**Project**

On

**DA3M.com(LUXURY WATCHES) E-Commerce Website**

Submitted towards Digital Transformation Training NIIT DELHI

**Done By**

AVINASH PALTANI

NIIT PREET VIHAR

(DELHI)

Contents

[Introduction 4](#_Toc450894923)

[Objective 4](#_Toc450894925)

[Hardware and Software Requirements 5](#_Toc450894927)

[Software Installation 5](#_Toc450894928)

[Getting Started with Eclipse IDE 6](#_Toc450894929)

[Getting Started with H2 Database 6](#_Toc450894930)

[Getting Started with Maven 7](#_Toc450894931)

[Configuring Tomcat8 server into Eclipse 8](#_Toc450894932)

[Getting Started with GITHUB 8](#_Toc450894933)

[Creating Maven Project on Eclipse 9](#_Toc450894934)

[Adding Dependencies: 11](#_Toc450894935)

[Adding Boot Strap 12](#_Toc450894936)

[Views and Resolving them 14](#_Toc450894939)

[Adding the Front Controller and SPRING MVC Framework 15](#_Toc450894940)

[Spring 3 MVC Basic Architecture: 16](#_Toc450894941)

[Spring 3 MVC Request Flow: 17](#_Toc450894942)

[Required Spring 3.0 MVC Configuration: 18](#_Toc450894943)

[Defining a Controller 20](#_Toc450894944)

[Spring MVC Form Model Implementation 21](#_Toc450894945)

[CRUD Operations Spring 3 Hibernate Integration 23](#_Toc450894946)

[Create DAO Class 26](#_Toc450894947)

[Added Multi-Part 27](#_Toc450894948)

[Implementing Spring Security 28](#_Toc450894949)

[Implementing Web flow 29](#_Toc450894950)

**Introduction**

In today's wired age, it is common knowledge that setting up one's own web page is not a particularly difficult endeavor. Indeed, anyone who has run a broad-based search through one of the major search engines has likely come across myriad "homemade" pages created by individuals reflecting their personal interests or some life ambition. To set up a web site, one really only needs an Internet-connected computer, a web browser equipped with a basic text-editing application and an Internet service provider (ISP) that offers web hosting for its users. Such users are unlikely to seek legal advice, and the legal issues that arise in relation to such sites tend to be limited to copyright and trademark violations by the site creators.

E-commerce, however, presents a wholly different challenge for the site creator and the legal practitioner. The sophisticated nature of the technology required, the number of players involved in setting up a site and facilitating transactions, the privacy concerns of customers who may be giving the site information about themselves, and a host of other realities of online business make effective legal representation critical in this arena. In addition, e-commerce set-up often requires great speed due to the nature of the industry, therefore adding an additional layer of complexity that calls for even more vigilance and preparedness on the part of the legal practitioners in structuring transactions and advising clients.

**Objective**

The main objective of the project is to design, develop and validate the concept of following:

* Perceive and implement object-oriented concepts using Java technology programs
* Write SQL queries to retrieve, manage, and manipulate data
* Design responsive Web/enterprise apps using HTML5, CSS3, and various JavaScript frameworks such as Bootstrap & AngularJS
* Build persistent and loosely coupled Web apps using Hibernate and Spring technologies
* Build Slumbrous Web Services
* Build high quality Web/enterprise apps by using DevOps platform

**Hardware and Software Requirements**

To complete the given project we will need the following hardware and software requirements

Hardware Requirements:

* Inter l-Core i3 processer
* 4 GB RAM
* System type : 64bit
* Hard Disk Space Required 20GB Minimum
* Internet Connection with 1 MBPS Recommended

Software Requirements:

* Operating System Windows 7, 8, 10
* Eclipse Mars IDE
* Apache tomcat 8 server
* Java Development Kit version 8
* H2 Database
* Bootstrap Scripts
* Angular JS Scripts
* GitHub

