basic structure that Hibernate programs should have, concerning Transaction Handling

```
Transaction transObj = null;
Session sessionObj = null;
try {
    sessionObj = HibernateUtil.buildSessionFactory().openSession();
    transObj = sessionObj.beginTransaction();

    //Perform Some Operation Here
}
```

```
        transObj.commit();
    } catch (HibernateException exObj) {
        if(transObj!=null){
            transObj.rollback();
        }
        exObj.printStackTrace();
    } finally {
        sessionObj.close();
    }
}
```

Whenever a `HibernateException` happens we call `rollback()` method that forces the rollback of the transaction. This means that every operation of that specific transaction that occurred before the exception, will be canceled and the database will return to its state before these operations took place.

UNIT- V

Spring Framework: Spring Basics, Spring Container, Spring AOP, Spring Data Access, Spring O-R/mapping, Spring Transaction Management, Spring Remoting and Enterprise Services, Spring Web MVC Framework, Securing Spring Application

Spring Framework:

Spring framework provides plenty of features. It helps application developers to perform the following functions: Create a Java method that runs in a database transaction with no help from transaction APIs. Create a local Java method that defines a remote procedure with no help from remote APIs. The Spring Framework is an application framework and inversion of control container for the Java platform. Spring is an open source development framework for enterprise Java. The core features of the Spring Framework can be used in developing any Java application, but there are extensions for building web applications on top of the Java EE platform. Spring framework targets to make J2EE development easier to use and promote good programming practice by enabling a POJO-based programming model. Spring framework is an open source Java platform that provides comprehensive infrastructure support for developing robust Java applications very easily and very rapidly. Spring framework was initially written by Rod Johnson and was first released under the Apache 2.0 license in June 2003.

Why to Learn Spring?

Spring is the most popular application development framework for enterprise Java. Millions of developers around the world use Spring Framework to create high performing, easily testable, and reusable code.

Spring framework is an open source Java platform. It was initially written by Rod Johnson and was first released under the Apache 2.0 license in June 2003.

Spring is lightweight when it comes to size and transparency. The basic version of Spring framework is around 2MB.

The core features of the Spring Framework can be used in developing any Java application, but there are extensions for building web applications on top of the Java EE platform. Spring framework targets to make J2EE development easier to use and promotes good programming practices by enabling a POJO-based programming model.

Applications of Spring

Following is the list of few of the great benefits of using Spring Framework –

- **POJO Based** - Spring enables developers to develop enterprise-class applications using POJOs. The benefit of using only POJOs is that you do not need an EJB container product such as an application server but you have the option of using only a robust servlet container such as Tomcat or some commercial product.
- **Modular** - Spring is organized in a modular fashion. Even though the number of packages and classes are substantial, you have to worry only about the ones you need and ignore the rest.
- **Integration with existing frameworks** - Spring does not reinvent the wheel, instead it truly makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, and other view technologies.
- **Testability** - Testing an application written with Spring is simple because environment-dependent code is moved into this framework. Furthermore, by using JavaBeanstyle POJOs, it becomes easier to use dependency injection for injecting test data.
- **Web MVC** - Spring's web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks such as Struts or other over-engineered or less popular web frameworks.
- **Central Exception Handling** - Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.

- **Lightweight** - Lightweight IoC containers tend to be lightweight, especially when compared to EJB containers, for example. This is beneficial for developing and deploying applications on computers with limited memory and CPU resources.
- **Transaction management** - Spring provides a consistent transaction management interface that can scale down to a local transaction (using a single database, for example) and scale up to global transactions (using JTA, for example).

HelloWorld.java

```
public class HelloWorld {
    private String message;

    public void setMessage(String message){
        this.message = message;
    }
    public void getMessage(){
        System.out.println("Your Message : " + message);
    }
}
```

MainApp.java

```
import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

    public static void main(String[] args) {

        ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

        HelloWorld obj = (HelloWorld) context.getBean("helloWorld");

        obj.getMessage();

    }
}
```

Following two important points are to be noted about the main program

- The first step is to create an application context where we used framework API `ClassPathXmlApplicationContext()`. This API loads beans configuration file and eventually based on the provided API, it takes care of creating and initializing all the objects, i.e. beans mentioned in the configuration file.
- The second step is used to get the required bean using `getBean()` method of the created context. This method uses bean ID to return a generic object, which finally can be casted to the actual object. Once you have an object, you can use this object to call any class method.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation = "http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
<bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld">
<property name = "message" value = "Hello World!"/>
</bean>
</beans>
```

When Spring application gets loaded into the memory, Framework makes use of the above configuration file to create all the beans defined and assigns them a unique ID as defined in `<bean>` tag. You can use `<property>` tag to pass the values of different variables used at the time of object creation.

Output:

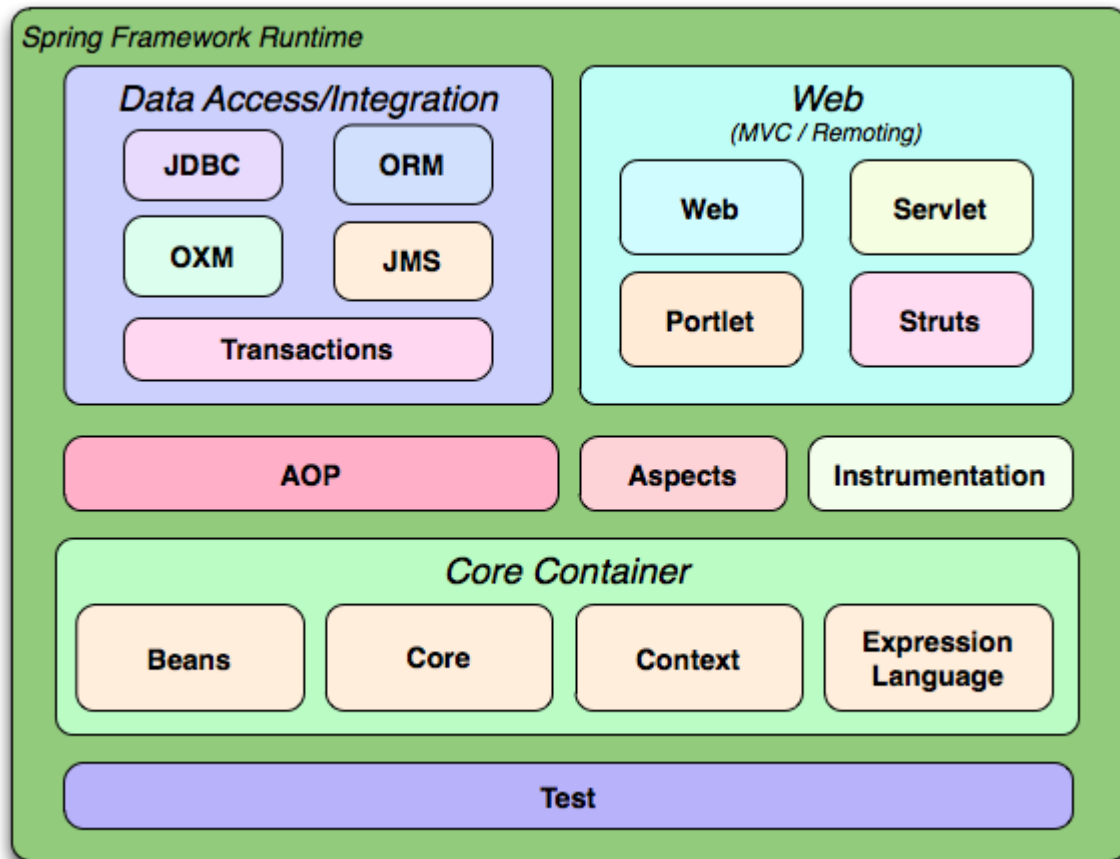
Your Message : Hello World!

The Spring Framework:

The Spring Framework Inversion of Control (IoC) component addresses this concern by providing a formalized means of composing disparate components into a fully working application ready for use. The Spring Framework codifies formalized design patterns as first-class objects that you can integrate into your own application(s). Numerous organizations and institutions use the Spring Framework in this manner to engineer robust, maintainable applications.

The Spring Framework consists of features organized into about 20 modules. These modules are grouped into Core Container, Data

Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, and Test, as shown in the following diagram.



Core Container:

The *Core Container* consists of the Core, Beans, Context, and Expression Language modules.

The *Core and Beans* modules provide the fundamental parts of the framework, including the IoC and Dependency Injection features. The `BeanFactory` is a sophisticated implementation of the factory pattern. It removes the need for programmatic singletons and allows you to decouple the configuration and specification of dependencies from your actual program logic.

The *Context* module builds on the solid base provided by the *Core and Beans* modules: it is a means to access objects in a framework-style manner that is similar to a JNDI registry. The Context module inherits its features from the Beans module and adds support for internationalization (using, for example, resource bundles), event-propagation, resource-loading, and the transparent creation of contexts by, for example, a servlet

container. The Context module also supports Java EE features such as EJB, JMX, and basic remoting. The `ApplicationContext` interface is the focal point of the Context module.

The *Expression Language* module provides a powerful expression language for querying and manipulating an object graph at runtime. It is an extension of the unified expression language (unified EL) as specified in the JSP 2.1 specification. The language supports setting and getting property values, property assignment, method invocation, accessing the context of arrays, collections and indexers, logical and arithmetic operators, named variables, and retrieval of objects by name from Spring's IoC container. It also supports list projection and selection as well as common list aggregations.

Data Access/Integration

The *Data Access/Integration* layer consists of the JDBC, ORM, OXM, JMS and Transaction modules.

The [JDBC](#) module provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes.

The [ORM](#) module provides integration layers for popular object-relational mapping APIs, including [JPA](#), [JDO](#), [Hibernate](#), and [iBatis](#). Using the ORM package you can use all of these O/R-mapping frameworks in combination with all of the other features Spring offers, such as the simple declarative transaction management feature mentioned previously.

The [OXM](#) module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.

The Java Messaging Service ([JMS](#)) module contains features for producing and consuming messages.

The [Transaction](#) module supports programmatic and declarative transaction management for classes that implement special interfaces and for *all your POJOs (plain old Java objects)*.

Web

The *Web* layer consists of the Web, Web-Servlet, Web-Struts, and Web-Portlet modules.

Spring's *Web* module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC

container using servlet listeners and a web-oriented application context. It also contains the web-related parts of Spring's remoting support.

The *Web-Servlet* module contains Spring's model-view-controller ([MVC](#)) implementation for web applications. Spring's MVC framework provides a clean separation between domain model code and web forms, and integrates with all the other features of the Spring Framework.

The *Web-Struts* module contains the support classes for integrating a classic Struts web tier within a Spring application. Note that this support is now deprecated as of Spring 3.0. Consider migrating your application to Struts 2.0 and its Spring integration or to a Spring MVC solution.

The *Web-Portlet* module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

AOP and Instrumentation

Spring's [AOP](#) module provides an *AOP Alliance*-compliant aspect-oriented programming implementation allowing you to define, for example, method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated. Using source-level metadata functionality, you can also incorporate behavioral information into your code, in a manner similar to that of .NET attributes.

The separate *Aspects* module provides integration with AspectJ. The *Instrumentation* module provides class instrumentation support and classloader implementations to be used in certain application servers.

Test

The Test module supports the testing of Spring components with JUnit or TestNG. It provides consistent loading of Spring ApplicationContexts and caching of those contexts. It also provides mock objects that you can use to test your code in isolation.

POJO vs Java Beans

POJO stands for Plain Old Java Object. It is an ordinary Java object, not bound by any special restriction other than those forced by the Java Language Specification and not requiring any classpath. POJOs are used for increasing the readability and re-usability of a program. POJOs have gained the most acceptance because they are easy to write and understand. They were introduced in EJB 3.0 by Sun Microsystems.

Properties of POJO

1. Extend prespecified classes, Ex: public class GFG extends javax.servlet.http.HttpServlet { ... } is **not** a POJO class.
2. Implement prespecified interfaces, Ex: public class Bar implements javax.ejb.EntityBean { ... } is **not** a POJO class.
3. Contain prespecified annotations, Ex: @javax.persistence.Entity public class Baz { ... } is **not** a POJO class.

POJOs basically define an entity. Like in your program, if you want an Employee class, then you can create a POJO as follows:

