

SYMBIOSIS UNIVERSITY OF APPLIED SCIENCES (SUAS)

**India’s 1st Skill Development University**

**PRACTICAL JOURNAL**

Enrollment Number – 2019BTCS088 Year of Enrollment – 2019-2023

Name of the Student – YASH GUPTA

School of COMPUTER SCIENCE & INFORMATION TECHNOLOGY

Program – B. TECH

Specialization/ Branch – CS&IT

Semester - 3rd Section – B2 Branch – CS&IT

Paper Code \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Name of Paper – JAVA 2 EE(J2EE)

Faculty-In-Charge - MR. MANISH KHULE SIR

CERTIFICATE

THE PRACTICAL EXPERIMENTS

ENTERED IN THIS JOURNAL HAVE BEEN SATISFACTORY PERFORMED BY

ENROLLMENT NO - 2019BTCS088 MR/MS­­­­­­­­­­­­­­­­­­­­ YASH GUPTA

STUDYING IN PROGRAM B.TECH BRANCH CS&IT IN

SCHOOL OF COMPUTER SCIENCE & INFORMATION TECHNOLOGY

DURING SEMESTER 3RD OF ACADEMIC YEAR 2020-2021

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

( )

Date: 10/12/2020

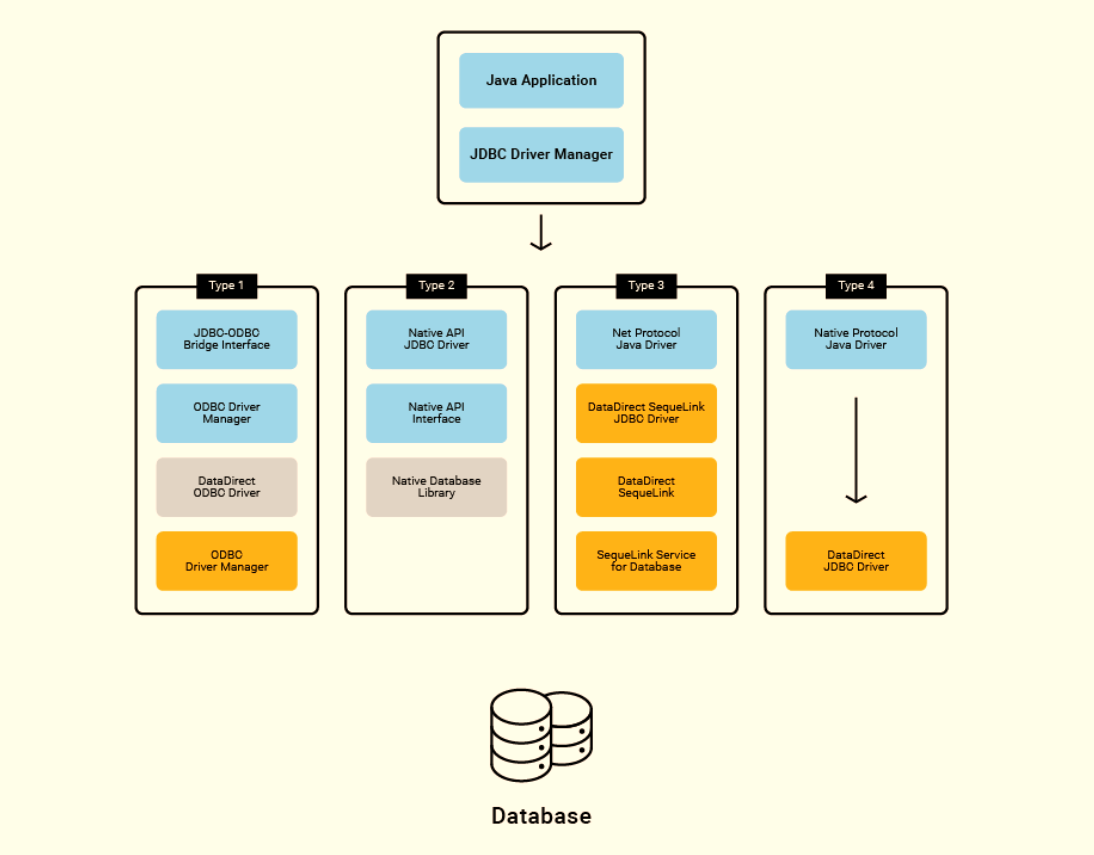
INDEX

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.NO** | **Title of Skill Activity** | **Date of Allocation** | **Date of Submission** | **Sign of Faculty** |
| 1 | Practical-01 | 05/12/2020 | 10/12/2020 |  |
| 2 | Practical-02 | 06/12/2020 | 10/12/2020 |  |
| 3 | Practical-03 | 07/12/2020 | 10/12/2020 |  |
| 4 | Practical-04 | 08/12/2020 | 10/12/2020 |  |
| 5 | Practical-05 | 09/12/2020 | 10/12/2020 |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Practical-01**

**Explain different types of JDBC Driver. Write a code to implement Driver.  
Ans: -** There are five types of JDBC drivers in use:

* **Type 1:** JDBC-ODBC bridge
* **Type 2:** partial Java driver
* **Type 3:** pure Java driver for database middleware
* **Type 4:** pure Java driver for direct-to-database
* **Type 5:** highly-functional drivers with superior performance

 **Fig: Shows the working of JDBC drivers**

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71 | //JDBC\_Drivers\_Implementation\_BY\_YASH\_GUPTA\_2019BTCS088//  //-----------------------------------------------------//  //STEP 1. Import required packages  **import** **java.sql.