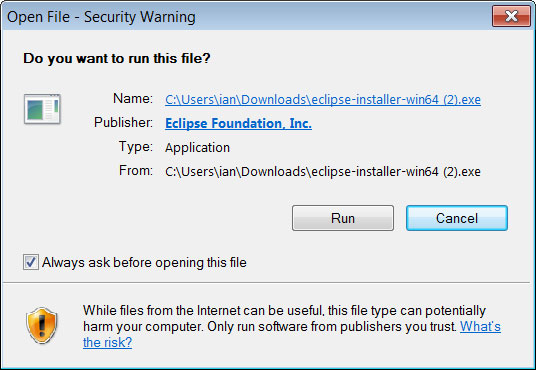
**Software Installation**

Getting Started with Eclipse IDE

**Steps to Install Eclipse**

* For this project we will be installing the Eclipse Mars release, we are introducing a new Eclipse installer. This is a new and more efficient way to install Eclipse. It is a proper installer, so no more zip files, with a self-extracting download that will lead you through the installation experience. For those not into installers, we still have the packages and zip files available on our download pages.
* Download the Eclipse Installer windows 64bit (Recommended) or Windows 32bit, from <https://eclipse.org/downloads/index.php>
* Start the Eclipse Installer executable For Windows users, after the Eclipse Installer executable has finished downloading it should be available in your download directory. Start the Eclipse Installer executable. You may get a security warning to run this file. If the Eclipse Foundation is the Publisher, you are good to select **Run**.

Installation Video: <https://www.youtube.com/watch?v=35NUuhmQuB4>



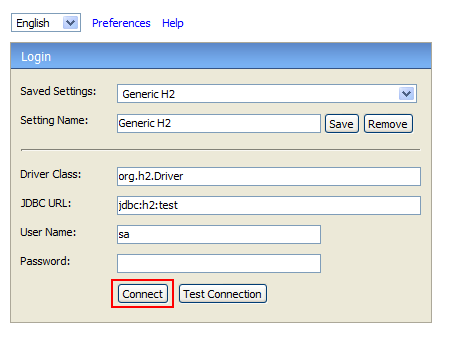
**Getting Starting with H2 Database**

**Requirements:** To run this database, the following software stack is known to work. Other software most likely also works, but is not tested as much.

**Database Engine:** Windows XP / Vista / Windows 7 / Windows 8 and above, Mac OS X, or Linux Sun Java 6 or newer Recommended Windows file system: NTFS (FAT32 only supports files up to 4 GB).

**Installing the Software:** To install the software, run the installer or unzip it to a directory of your choice. You can download the installer from <http://www.h2database.com/html/download.html>.

Installation Video link: <https://www.youtube.com/watch?v=1BkvXEv65Z8>



**Getting Started with Maven**

**Prerequisites**

You must have an understanding of how to install software on your computer.

**Installation**

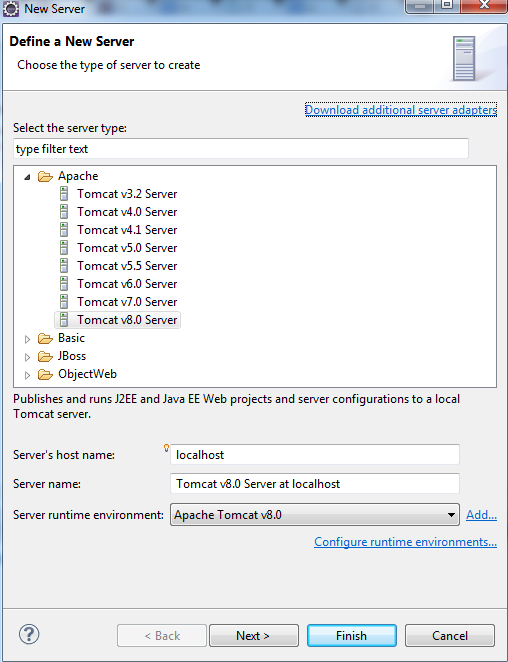
Maven is a Java tool, so you must have Java installed in order to proceed.

**Downloading Apache Maven 3.3.9**

Apache Maven 3.3.9 is the latest release and recommended version for all users. Which can be downloaded from <http://maven.apache.org/download.cgi>.

**Configuring Tomcat8 server into Eclipse**

* Download Tomcat 8 from https://tomcat.apache.org/download-80.cgi and place it within any local folder.
* Start Eclipse and click on “Servers” tab in the workbench. Go ahead and try adding a new server. You would find option for Tomcat 8 available for selection as shown below.
* After clicking Finish, you would see a new server added with the name as “Tomcat v8.0 Server at localhost”. Start the server.
* Video Url: <https://www.youtube.com/watch?v=kLgquZ2FiuQ>



**Getting Started with GITHUB**

GitHub offers free accounts for users and organizations working on public and open source projects, as well as paid accounts that offer unlimited private repositories.