```
// Employee POJO class to represent entity Employee
public class Employee
{
    // default field
    String name;

    // public field
    public String id;

    // private salary
    private double salary;

    //arg-constructor to initialize fields
    public Employee(String name, String id,
                    double salary)
    {
        this.name = name;
        this.id = id;
        this.salary = salary;
    }

    // getter method for name
    public String getName()
    {
        return name;
    }

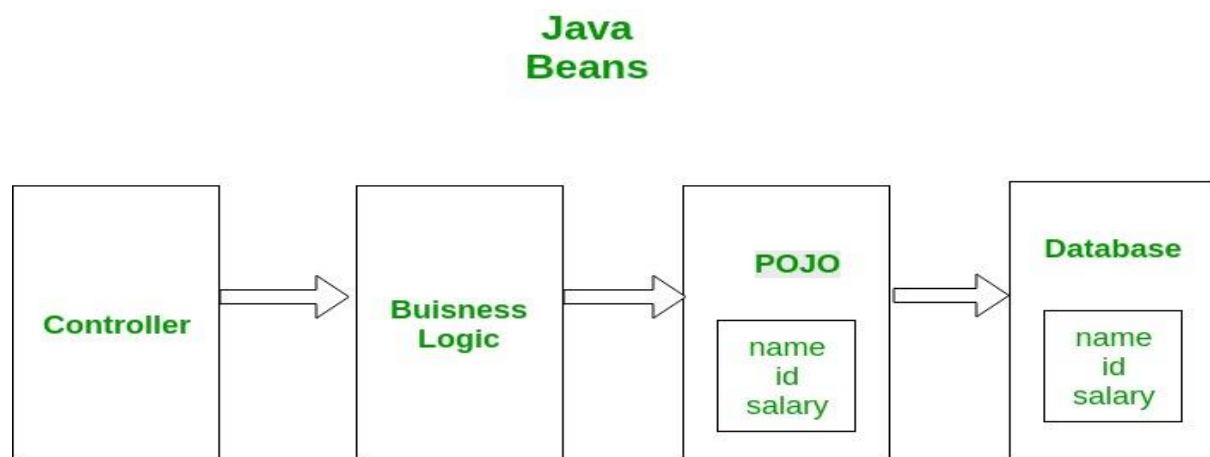
    // getter method for id
    public String getId()
    {
        return id;
    }

    // getter method for salary
    public Double getSalary()
    {
        return salary;
    }
}
```

POJO classes and Beans both are used to define java objects to increase their readability and reusability. POJOs don't have other restrictions while beans are special POJOs with some restrictions.

Explanation of the above program:

The above example is a well-defined example of the POJO class. As you can see, there is no restriction on access-modifiers of fields. They can be private, default, protected, or public. It is also not necessary to include any constructor in it. POJO is an object which encapsulates Business Logic. The following image shows a working example of the POJO class. Controllers interact with your business logic which in turn interact with POJO to access the database. In this example, a database entity is represented by POJO. This POJO has the same members as the database entity.



Java Beans

Beans are special type of Pojos. There are some restrictions on POJO to be a bean.

1. All JavaBeans are POJOs but not all POJOs are JavaBeans.
2. Serializable i.e. they should implement Serializable interface. Still, some POJOs who don't implement a Serializable interface are called POJOs

because Serializable is a marker interface and therefore not of many burdens.

3. Fields should be private. This is to provide complete control on fields.
4. Fields should have getters or setters or both.
5. A no-arg constructor should be there in a bean.
6. Fields are accessed only by constructor or getter setters.

Getters and Setters have some special names depending on field name. For example, if field name is someProperty then its **getter** preferably will be:

```
public "returnType" getSomeProperty()  
{  
    return someProperty;  
}
```

and setter will be

```
public void setSomePRoperty(someProperty)  
{  
    this.someProperty=someProperty;  
}
```

Visibility of getters and setters is generally public. Getters and setters provide the complete restriction on fields. e.g. consider below the property, Integer age;

If you set visibility of age to the public, then any object can use this. Suppose you want that age can't be 0. In that case, you can't have control. Any object can set it 0. But by using the setter method, you have control. You can have a condition in your setter method. Similarly, for the getter method if you want that if your age is 0 then it should return null, you can achieve this by using the getter method.

Below is the implementation of the above topic:

```
// Java program to illustrate JavaBeans  
class Bean implements Serializable {  
    // private field property  
    private Integer property;  
    public Bean()  
    {  
        // No-arg constructor  
    }  
  
    // setter method for property  
    public void setProperty(Integer property)  
    {  
        if (property == 0) {  
            // if property is 0 return  
            return;  
        }  
    }  
}
```

```

        this.property = property;
    }

    // getter method for property
    public Integer getProperty()
    {
        if (property == 0) {
            // if property is 0 return null
            return null;
        }
        return property;
    }
}

// Class to test above bean
public class GFG {
    public static void main(String[] args)
    {
        Bean bean = new Bean();

        bean.setProperty(0);
        System.out.println("After setting to 0: "
                           + bean.getProperty());