\***;  **public** **class** **FirstExample** {  // JDBC driver name and database URL  **static** **final** String JDBC\_DRIVER = "com.mysql.jdbc.Driver";  **static** **final** String DB\_URL = "jdbc:mysql://localhost/EMP";  // Database credentials  **static** **final** String USER = "username";  **static** **final** String PASS = "password";    **public** **static** **void** **main**(String[] args) {  Connection conn = **null**;  Statement stmt = **null**;  **try**{  //STEP 2: Register JDBC driver  Class.forName("com.mysql.jdbc.Driver");  //STEP 3: Open a connection  System.out.println("Connecting to database...");  conn = DriverManager.getConnection(DB\_URL,USER,PASS);  //STEP 4: Execute a query  System.out.println("Creating statement...");  stmt = conn.createStatement();  String sql;  sql = "SELECT id, first, last, age FROM Employees";  ResultSet rs = stmt.executeQuery(sql);  //STEP 5: Extract data from result set  **while**(rs.next()){  //Retrieve by column name  **int** id = rs.getInt("id");  **int** age = rs.getInt("age");  String first = rs.getString("first");  String last = rs.getString("last");  //Display values  System.out.print("ID: " + id);  System.out.print(", Age: " + age);  System.out.print(", First: " + first);  System.out.println(", Last: " + last);  }  //STEP 6: Clean-up environment  rs.close();  stmt.close();  conn.close();  }**catch**(SQLException se){  //Handle errors for JDBC  se.printStackTrace();  }**catch**(Exception e){  //Handle errors for Class.forName  e.printStackTrace();  }**finally**{  //finally block used to close resources  **try**{  **if**(stmt!=**null**)  stmt.close();  }**catch**(SQLException se2){  }// nothing we can do  **try**{  **if**(conn!=**null**)  conn.close();  }**catch**(SQLException se){  se.printStackTrace();  }//end finally try  }//end try  System.out.println("Goodbye!");  }//end main  }//end FirstExample | |

**----------<<<THE END>>>----------**

**Practical-02**

Q2. Explain Hibernate Architecture. Write a code to implement Hibernate. **Ans: -** Hibernate has a layered architecture which helps the user to operate without having to know the underlying APIs. Hibernate makes use of the database and configuration data to provide persistence services (and persistent objects) to the application.