**Signing up for service**

* Go to GitHub's home page. [www.github.com](http://www.github.com)
* Read the information about the different accounts GitHub offers and decide which type of account you'd like to create, then click Join GitHub for free or Upgrade your account.
* Under "Create your personal account," type your username, email address, and password, then click Create an account.
* Select your plan type. If you're unsure about what you need, you can just select the Free account type.
* Click Finish sign up.

Video URL: <https://www.youtube.com/watch?v=ezxRcdJ8glM>

**Creating Maven Project on Eclipse**

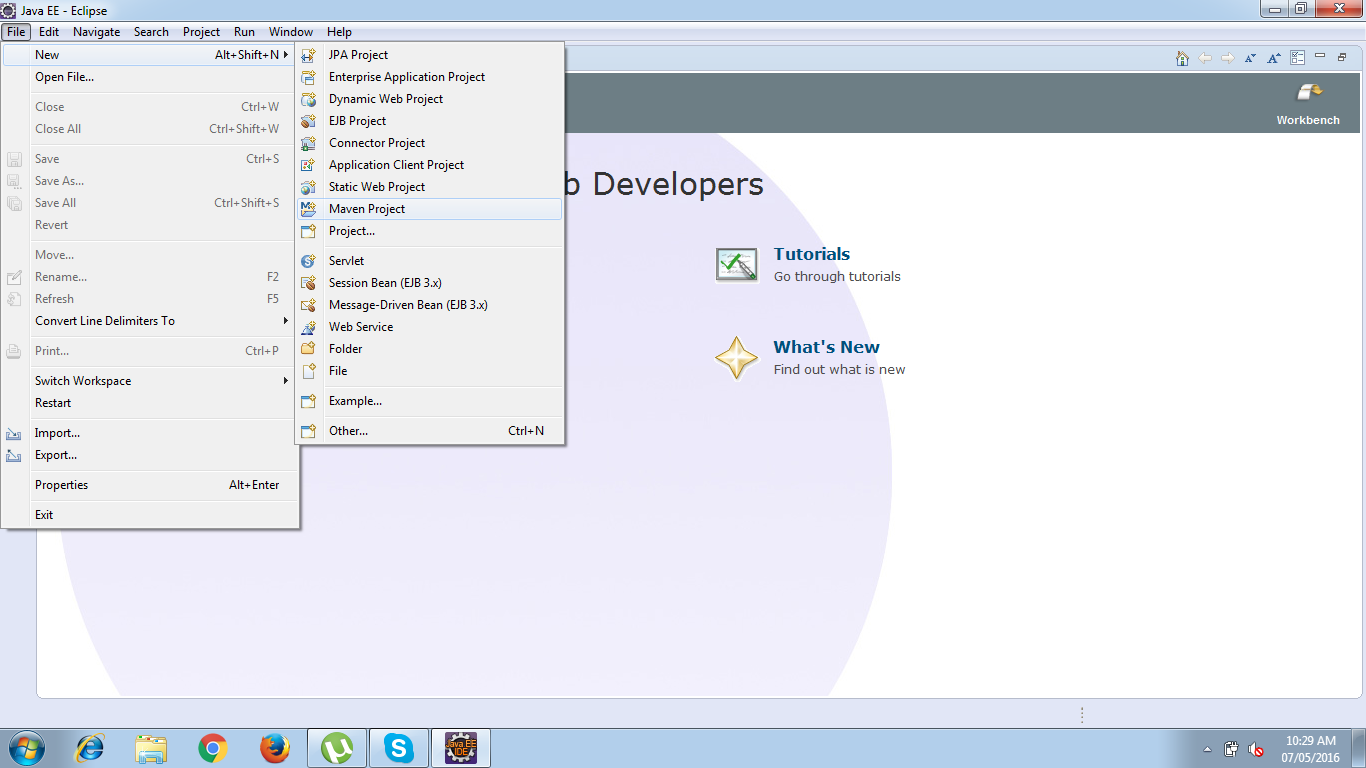
**Prerequisite**

You will need to install the Eclipse IDE Maven plugin found at the Eclipse Marketplace.