        bean.setProperty(5);
        System.out.println("After setting to valid"
                           + " value: "
                           + bean.getProperty());
    }
}

```

Output:

After setting to 0: null

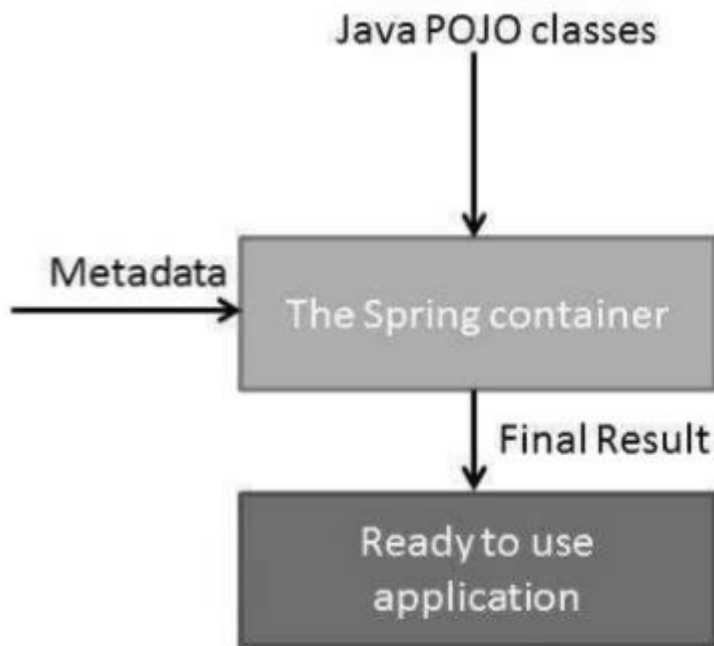
After setting to valid value: 5

POJO	Java Bean
It doesn't have special restrictions other than those forced by Java language.	It is a special POJO which have some restrictions.
It doesn't provide much control on members.	It provides complete control on members.
It can implement Serializable interface.	It should implement serializable interface.
Fields can be accessed by their names.	Fields are accessed only by getters and setters.
Fields can have any visibility.	Fields have only private visibility.
There may/may-not be a no-arg constructor.	It must have a no-arg constructor.
It is used when you don't want to give restriction on your members and give user complete access of your entity	It is used when you want to provide user your entity but only some part of your entity.

Spring Basics and Spring Container:

The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The Spring container uses DI to manage the components that make up an application. These objects are called Spring Beans.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code. The following diagram represents a high-level view of how Spring works. The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.



Spring provides the following two distinct types of containers.

Container & Description

Spring BeanFactory Container

This is the simplest container providing the basic support for DI and is defined by the *org.springframework.beans.factory.BeanFactory* interface. The *BeanFactory* and related interfaces, such as *BeanFactoryAware*, *InitializingBean*, *DisposableBean*, are still present in Spring for the purpose of backward compatibility with a large number of third-party frameworks that integrate with Spring.

Spring ApplicationContext Container

This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by the *org.springframework.context.ApplicationContext* interface.

The *ApplicationContext* container includes all functionality of the *BeanFactory* container, so it is generally recommended over *BeanFactory*. *BeanFactory* can still be used for lightweight applications like mobile devices or applet-based applications where data volume and speed is significant.

The differences between BeanFactory vs ApplicationContext in order to get a clear cutaway understanding of the spring IoC container which is as shown below in a tabular format below as follows:

The differences between BeanFactory vs ApplicationContext container

Feature	BeanFactory	ApplicationContext
Annotation Support	No	Yes
Bean Instantiation/Wiring	Yes	Yes
Internationalization	No	Yes
Enterprise Services	No	Yes
ApplicationEvent publication	No	Yes
Automatic BeanPostProcessor registration	No	Yes
Loading Mechanism	Lazy loading	Aggressive loading
Automatic BeanFactoryPostProcessor registration	No	Yes

Spring AOP:

Aspect-oriented Programming (AOP) complements Object-oriented Programming (OOP) by providing another way of thinking about program structure. The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Aspects enable the modularization of concerns (such as transaction management) that cut across multiple types and objects. (Such concerns are often termed "crosscutting" concerns in AOP literature.)

One of the key components of Spring is the AOP framework. While the Spring IoC container does not depend on AOP (meaning you do not need to use AOP if you don't want to), AOP complements Spring IoC to provide a very capable middleware solution. It accomplishes this task by adding extra behavior to existing code without modifying and changing the code of the software. We can declare the new code and the new behaviors separately as per our software requirements. Spring's AOP framework helps us to implement these cross-cutting concerns for our web applications.

One of the key components of Spring Framework is the Aspect oriented programming (AOP) framework. Aspect-Oriented Programming entails breaking down program logic into distinct parts called so-called concerns. The functions that span multiple points of an application are called cross-cutting concerns and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects like logging, auditing, declarative transactions, security, caching, etc.

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Dependency Injection helps you decouple your application objects from each other and AOP helps you decouple cross-cutting concerns from the objects that they affect. AOP is like triggers in programming languages such as Perl, .NET, Java, and others.

Spring AOP module provides interceptors to intercept an application. For example, when a method is executed, you can add extra functionality before or after the method execution.

AOP Terminologies

Before we start working with AOP, let us become familiar with the AOP concepts and terminology. These terms are not specific to Spring, rather they are related to AOP

S
r
.
N
o

Terms & Description

Aspect

This is a module which has a set of APIs providing cross-cutting requirements. For example, a logging module would be called AOP aspect for logging. An application can have any number of aspects depending on the requirement.

Join point

This represents a point in your application where you can plug-in the AOP aspect. You can also say, it is the actual place in the application where an action will be taken using Spring AOP framework.

Advice

This is the actual action to be taken either before or after the method execution. This is an actual piece of code that is invoked during the program execution by Spring AOP framework.

Pointcut

This is a set of one or more join points where an advice should be executed. You can specify pointcuts using expressions or patterns as we will see in our AOP examples.

Introduction

An introduction allows you to add new methods or attributes to the existing classes.

Target object

The object being advised by one or more aspects. This object will always be a proxied object, also referred to as

the advised object.

Weaving

Weaving is the process of linking aspects with other application types or objects to create an advised object. This can be done at compile time, load time, or at runtime.

Types of Advice:

Spring aspects can work with five kinds of advice mentioned as follows

Sr.No	Advice & Description
1	before Run advice before the a method execution.
2	after Run advice after the method execution, regardless of its outcome.
7 3	after-returning Run advice after the a method execution only if method completes successfully.
4	after-throwing Run advice after the a method execution only if method exits by throwing an exception.
5	around Run advice before and after the advised method is invoked.

Custom Aspects Implementation

Spring supports the **@AspectJ annotation style** approach and the **schema-based** approach to implement custom aspects. These two approaches have been explained in detail in the following sections.

Sr.No	Approach & Description
	XML Schema based
1	Aspects are implemented using the regular classes along with XML based configuration.
	@AspectJ based
	@AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations.
2	

Spring Data Access and Spring O-R/mapping

Spring Framework provides integration with *Hibernate*, *JDO*, *Oracle TopLink*, *iBATIS SQL Maps* and *JPA*: in terms of resource management, DAO implementation support, and transaction strategies. For example for Hibernate, there is first-class support with lots of IoC convenience features, addressing many typical Hibernate integration issues. All of these support packages for O/R (Object Relational) mappers comply with Spring's generic transaction and DAO exception hierarchies. There are usually two integration styles: either using Spring's DAO 'templates' or coding DAOs against plain Hibernate/JDO/TopLink/etc APIs. In both cases, DAOs can be configured through Dependency Injection and participate in Spring's resource and transaction management.

Spring adds significant support when using the O/R mapping layer of your choice to create data access applications. First of all, you should know that once you started using Spring's support for O/R mapping, you don't have to go all the way. No matter to what extent, you're invited to review and leverage the Spring approach, before deciding to take the effort and risk of building a similar infrastructure in-house. Much of the O/R mapping support, no matter what technology you're using may be used in a library style, as everything is designed as a set of reusable JavaBeans. Usage inside a Spring IoC container does provide additional benefits in terms of ease of configuration and deployment; as such, most examples in this section show configuration inside a Spring container.

Some of the benefits of using the Spring Framework to create your ORM DAOs include:

- ***Ease of testing.*** Spring's IoC approach makes it easy to swap the implementations and config locations of Hibernate `SessionFactory` instances, `JDBC DataSource` instances, transaction managers, and mapped object implementations (if needed). This makes it much easier to isolate and test each piece of persistence-related code in isolation.
- ***Common data access exceptions.*** Spring can wrap exceptions from your O/R mapping tool of choice, converting them from proprietary (potentially checked) exceptions to a common runtime `DataAccessException` hierarchy. This allows you to handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without annoying boilerplate catches/throws, and exception declarations. You can still trap and handle exceptions anywhere you need to. Remember that `JDBC` exceptions (including DB specific dialects) are also converted to the same hierarchy, meaning that you can perform some operations with `JDBC` within a consistent programming model.
- ***General resource management.*** Spring application contexts can handle the location and configuration of Hibernate `SessionFactory` instances, `JDBC DataSource` instances, iBATIS SQL Maps configuration objects, and other related resources. This makes these values easy to manage and change. Spring offers efficient, easy and safe handling of persistence resources. For example: related code using Hibernate generally needs to use the same Hibernate `Session` for efficiency and proper transaction handling. Spring makes it easy to transparently create and bind a `Session` to the current thread, either by using an explicit 'template' wrapper class at the Java code level or by exposing a current `Session` through the Hibernate `SessionFactory` (for DAOs based on plain Hibernate API). Thus Spring solves many of the issues that repeatedly arise from typical Hibernate usage, for any transaction environment (local or **JTA**).
- ***Integrated transaction management.*** Spring allows you to wrap your O/R mapping code with either a declarative, AOP style method interceptor, or an explicit 'template' wrapper class at the Java code level. In either case, transaction semantics are handled for you, and proper transaction handling (rollback, etc) in case of exceptions is taken care of. As discussed below, you also get the benefit of being able to use and swap various transaction managers, without your Hibernate/JDO related code being affected: for example, between local transactions and **JTA**, with the same full services (such as declarative transactions) available in both scenarios. As an additional benefit, `JDBC`-related code can

fully integrate transactionally with the code you use to do O/R mapping. This is useful for data access that's not suitable for O/R mapping, such as batch processing or streaming of BLOBs, which still needs to share common transactions with ORM operations.

The PetClinic sample in the Spring distribution offers alternative DAO implementations and application context configurations for JDBC, Hibernate, Oracle TopLink, and JPA. PetClinic can therefore serve as working sample app that illustrates the use of Hibernate, TopLink and JPA in a Spring web application. It also leverages declarative transaction demarcation with different transaction strategies.

Spring ORM

Spring-ORM is a technique or a Design Pattern used to access a relational database from an object-oriented language. ORM (Object Relation Mapping) covers many persistence technologies. They are as follows:

JPA(Java Persistence API): It is mainly used to persist data between Java objects and relational databases. It acts as a bridge between object-oriented domain models and relational database systems.

JDO(Java Data Objects): It is one of the standard ways to access persistent data in databases, by using plain old Java objects (POJO) to represent persistent data.

Hibernate – It is a Java framework that simplifies the development of Java applications to interact with the database.

Oracle Toplink, and iBATIS: Oracle TopLink is a mapping and persistence framework for Java development.

For the above technologies, Spring provides integration classes so that each of these techniques can be used following Spring principles of configuration, and easily integrates with Spring transaction management. For each of the above technologies, the configuration consists of injecting the DataSource bean into some kind of SessionFactory or EntityManagerFactory, etc. For pure JDBC(Java Database Connectivity), integration classes are not required apart from JdbcTemplate because JDBC only depends on a DataSource. If someone wants to use an ORM like JPA(Java Persistence API) or Hibernate then you do not need spring-JDBC, but only this module.

Note: The Spring Framework is an application framework and also an inversion of the control container for the Java platform. The framework's core features can be used by any of the Java applications, but there are some extensions for building web applications on top of the Java EE (Enterprise Edition) platform.

Advantages of the Spring Framework About ORM Frameworks

Due to the Spring framework, you do not need to write extra codes before and after the actual database logic such as getting the connection, starting the transaction, committing the transaction, closing the connection, etc.

Spring has an IoC(Inversion of control) approach which makes it easy to test the application.

Spring framework provides its API for exception handling along with the ORM framework.

By using the Spring framework, we can wrap our mapping code with an explicit template wrapper class or AOP(Aspect-oriented programming) style method interceptor.

Spring - Transaction Management

A database transaction is a sequence of actions that are treated as a single unit of work. These actions should either complete entirely or take no effect at all. Transaction management is an important part of RDBMS-oriented enterprise application to ensure data integrity and consistency. The concept of transactions can be described with the following four key properties described as **ACID** –

- **Atomicity** – A transaction should be treated as a single unit of operation, which means either the entire sequence of operations is successful or unsuccessful.
- **Consistency** – This represents the consistency of the referential integrity of the database, unique primary keys in tables, etc.
- **Isolation** – There may be many transaction processing with the same data set at the same time. Each transaction should be isolated from others to prevent data corruption.
- **Durability** – Once a transaction has completed, the results of this transaction have to be made permanent and cannot be erased from the database due to system failure.

A real RDBMS database system will guarantee all four properties for each transaction. The simplistic view of a transaction issued to the database using SQL is as follows –

- Begin the transaction using *begin transaction* command.
- Perform various deleted, update or insert operations using SQL queries.
- If all the operation are successful then perform *commit* otherwise *rollback* all the operations.

Spring framework provides an abstract layer on top of different underlying transaction management APIs. Spring's transaction support aims to provide an alternative to EJB transactions by adding transaction capabilities to POJOs. Spring supports both programmatic and declarative transaction management. EJBs require an application server, but Spring transaction management can be implemented without the need of an application server.

Local vs. Global Transactions

Local transactions are specific to a single transactional resource like a JDBC connection, whereas global transactions can span multiple transactional resources like transaction in a distributed system.

Local transaction management can be useful in a centralized computing environment where application components and resources are located at a single site, and transaction management only involves a local data manager running on a single machine. Local transactions are easier to be implemented.

Global transaction management is required in a distributed computing environment where all the resources are distributed across multiple systems. In such a case, transaction management needs to be done both at local and global levels. A distributed or a global transaction is executed across multiple systems, and its execution requires coordination between the global transaction management system and all the local data managers of all the involved systems.

Programmatic vs. Declarative

Spring supports two types of transaction management –

- **Programmatic transaction management** – This means that you have to manage the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.
- **Declarative transaction management** – This means you separate transaction management from the business code. You only use annotations or XML-based configuration to manage the transactions.

Declarative transaction management is preferable over programmatic transaction management though it is less flexible than programmatic transaction management, which allows you to control transactions through your code. But as a kind of crosscutting concern, declarative transaction management can be modularized with the AOP approach. Spring supports declarative transaction management through the Spring AOP framework.

Spring Transaction Abstractions

The key to the Spring transaction abstraction is defined by the *org.springframework.transaction.PlatformTransactionManager* interface, which is as follows –