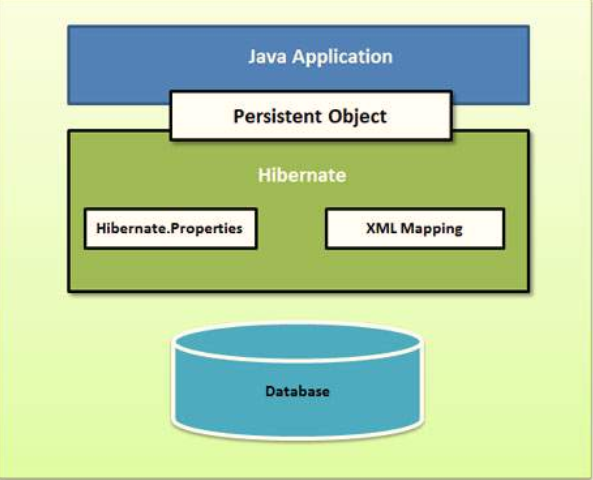


Fig: High-level view of the Hibernate Application Architecture

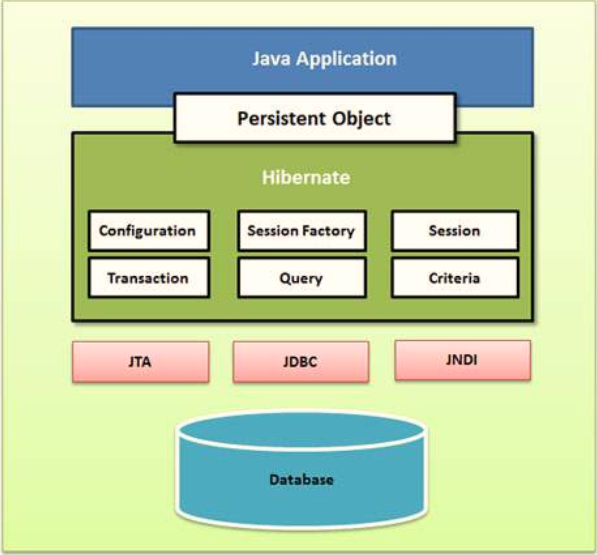


Fig: Hibernate Application Architecture

**Working of Hibernate**

Hibernate uses various existing Java APIs, like JDBC, Java Transaction API(JTA), and Java Naming and Directory Interface (JNDI). JDBC provides a rudimentary level of abstraction of functionality common to relational databases, allowing almost any database with a JDBC driver to be supported by Hibernate. JNDI and JTA allow Hibernate to be integrated with J2EE application servers.

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

**Configuration Object**

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

The Configuration object provides two keys components −

* **Database Connection** − This is handled through one or more configuration files supported by Hibernate. These files are **hibernate.properties** and **hibernate.cfg.xml**.
* **Class Mapping Setup** − This component creates the connection between the Java classes and database tables.

**SessionFactory Object**

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

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

**Session Object**

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

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

**Transaction Object**

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

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

**Query Object**

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

**Criteria Object**

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

**Code Implementation**

Let’s create our step by step ***hibernate 5 hello world examples***. In this example, I have created an **Employee** class and declared four attributes **id, email, firstname** and **lastname.**

The files that need to be created are as follows:- o

1. **hibernate.cfg.xml** -This configuration file will be used to store database connection information and schema level settings.
2. **EmployeeEntity.java** – This class will refer Java POJOs having hibernate annotations.
3. **HibernateUtil.java** – This class will have utility methods which will be used for creating session factory and session objects.
4. **TestHibernate.java** – This class will be used to test our configuration settings and Employee entity annotations.

**Starting our Project:**

#### Make project to support eclipse

|  |
| --- |
| $ mvn **eclipse:**eclipse |

#### Hibernate Maven Dependencies(pom.xml)

At minimum, we will need **hibernate-core** dependency. We are using in-memory database H2 for this example. So, include com.h2database dependency as well.