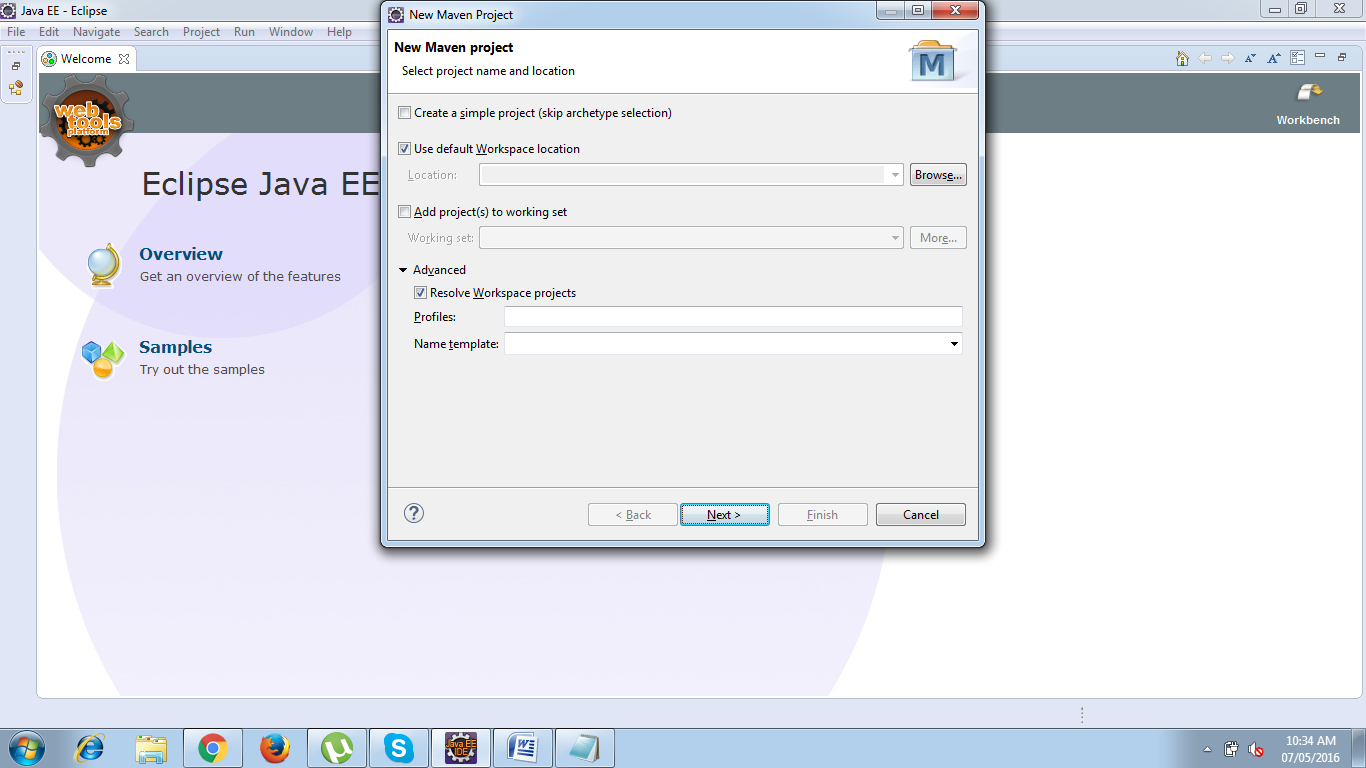
Video URL: <https://www.youtube.com/watch?v=YeC7XQho-O0>