```
public interface PlatformTransactionManager {
    TransactionStatus
    getTransaction(TransactionDefinition definition);
    throws TransactionException;

    void commit(TransactionStatus status) throws
    TransactionException;
    void rollback(TransactionStatus status) throws
    TransactionException;
}
```

Sr.No	Method & Description
1	TransactionStatus getTransaction(TransactionDefinition definition) This method returns a currently active transaction or

creates a new one, according to the specified propagation behavior.

void commit(TransactionStatus status)

- 2 This method commits the given transaction, with regard to its status.

void rollback(TransactionStatus status)

- 3 This method performs a rollback of the given transaction.

The *TransactionDefinition* is the core interface of the transaction support in Spring and it is defined as follows –

```
public interface TransactionDefinition {  
    int getPropagationBehavior();  
    int getIsolationLevel();  
    String getName();  
    int getTimeout();  
    boolean isReadOnly();  
}
```

Sr.No	Method & Description
	int getPropagationBehavior()
1	This method returns the propagation behavior. Spring offers all of the transaction propagation options familiar from EJB CMT.
	int getIsolationLevel()
2	This method returns the degree to which this transaction is isolated from the work of other transactions.
	String getName()
3	This method returns the name of this transaction.
	int getTimeout()
4	This method returns the time in seconds in which the transaction must complete.
	boolean isReadOnly()
5	This method returns whether the transaction is read-only.

Following are the possible values for isolation level –

Sr.No	Isolation & Description
1	TransactionDefinition.ISOLATION_DEFAULT This is the default isolation level.
2	TransactionDefinition.ISOLATION_READ_COMMITTED Indicates that dirty reads are prevented; non-repeatable reads and phantom reads can occur.
3	TransactionDefinition.ISOLATION_READ_UNCOMMITTED Indicates that dirty reads, non-repeatable reads, and phantom reads can occur.
4	TransactionDefinition.ISOLATION_REPEATABLE_READ Indicates that dirty reads and non-repeatable reads are prevented; phantom reads can occur.
5	TransactionDefinition.ISOLATION_SERIALIZABLE Indicates that dirty reads, non-repeatable reads, and phantom reads are prevented.

Following are the possible values for propagation types –

Sr.No.	Propagation & Description
1	TransactionDefinition.PROPROPAGATION_MANDATORY Supports a current transaction; throws an exception if no current transaction exists.
2	TransactionDefinition.PROPROPAGATION_NESTED Executes within a nested transaction if a current transaction exists.
3	TransactionDefinition.PROPROPAGATION_NEVER Does not support a current transaction; throws an exception if a current transaction exists.
4	TransactionDefinition.PROPROPAGATION_NOT_SUPPORTED Does not support a current transaction; rather always execute nontransactionally.
5	TransactionDefinition.PROPROPAGATION_REQUIRED Supports a current transaction; creates a new one if none exists.
6	TransactionDefinition.PROPROPAGATION_REQUIRES_NEW Creates a new transaction, suspending the current transaction if one exists.

- 7 **TransactionDefinition.PROPGATION_SUPPORTS**
Supports a current transaction; executes non-transactionally if none exists.
- 8 **TransactionDefinition.TIMEOUT_DEFAULT**
Uses the default timeout of the underlying transaction system, or none if timeouts are not supported.

The *TransactionStatus* interface provides a simple way for transactional code to control transaction execution and query transaction status.