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | //Import java project to eclipse workspace  pom.xml  &lt;project xmlns=&quot;**http:**//maven.apache.org/POM/4.0.0&quot;  **xmlns:**xsi=&quot;**http:**//www.w3.org/2001/XMLSchema-instance&quot;  **xsi:**schemaLocation=&quot;**http:**//maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd&quot;&gt;  &lt;modelVersion&gt;**4.0**.**0**&lt;/modelVersion&gt;  &lt;groupId&gt;com.howtodoinjava&lt;/groupId&gt;  &lt;artifactId&gt;hibernate-hello-world&lt;/artifactId&gt;  &lt;version&gt;**0.0**.**1**-SNAPSHOT&lt;/version&gt;  &lt;dependencies&gt;  &lt;dependency&gt;  &lt;groupId&gt;org.hibernate&lt;/groupId&gt;  &lt;artifactId&gt;hibernate-core&lt;/artifactId&gt;  &lt;version&gt;**5.3**.**7**.Final&lt;/version&gt;  &lt;/dependency&gt;  &lt;dependency&gt;  &lt;groupId&gt;com.h2database&lt;/groupId&gt;  &lt;artifactId&gt;h2&lt;/artifactId&gt;  &lt;version&gt;**1.4**.**200**&lt;/version&gt;  &lt;/dependency&gt;  &lt;dependency&gt;  &lt;groupId&gt;junit&lt;/groupId&gt;  &lt;artifactId&gt;junit&lt;/artifactId&gt;  &lt;version&gt;**4.12**&lt;/version&gt;  &lt;scope&gt;test&lt;/scope&gt;  &lt;/dependency&gt;  &lt;/dependencies&gt;  &lt;build&gt;  &lt;sourceDirectory&gt;src/main/java&lt;/sourceDirectory&gt;  &lt;plugins&gt;  &lt;plugin&gt;  &lt;artifactId&gt;maven-compiler-plugin&lt;/artifactId&gt;  &lt;version&gt;**3.5**.**1**&lt;/version&gt;  &lt;configuration&gt;  &lt;source&gt;**1.8**&lt;/source&gt;  &lt;target&gt;**1.8**&lt;/target&gt;  &lt;/configuration&gt;  &lt;/plugin&gt;  &lt;/plugins&gt;  &lt;/build&gt;  &lt;/project&gt; | |

#### Hibernate Configuration(hibernate.cfg.xml)

Notice the connection parameters use the H2 database related properties.

|  |
| --- |
|  |
|  | &lt;?xml version=&quot;**1.0**&quot; encoding=&quot;utf-**8**&quot;?&gt;  &lt;!DOCTYPE hibernate-configuration PUBLIC  &quot;-//Hibernate/Hibernate Configuration DTD 3.0//EN&quot;  &quot;**http:**//hibernate.sourceforge.net/hibernate-configuration-3.0.dtd&quot;&gt;  &lt;hibernate-configuration&gt;  &lt;session-factory&gt;  &lt;property name=&quot;hibernate.connection.driver\_class&quot;&gt;org.h2.Driver&lt;/property&gt;  &lt;property name=&quot;hibernate.connection.url&quot;&gt;**jdbc:h2:mem:**test&lt;/property&gt;  &lt;property name=&quot;hibernate.connection.username&quot;&gt;sa&lt;/property&gt;  &lt;property name=&quot;hibernate.connection.password&quot;&gt;&lt;/property&gt;  &lt;property name=&quot;hibernate.dialect&quot;&gt;org.hibernate.dialect.H2Dialect&lt;/property&gt;  &lt;property name=&quot;show\_sql&quot;&gt;**true**&lt;/property&gt;  &lt;property name=&quot;hbm2ddl.auto&quot;&gt;create-drop&lt;/property&gt;  &lt;mapping class=&quot;com.howtodoinjava.hibernate.test.dto.EmployeeEntity&quot;&gt;&lt;/mapping&gt;  &lt;/session-factory&gt;  &lt;/hibernate-configuration&gt; |

#### Entity class(EmployeeEntity.java)

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | **package** hibernate.test.dto;  **import** **java.io.Serializable**;    **import** **javax.persistence.Column**;  **import** **javax.persistence.Entity**;  **import** **javax.persistence.GeneratedValue**;  **import** **javax.persistence.GenerationType**;  **import** **javax.persistence.Id**;  **import** **javax.persistence.Table**;  **import** **javax.persistence.UniqueConstraint**;    **import** **org.hibernate.annotations.OptimisticLockType**;    **@Entity**  **@org.hibernate.annotations.Entity**(optimisticLock = OptimisticLockType.ALL)  **@Table**(name = &quot;Employee&quot;, uniqueConstraints = {  **@UniqueConstraint**(columnNames = &quot;ID&quot;),  **@UniqueConstraint**(columnNames = &quot;EMAIL&quot;) })  **public** **class** **EmployeeEntity** **implements** Serializable {    **private** **static** **final** **long** serialVersionUID = -**1798070786993154676L**;    **@Id**  **@GeneratedValue**(strategy = GenerationType.IDENTITY)  **@Column**(name = &quot;ID&quot;, unique = **true**, nullable = **false**)  **private** Integer employeeId;    **@Column**(name = &quot;EMAIL&quot;, unique = **true**, nullable = **false**, length = **100**)  **private** String email;    **@Column**(name = &quot;FIRST\_NAME&quot;, unique = **false**, nullable = **false**, length = **100**)  **private** String firstName;    **@Column**(name = &quot;LAST\_NAME&quot;, unique = **false**, nullable = **false**, length = **100**)  **private** String lastName;    // Accessors and mutators for all four fields  } | |

#### Hibernate Session Factory

This is important. [org.hibernate.cfg.Configuration](https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/cfg/Configuration.html) is used for bootstrapping Hibernate.

Here, we define all the configuration options and entity definitions in **hibernate.cgf.xml** and build the SessionFactory in single statement.

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | **package** com.howtodoinjava.hibernate.test;  **import** **java.io.File**;  **import** **org.hibernate.SessionFactory**;  **import** **org.hibernate.cfg.Configuration**;    **public** **class** **HibernateUtil** {  **private** **static** **final** SessionFactory sessionFactory = buildSessionFactory();    **private** **static** SessionFactory **buildSessionFactory**() {  **try** {  // Create the SessionFactory from hibernate.cfg.xml  **return** **new** **Configuration**().configure(**new** File(&quot;hibernate.cgf.xml&quot;)).buildSessionFactory();    }  **catch** (Throwable ex) {  // Make sure you log the exception, as it might be swallowed  System.err.println(&quot;Initial SessionFactory creation failed.&quot; + ex);  **throw** **new** **ExceptionInInitializerError**(ex);  }  }    **public** **static** SessionFactory **getSessionFactory**() {  **return** sessionFactory;  }    **public** **static** **void** **shutdown**() {  // Close caches and connection pools  getSessionFactory().close();  }  } | |

#### Demo

*TestHibernate.java*

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | **package** com.howtodoinjava.hibernate.test;    **import** **org.hibernate.Session**;    **import** **com.howtodoinjava.hibernate.test.dto.EmployeeEntity**;    **public** **class** **TestHibernate** {    **public** **static** **void** **main**(String[] args) {  Session session = HibernateUtil.getSessionFactory().openSession();  session.beginTransaction();    //Add new Employee object  EmployeeEntity emp = **new** EmployeeEntity();  emp.setEmail(&quot;demo-user**@mail.com**&quot;);  emp.setFirstName(&quot;demo&quot;);  emp.setLastName(&quot;user&quot;);    session.save(emp);    session.getTransaction().commit();  HibernateUtil.shutdown();  }  } | |

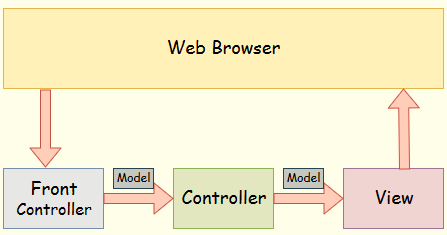
The above code will create a new table **employee** in the database and insert one row in this table. If you want to **verify** the **insert statement** which got executed, we can easily verify in **LOGS**.

**----------<<<THE END>>>----------**

**Practical-03**

Q3. Define Spring MVC Framework with block diagram. **Ans: -** 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.



## Fig: Spring Web Model-View-Controller

* **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.