```
public interface TransactionStatus extends
SavepointManager {
    boolean isNewTransaction();
    boolean hasSavepoint();
    void setRollbackOnly();
    boolean isRollbackOnly();
    boolean isCompleted();
}
```

Sr.No.	Method & Description
1	boolean hasSavepoint() This method returns whether this transaction internally carries a savepoint, i.e., has been created as nested transaction based on a savepoint.
2	boolean isCompleted() This method returns whether this transaction is completed, i.e., whether it has already been committed or rolled back.
3	boolean isNewTransaction() This method returns true in case the present transaction is new.
4	boolean isRollbackOnly() This method returns whether the transaction has been marked as rollback-only.
5	void setRollbackOnly() This method sets the transaction as rollback-only.

Spring Remoting and Enterprise Services

Spring has integration classes for remoting support that use a variety of technologies. The Spring framework simplifies the development of remote-enabled services. It saves a significant amount of code by having its own API. The remote support simplifies the building of remote-enabled services, which are implemented by your standard (Spring) POJOs. Spring now supports the following remoting technologies:

1. Remote Method Invocation (RMI)
2. Hessian
3. Burlap
4. Spring's HTTP invoker
5. JAX-RPC
6. JAX-WS
7. JMS

1. Exposing services using RMI

Transparently expose your services across the RMI infrastructure by using Spring's RMI support. After you've done this, you'll have a setup that's similar to remote EJBs, except there's no standard support for security context propagation or remote transaction propagation. When utilizing the RMI invoker, Spring provides hooks for such extra invocation context, so you may, for example, plug-in security frameworks or custom security credentials.

Using the `RmiServiceExporter` to export the service.
Client-side service integration.

2. Using Hessian to call services remotely through HTTP

Spring has its own remoting service that supports HTTP serialization. HTTP Invoker makes use of the classes `HttpInvokerServiceExporter` and `HttpInvokerProxyFactoryBean`.

- Configuring the `DispatcherServlet` for Hessian and company.
- Using the `HessianServiceExporter` to expose your beans
- Connecting the client to the service

3. Using Burlap

It is the same as Hessian but XML-based implementation provided by Coucho. The classes used in Burlap are `BurlapServiceExporter` and `BurlapProxyFactoryBean`.

4. Using HTTP invokers to expose services

Spring HTTP invokers employ the regular Java serialization technique to expose services through HTTP, as opposed to Burlap and Hessian, which are both lightweight protocols with their own compact serialization mechanisms. This is especially useful if your arguments and return types are complicated and cannot be serialized using the serialization procedures used by Hessian and Burlap (refer to the next section for more considerations when choosing a remoting technology).

- Displaying the service object
- Client-side service integration

5. Exposing servlet-based web services using JAX-RPC

Spring provides a convenience base class for JAX-RPC servlet endpoint implementations – `ServletEndpointSupport`. To expose our `AccountService` we extend Spring's `ServletEndpointSupport` class and implement our business logic here, usually delegating the call to the business layer.

Using JAX-RPC to gain access to online services

JAX-RPC Bean Mappings Registration

Adding your own JAX-RPC Handler

6. Using JAX-WS to provide servlet-based web services

`SpringBeanAutowiringSupport` is a useful basic class for JAX-WS servlet endpoint implementations. We modify Spring's `SpringBeanAutowiringSupport` class to expose our `AccountService` and execute our business logic here, generally outsourcing the request to the business layer.

7. JMS

Using JMS (Java Message Service) as the underlying communication protocol, it is also feasible to provide services transparently. The Spring Framework's JMS remoting functionality is fairly minimal – it transmits and receives on the same thread and in the same non-transactional Session, thus performance will be very implementation dependant. It's worth noting that these single-threaded and non-transactional restrictions only apply to Spring's JMS remoting functionality.

Enterprise Services

Spring framework helps develop various types of applications using the Java platforms. It provides an extensive level of infrastructure support. Spring also provides the “Plain Old Java Objects” (POJOs) mechanisms using which developers can easily create the Java SE programming model with the full and partial JAVA EE(Enterprise Edition).

Spring strives to facilitate the complex and unmanageable enterprise Java application development revolution by offering a framework that incorporates technologies, such as:

Aspect-oriented Programming (AOP)

Dependency Injection (DI)

Plain Old Java Object (POJO)

Spring framework provides plenty of features. It helps application developers to perform the following functions:

- Create a Java method that runs in a database transaction with no help from transaction APIs.
- Create a local Java method that defines a remote procedure with no help from remote APIs.
- Create a local Java method for a management operation with no help from JMX APIs.
- Create a local Java method for a message handler with no help from JMS APIs.

Spring is a lightweight framework. It provides the best mechanisms for different frameworks, such as Struts, [Hibernate](#), EJB, JSF, and Tapestry. It helps solve real-time technical problems. Spring contains multiple modules, such as WEB MVC, IOC, DAO, AOP, Context, and ORM.

Spring also helps create scalable, secure, and robust business-based web applications. We can consider the Spring framework a cluster of sub frameworks such as Spring Web Flow, Spring ORM, and Spring MVC. In expansion to Java, Spring also sustains Kotlin and Groovy.

The Spring framework provides a base that controls all the other Spring-based projects, such as:

- Spring Boot
- Spring Cloud
- Spring GraphQL

Spring Framework in Java: Advantages

Using the Spring framework, developers can leverage the below-listed advantages:

- Pre-defined Templates

Spring framework contains various types of templates for Hibernate, JDBC, and JPA technologies. With the help of this approach, developers are not required to define complex code.

Example: JdbcTemplate - Here, we do not need to write the logic for creating a statement, committing the transaction, creating a connection, and exception handling. It saves the time-consuming approach.

- Loose Coupling

We can consider Spring applications to be loosely coupled as per the dependency injection mechanisms.

- Easy and Simple to Test

It is easy to test the entire application using a spring framework with a dependency injection mechanism. The EJB or Struts application requires the server to execute the application.

- Non-invasive

As per the Plain Old Java Object (POJO) technique, Spring is easy to implement as it does not force the developer to inherit certain classes or implementations on any interface.

- Fast Development

With the help of Dependency Injection, it is easy to integrate the framework and support the development of JavaEE-based applications.

- Strong Abstraction Support

Spring supports the strong abstraction capability for Java EE-based specifications, such as JMS, JDBC, JPA, and JTA.

- Spring's Web Framework is Well-Organized

It is a web [MVC framework](#) that delivers a fantastic option to web frameworks for developing applications using Struts or different widespread web frameworks.

- Spring Delivers a Suitable API

It translates technology-specific anomalies thrown by JDBC, Hibernate, or JDO into uniform, uncontrolled exceptions.

- Lightweight IoC

It is lightweight, particularly when compared to EJB containers, for example. This helps create and deploy applications on computers with restricted memory and CPU resources.

- Constant Transaction Management

Spring provides an interface that can help scale down to a local transaction (for example, using a single database) and scale up to global transactions (for example, JTA).

Spring Core

In the Spring framework, we have certain features as discussed below:

- **Dependency Injection (DI)**

Dependency Injection is the core of Spring Framework. We can define the concept of Spring with the Inversion of Control (IoC). DI allows the creation of dependent objects outside of a class and provides those objects to a class in different ways. Dependency Injection can be utilized while defining the parameters to the Constructor or by post-construction using Setter methods.

The dependency feature can be summarized into an association between two classes. For example, suppose class X is dependent on class Y. Now, it can create many problems in the real world, including system failure. Hence such dependencies need to be avoided. [Spring](#) IOC resolves such dependencies with Dependency Injection. Here, it indicates that class Y will get injected into class X by the IoC. DI thus makes the code easier to test and reuse.

While creating a complex Java application, application classes should be independent of other Java classes to improve the possibility of reusing these classes and to test them independently of other classes during unit testing. Dependency Injection enables these classes to be together, and at the same time, keeps them independent.

- **Support for Aspect-Oriented Programming**

AOP provides more modularity to the cross-cutting challenges in applications.

Here are the functions we can use in our applications as per certain real-time challenges:

1. Logging
2. Caching
3. Transaction management
4. Authentication

AOP has the in-built object-oriented programming capabilities to define the structure of the program, where OOP modularity is established in classes.

In AOP, the primary unit of modularity is a factor (cross-cutting concern). This allows users to use AOP to build custom aspects and declarative enterprise

services. The IoC container does not depend on AOP; it provides the custom enabled based capabilities which allow writing logic as per the programming method.

However, Aspect-Oriented Programming integrated with the Spring IoC delivers a robust middleware solution.

- Data Access Framework

Database communication problems are one of the common challenges which developers encounter when creating applications. Spring facilitates the database communication strategy by delivering immediate support for widespread data access frameworks in Java, such as Hibernate, JDBC, and Java Persistence API (JPA).

Additionally, it suggests resource management, exception handling, and resource wrapping for all the supported data access frameworks, further streamlining the development revolution.

- Transaction Management Framework

Java Transaction API (JTA), the Spring Transaction Management Framework, is not restricted to nested and global types of transactions. Spring presents an abstraction mechanism for Java that permits users to:

1. Work with local, international, and nested transactions wise logics
2. Savepoints
3. Simplify transaction management across the application

The Spring Data Access Framework instantly combines with the Transaction Management Framework with help for messaging and caching. This allows developers to build feature-rich transactional systems that span across the applications without relying on EJB or JTA.

- Spring MVC Framework

The Spring MVC allows developers to develop applications utilizing the popular MVC pattern. It is a request-based framework that enables developers to develop custom MVC implementations that efficiently serve their needs.

The core component of Spring MVC is the `DispatcherServlet` class, which manages user requests and then delivers them to the right controller. This permits the controller to process the request, create the model, and then deliver the data to the end-user via a restricted view.

- Spring Web Service

This Spring Web Service component supplies a streamlined way to build and handle web service endpoints in the application. It delivers a layered approach that can be controlled using [XML](#). It can also be used to deliver mapping for web requests to a specific object.

- Spring Test Frameworks

Testing is a key component of application development. Spring streamlines testing within the framework with components like:

1. Mock objects
2. TestContext framework
3. Spring MVC Test

Core Container

This includes the essential modules that are the cornerstone of the Spring framework.

- Core (spring-core) is the framework's core that controls features such as Inversion of Control and dependency injection.
- Beans (spring-beans) deliver BeanFactory, an advanced execution of the factory pattern.
- Context (spring-context) produces on Core and Beans and delivers a medium to access restricted objects. ApplicationContext interface is the core part of the Context module, and the spring-context support provides help for third-party interactions such as caching, mailing, and template engines.
- SpEL (spring-expression) allows users to use the Spring Expression Language to query and control the object graph at execution time.

Data Access/Integration

This contains the modules used to manage data access and transaction processing in an application.

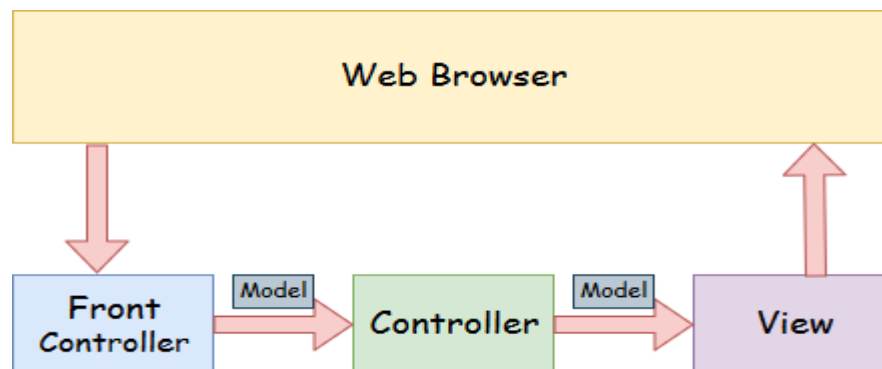
- JDBC (spring-jdbc) delivers a JDBC abstraction layer that eliminates the need to split JDBC coding when dealing using databases.

- ORM (spring-orm) are essential integration layers for overall object-relational mapping API, for example, JDO Hibernate, JPA, etc.
- OXM (spring-oxm) is the abstraction layer that supports Object/XML mapping implementations, for example, JAXB, XStream, etc.
- JMS (spring-jms) is the Java Messaging Service module that constructs and consumes messages that instantly incorporate the Spring messaging module.
- Transaction (spring-tx) offers programmatic and declarative transaction management for classes that include unique interfaces and POJOs.

Spring Web MVC Framework

A Spring MVC is a Java framework which is used to build web applications. It follows the Model-View-Controller design pattern. It implements all the basic features of a core spring framework like Inversion of Control, Dependency Injection.

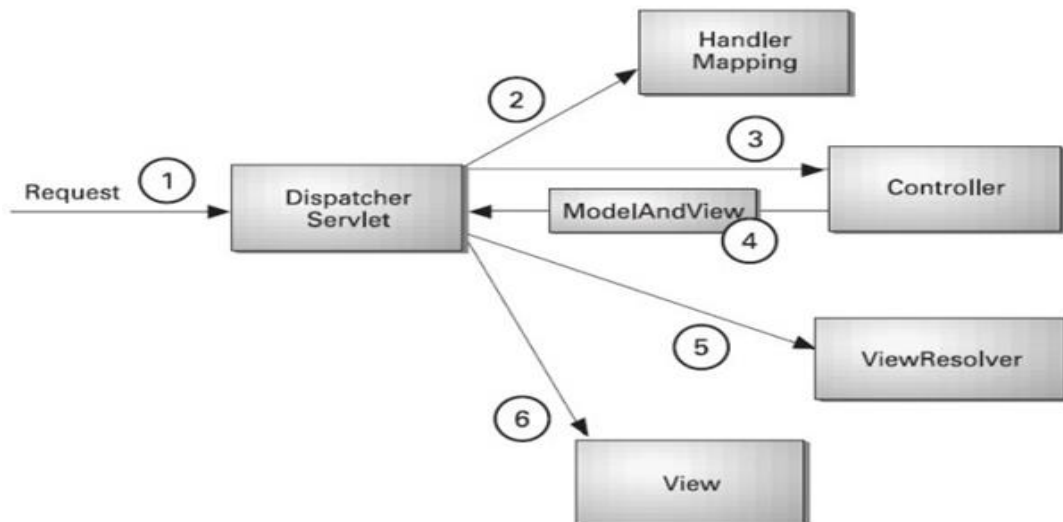
A Spring MVC provides an elegant solution to use MVC in spring framework by the help of **DispatcherServlet**. Here, **DispatcherServlet** is a class that receives the incoming request and maps it to the right resource such as controllers, models, and views.



- **Model** - A model contains the data of the application. A data can be a single object or a collection of objects.
- **Controller** - A controller contains the business logic of an application. Here, the @Controller annotation is used to mark the class as the controller.
- **View** - A view represents the provided information in a particular format. Generally, JSP+JSTL is used to create a view page. Although spring also supports other view technologies such as Apache Velocity, Thymeleaf and FreeMarker.

- **Front Controller** - In Spring Web MVC, the DispatcherServlet class works as the front controller. It is responsible to manage the flow of the Spring MVC application.

Understanding the flow of Spring Web MVC



- As displayed in the figure, all the incoming request is intercepted by the DispatcherServlet that works as the front controller.
- The DispatcherServlet gets an entry of handler mapping from the XML file and forwards the request to the controller.
- The controller returns an object of ModelAndView.
- The DispatcherServlet checks the entry of view resolver in the XML file and invokes the specified view component.

Advantages of Spring MVC Framework

- **Separate roles** - The Spring MVC separates each role, where the model object, controller, command object, view resolver, DispatcherServlet, validator, etc. can be fulfilled by a specialized object.
- **Light-weight** - It uses light-weight servlet container to develop and deploy your application.
- **Powerful Configuration** - It provides a robust configuration for both framework and application classes that includes easy referencing across contexts, such as from web controllers to business objects and validators.

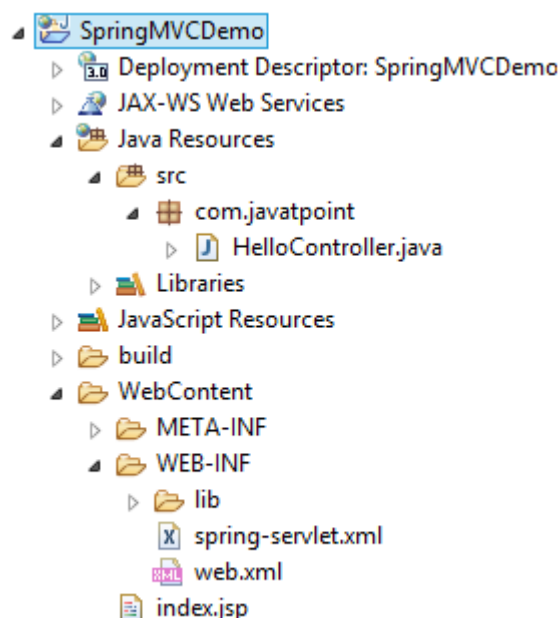
- **Rapid development** - The Spring MVC facilitates fast and parallel development.
- **Reusable business code** - Instead of creating new objects, it allows us to use the existing business objects.
- **Easy to test** - In Spring, generally we create JavaBeans classes that enable you to inject test data using the setter methods.
- **Flexible Mapping** - It provides the specific annotations that easily redirect the page.

Example of a Spring Web MVC framework

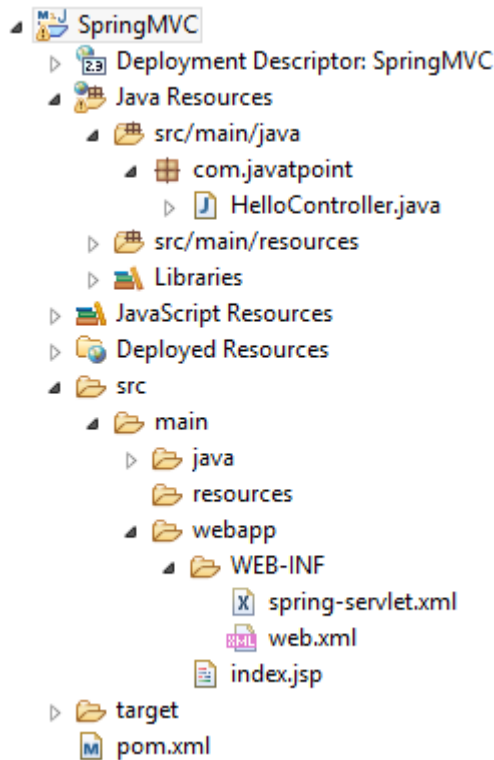
Procedures:

- > Load the spring jar files or add dependencies in the case of Maven
- > Create the controller class
- > Provide the entry of controller in the web.xml file
- > Define the bean in the separate XML file
- > Display the message in the JSP page
- > Start the server and deploy the project

Directory Structure of Spring MVC



Directory Structure of Spring MVC using Maven



Required Jar files or Maven Dependency

To run this example, you need to load:

- Spring Core jar files
- Spring Web jar files
- JSP + JSTL jar files (If you are using any another view technology then load the corresponding jar files).

If you are using Maven, you don't need to add jar files. Now, you need to add maven dependency to the pom.xml file.

1. Provide project information and configuration in the pom.xml file.

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javatpoint</groupId>
  <artifactId>SpringMVC</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>SpringMVC Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>5.1.1.RELEASE</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>3.0-alpha-1</version>
    </dependency>

  </dependencies>
  <build>
```

```
<finalName>SpringMVC</finalName>
</build>
</project>
```