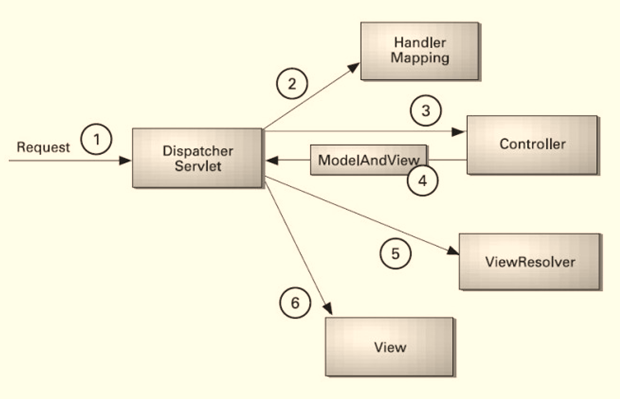


Fig: Flow of Spring MVC

## 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.

## Spring Web MVC Framework Examples:

* 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 without using Maven | Directory Structure of Spring MVC with using Maven |

**----------<<<THE END>>>----------**

**Practical-04**

Q4. Describe different types of JSP Directives. Write a code to implement JSP directives. **Ans: -** A JSP directive affects the overall structure of the servlet class. It usually has the following form −

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1 | <%@ directive attribute = "value" %> | |

Directives can have a number of attributes which you can list down as key-value pairs and separated by commas.

The blanks between the ***@*** symbol and the directive name, and between the last attribute and the closing ***%>***, **are optional.**

There are three types of directive tag −

|  |  |
| --- | --- |
| S.No. | Directive & Description |
| 1 | **<%@ page ... %>**  Defines page-dependent attributes, such as scripting language, error page, and buffering requirements. |
| 2 | **<%@ include ... %>**  Includes a file during the translation phase. |
| 3 | **<%@ taglib ... %>**  Declares a tag library, containing custom actions, used in the page |

Code to implement JSP directives:

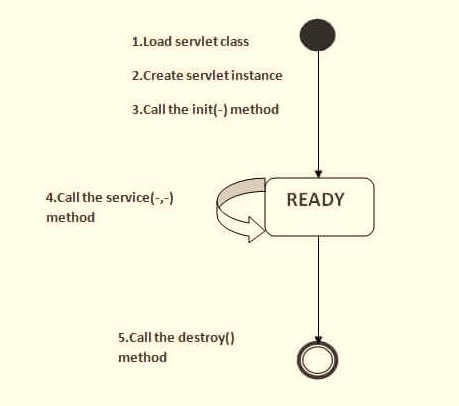
|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9 | <%@ page import="java.io.PrintWriter"%>  <%@ page isErrorPage="true"%>    An error is **reported:**<br/>  <i><%= exception %></i><br/>  This problem occurred in the following **place:**<br/>  <pre>  <% exception.printStackTrace( **new** PrintWriter( out ) ); %>  </pre> | |

**----------<<<THE END>>>----------**

**Practical-05**

Q5. Explain the life cycle of servlet. Write a code to implement servlet. **Ans: -** The web container maintains the life cycle of a servlet instance. Let's see the life cycle of the servlet:

1. **Servlet** **class** is loaded.
2. **Servlet** **instance** is created.
3. **init** **method** is invoked.
4. **service** **method** is invoked.
5. **destroy** **method** is invoked.



### **Servlet class is loaded**

The **classloader** is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.

### **Servlet instance is created**

The web container creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.

### **init method is invoked**

|  |
| --- |
| The web container calls the init method only once after creating the servlet instance. The init method is used to initialize the servlet. It is the life cycle method of the javax.servlet.Servlet interface. Syntax of the init method is given below: |

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1 | **public** **void** **init**(ServletConfig config) **throws** ServletException | |

### **service method is invoked**

The web container calls the service method each time when request for the servlet is received. If servlet is not initialized, it follows the first three steps as described above then calls the service method. If servlet is initialized, it calls the service method. Notice that servlet is initialized only once. The syntax of the service method of the Servlet interface is given below:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2 | **public** **void** **service**(ServletRequest request, ServletResponse response)  **throws** ServletException, IOException | |

### **destroy method is invoked**

The web container calls the destroy method before removing the servlet instance from the service. It gives the servlet an opportunity to clean up any resource for example memory, thread etc. The syntax of the destroy method of the Servlet interface is given below:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1 | **public** **void** **destroy**() | |

**----------<<<THE END>>>----------**