2. Create the controller class

To create the controller class, we are using two annotations @Controller and @RequestMapping.

The @Controller annotation marks this class as Controller.

The @RequestMapping annotation is used to map the class with the specified URL name.

HelloController.java

```
package com.javatpoint;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class HelloController {
    @RequestMapping("/")
    public String display()
    {
        return "index";
    }
}
```

3. Provide the entry of controller in the web.xml file

In this xml file, we are specifying the servlet class DispatcherServlet that acts as the front controller in Spring Web MVC. All the incoming request for the html file will be forwarded to the DispatcherServlet.

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://jav
a.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd" id="WebApp_ID" version="3.0">
    <display-name>SpringMVC</display-name>
    <servlet>
        <servlet-name>spring</servlet-name>
```

```

    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>

```

4. Define the bean in the xml file

This is the important configuration file where we need to specify the View components.

The context:component-scan element defines the base-package where DispatcherServlet will search the controller class.

This xml file should be located inside the WEB-INF directory.

spring-servlet.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <!-- Provide support for component scanning -->
    <context:component-scan base-package="com.javatpoint" />

    <!--Provide support for conversion, formatting and validation -->
    <mvc:annotation-driven/>

```

`</beans>`

5. Display the message in the JSP page

This is the simple JSP page, displaying the message returned by the Controller.

`index.jsp`

`<html>`

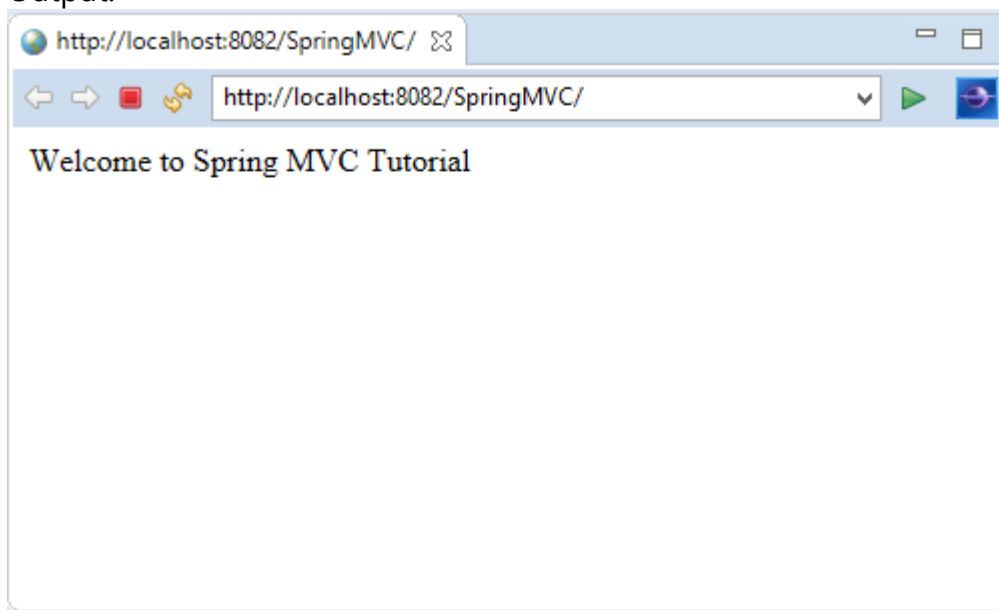
`<body>`

`<p>Welcome to Spring MVC Tutorial</p>`

`</body>`

`</html>`

Output:



Securing Spring Application

Spring Security provides ways to perform authentication and authorization in a web application. We can use spring security in any servlet based web application.

Some of the benefits of using Spring Security are:

1. Proven technology, it's better to use this than reinvent the wheel. Security is something where we need to take extra care, otherwise our application will be vulnerable for attackers.
2. Prevents some of the common attacks such as CSRF, session fixation attacks.
3. Easy to integrate in any web application. We don't need to modify web application configurations, spring automatically injects security filters to the web application.
4. Provides support for authentication by different ways - in-memory, DAO, JDBC, LDAP and many more.
5. Provides option to ignore specific URL patterns, good for serving static HTML, image files.

6. Support for groups and roles.

Spring Security is a powerful and highly customizable security framework that provides authentication, authorization, and other security features for Spring-based applications. It is a widely used open-source project that helps developers to secure their web applications by implementing security policies and rules. Spring Security provides a set of APIs and classes that can be used to easily configure and implement various security features in web applications. It also provides integration with other Spring Framework components such as Spring MVC, Spring Boot, and others, making it easier to use and integrate with existing applications.

Some of the key features of Spring Security include support for multiple authentication mechanisms, such as form-based authentication, token-based authentication, and others, robust authorization capabilities based on roles and permissions, support for various security protocols such as HTTPS, OAuth, and SAML, and comprehensive logging and auditing capabilities for monitoring security-related events. The core of Spring Security is based on a set of filters that intercept and process incoming HTTP requests, allowing developers to define rules for different types of requests and user roles. Spring Security also provides a range of configuration options that can be used to customize its behavior and integrate it with other Spring modules.

Why is Security Important for Spring Applications?

Security is crucial for any web application, and it is particularly important for Spring applications because of their widespread use in enterprise environments. Here are some reasons why security is important for Spring applications:

1. **Protection against cyber attacks:** Cyber attacks such as hacking, malware, and phishing attacks are becoming increasingly sophisticated and prevalent. A robust security framework is essential to protect against these threats and prevent unauthorized access to sensitive data.
2. **Trust from users:** Users expect their personal information to be protected when using web applications. If a Spring application is not secure, users may lose trust in the application and the company behind it, leading to a loss of business.
3. **Compliance:** Many organizations are subject to regulatory compliance requirements that mandate the implementation of specific security controls. Failure to comply with these requirements can result in legal and financial penalties.
4. **Competitive advantage:** Having strong security measures in place can give a company a competitive advantage by demonstrating to customers that they take their security seriously.

Basic Concepts of Spring Security

Authentication and authorization are two critical security aspects provided by Spring Security.

Authentication

- Authentication is the process of verifying the Identity of a user or system. Spring Security provides a wide range of authentication options, including in-memory authentication, JDBC authentication, LDAP authentication, and OAuth 2.0 authentication.
- The authentication process typically involves the following steps:
 - The user provides their credentials (e.g. username and password).
 - The credentials are validated against a configured authentication provider (e.g. a user database).
 - If the credentials are valid, a security context is established for the user.
- Once authentication is complete, Spring Security provides a range of options for authorization.

Authorization

Authorization refers to the process of determining whether a user is allowed to perform a particular action or access a particular resource.

Spring Security's authorization options include:

Role-based access control: This is a common authorization strategy where access to resources is granted based on the user's role or group membership. Spring Security provides support for role-based access control through the use of annotations, configuration files, or expressions.

Permission-based access control: This strategy grants access to specific resources based on the user's permissions or privileges. Spring Security supports permission-based access control through the use of annotations or expressions.

Web access control: This is a specialized form of authorization that controls access to web resources such as URLs or HTTP methods. Spring Security provides support for web access control through the use of request-matching rules and access control lists.

Security Filters

- In Spring Security, security filters are components that intercept incoming HTTP requests and perform various security-related tasks. These filters

are responsible for enforcing security policies, such as authentication, authorization, and other security checks.

- Security filters in Spring Security play a crucial role in ensuring the application's security by intercepting and processing requests and enforcing security rules and configurations defined in the application.
- Spring Security provides a number of pre-built security filters that can be used to configure security for your application, including filters for handling **authentication**, **authorization**, **CSRF protection**, **session management**, and more. You can also create your own custom security filters by implementing the [javax.servlet.Filter](#) the interface and configure them in your Spring Security configuration. This allows you to add additional security checks or customize the behavior of existing filters.

Security Providers

In Spring Security, security providers are responsible for authenticating users and managing user credentials. A security provider is essentially a pluggable authentication mechanism that Spring Security uses to validate user credentials and establish an authenticated session.

- Spring Security supports various types of security providers, including:
 - **InMemoryUserDetailsManager:** This provider stores user credentials and roles in memory and is useful for simple applications with a small number of users.
 - **JdbcUserDetailsManager:** This provider retrieves user credentials and roles from a database table and is useful for applications that need to store user information in a database.
 - **LDAP authentication provider:** This provider authenticates users against an LDAP server, which is commonly used in enterprise environments.
 - **OAuth2 authentication provider:** This provider authenticates users using OAuth2, which is useful for applications that need to allow users to sign in using their existing social media or Google accounts.
 - **OpenID Connect authentication provider:** This provider authenticates users using OpenID Connect, which is an authentication layer on top of OAuth2.
 - **Anonymous authentication provider:** This provider allows unauthenticated users to access certain resources in the application without the need to log in.

Example: Spring Security Login-Logout Module Example

Spring Security provides login and logout features that we can use in our application. It is helpful to create secure Spring application.

Here, we are creating a Spring MVC application with Spring Security and implementing login and logout features.

First we created a maven project and provided following project dependencies in pom.xml file.

Project Dependencies

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javatpoint</groupId>
  <artifactId>springSecurityLoginOut</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>
  <properties>
    <maven.compiler.target>1.8</maven.compiler.target>
    <maven.compiler.source>1.8</maven.compiler.source>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>5.0.2.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework.security</groupId>
      <artifactId>spring-security-web</artifactId>
      <version>5.0.0.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework.security</groupId>
```

```

    <artifactId>spring-security-core</artifactId>
    <version>5.0.0.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>5.0.0.RELEASE</version>
</dependency>

    <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
            <version>2.6</version>
            <configuration>
                <failOnMissingWebXml>>false</failOnMissingWebXml>
            </configuration>
        </plugin>
    </plugins>
</build>
</project>

```

Spring Security Configuration

After that we created configuration files to enable login feature and allowed access to the authorized user only.

This project contains the following four Java files.

AppConfig.java

```
package com.javatpoint;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.view.InternalResourceViewResolver;
import org.springframework.web.servlet.view.JstlView;

@EnableWebMvc
@Configuration
@ComponentScan({ "com.javatpoint.controller.*" })
public class AppConfig {
    @Bean
    public InternalResourceViewResolver viewResolver() {
        InternalResourceViewResolver viewResolver
            = new InternalResourceViewResolver();
        viewResolver.setViewClass(JstlView.class);
        viewResolver.setPrefix("/WEB-INF/views/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }
}
```

MvcWebApplicationInitializer.java

```
package com.javatpoint;
import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class MvcWebApplicationInitializer extends
    AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[] { WebSecurityConfig.class };
    }
    @Override
    protected Class<?>[] getServletConfigClasses() {
        // TODO Auto-generated method stub
        return null;
    }
    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

SecurityWebApplicationInitializer.java

```
package com.javatpoint;
import org.springframework.security.web.context.*;
public class SecurityWebApplicationInitializer
    extends AbstractSecurityWebApplicationInitializer {
}
```

WebSecurityConfig.java

```
package com.javatpoint;
import org.springframework.context.annotation.*;
//import org.springframework.security.config.annotation.authentication.builders.*;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.*;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
@EnableWebSecurity
@ComponentScan("com.javatpoint")
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Bean
    public UserDetailsService userDetailsService() {
        InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();

        manager.createUser(User.withDefaultPasswordEncoder()
            .username("irfan").password("khan").roles("ADMIN").build());
        return manager;
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {

        http
            .authorizeRequests()
                .anyRequest().hasRole("ADMIN")
                .and().formLogin().and()
            .httpBasic()
            .and()
            .logout()
            .logoutUrl("/j_spring_security_logout")
            .logoutSuccessUrl("/")
            ;
    }
}
```

```
}  
}
```

Controller

HomeController: Controller to handle user requests.

```
package com.javatpoint.controller;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import org.springframework.security.core.Authentication;  
import org.springframework.security.core.context.SecurityContextHolder;  
import org.springframework.security.web.authentication.logout.SecurityContextLogoutHandler;  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestMethod;  
  
@Controller  
public class HomeController {  
    @RequestMapping(value = "/", method = RequestMethod.GET)  
    public String index() {  
        return "index";  
    }  
  
    @RequestMapping(value="/logout", method=RequestMethod.GET)  
    public String logoutPage(HttpServletRequest request, HttpServletResponse response) {  
        Authentication auth = SecurityContextHolder.getContext().getAuthentication();  
        if (auth != null){  
            new SecurityContextLogoutHandler().logout(request, response, auth);  
        }  
        return "redirect:/";  
    }  
}
```

Views

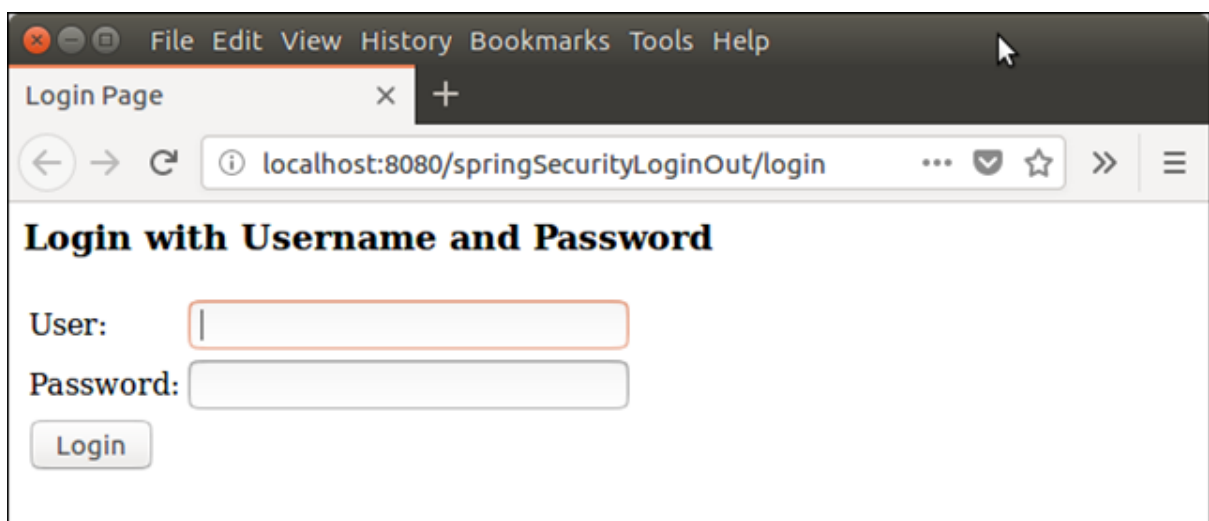
We have a JSP file **index.jsp** that contains the following code.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
    <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Home</title>
</head>
<body>
<h3> Hello ${pageContext.request.userPrincipal.name}, </h3>
<h4>Welcome to Javatpoint! </h4>
<a href="<c:url value='/logout' />">Click here to logout</a>
</body>
</html>
```




Output

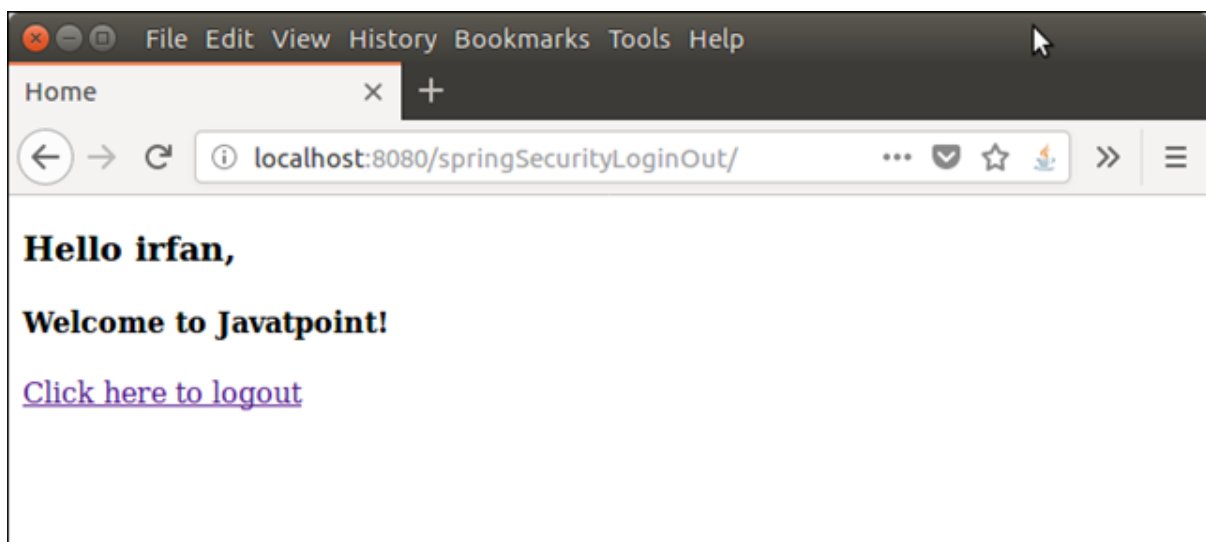
When run using apache tomcat, it produces the following output to the browser.



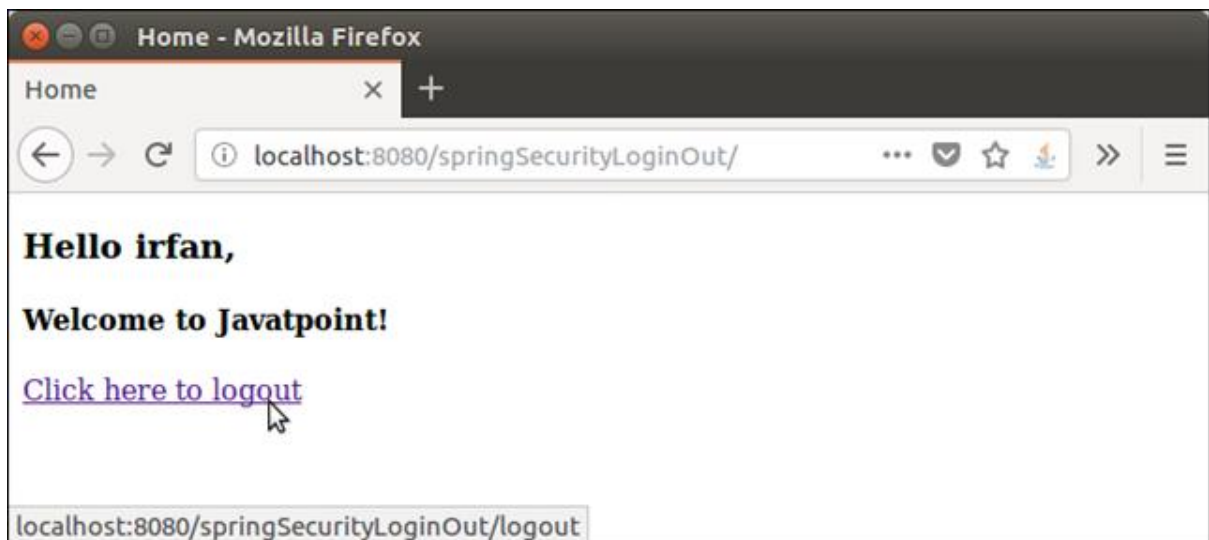
Now, providing user credentials to get logged in.



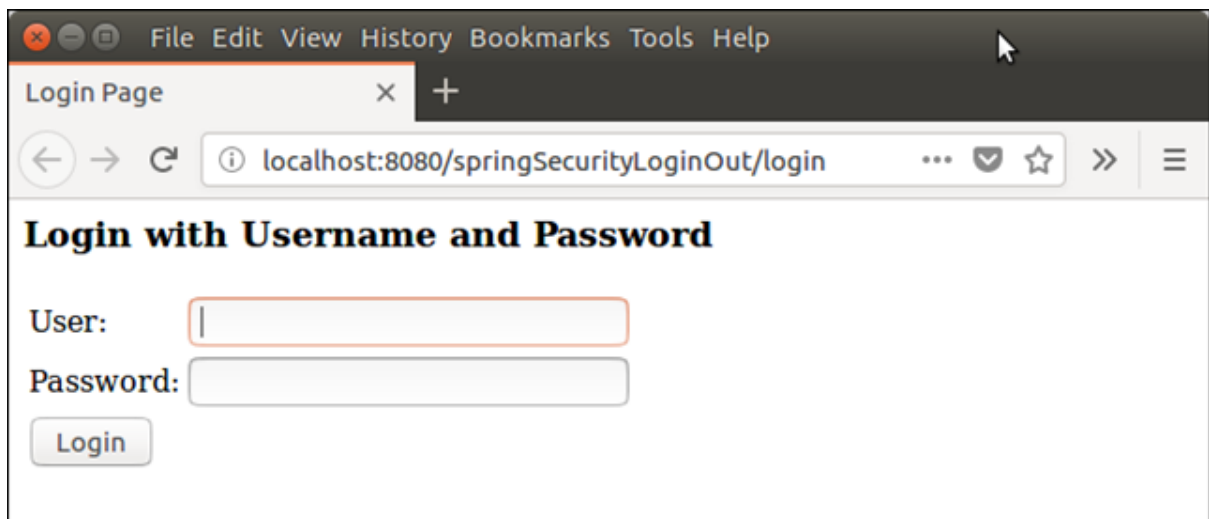
It shows home page after logged in successfully, see below.



Here, we are created a logout link which can be used to get logged out. Let's check out and log out from the application.



And it redirect back to the login page.



Well, we have created a successfully Spring MVC application that uses Spring Security to implement login and logout features.