**Step 1.** Create a New Maven Project

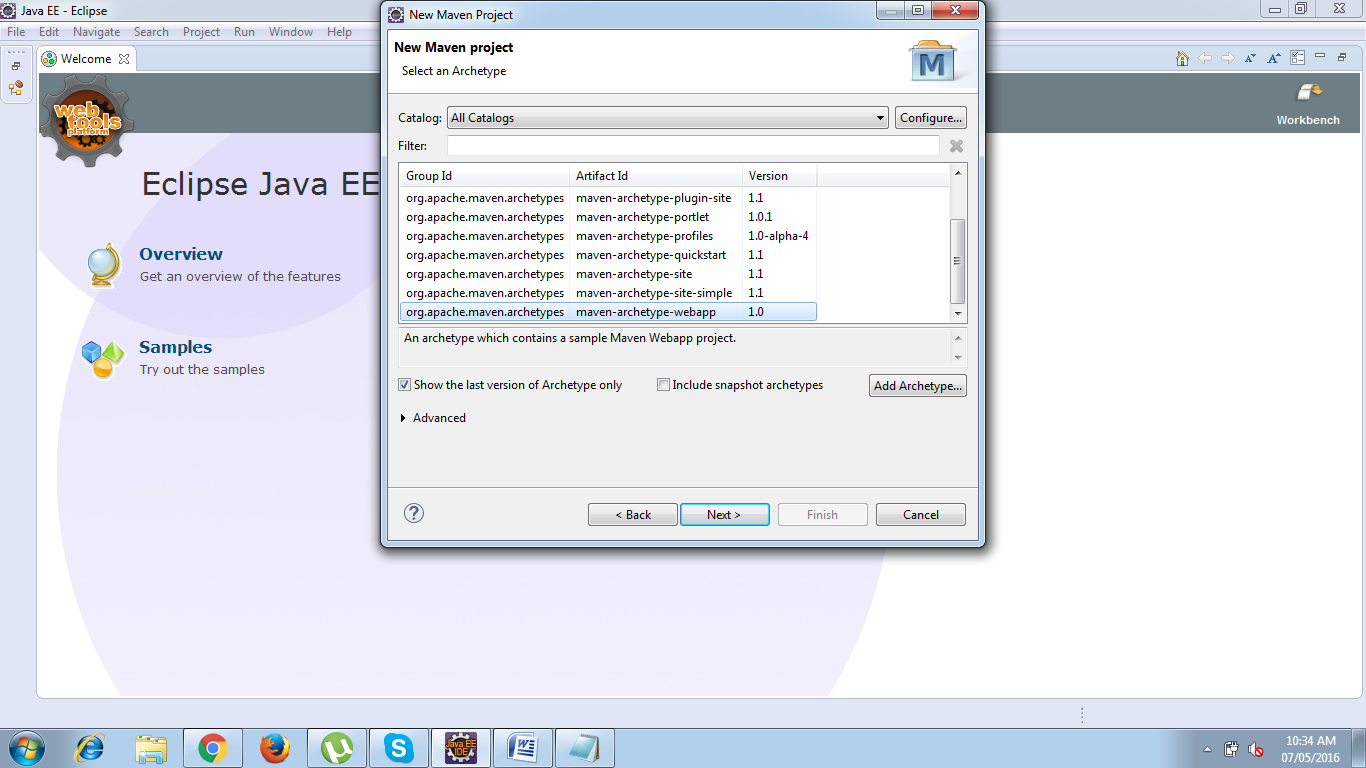
Click 'File' -> 'New' -> 'Other' -> 'Maven Project' and then click 'Next'.



**Step 2.** Select project name and location.



**Step 3.** Select an archetype



**Adding Dependencies**

**What is the POM?**

POM stands for "Project Object Model". It is an XML representation of a Maven project held in a file named pom.xml. When in the presence of Maven folks, speaking of a project is speaking in the philosophical sense, beyond a mere collection of files containing code. A project contains configuration files, as well as the developers involved and the roles they play, the defect tracking system, the organization and licenses, the URL of where the project lives, the project's dependencies, and all of the other little pieces that come into play to give code life. It is a one-stop-shop for all things concerning the project. In fact, in the Maven world, a project need not contain any code at all, merely a pom.xml.

The POM contains all necessary information about a project, as well as configurations of plugins to be used during the build process. It is, effectively, the declarative manifestation of the "who", "what", and "where", while the build lifecycle is the "when" and "how".

**Maven Coordinates**

“ groupId:artifactId:version “ are all required fields (although, groupId and version need not be explicitly defined if they are inherited from a parent - more on inheritance later). The three fields act much like an address and timestamp in one. This marks a specific place in a repository, acting like a coordinate system for Maven projects.

**groupId**: This is generally unique amongst an organization or a project. For example, all core Maven artifacts do (well, should) live under the groupId org.apache.maven. Group ID's do not necessarily use the dot notation, for example, the junit project. Note that the dot-notated groupId does not have to correspond to the package structure that the project contains. It is, however, a good practice to follow. When stored within a repository, the group acts much like the Java packaging structure does in an operating system.

**artifactId**: The artifactId is generally the name that the project is known by. Although the groupId is important, people within the group will rarely mention the groupId in discussion. It, along with the groupId, create a key that separates this project from every other project in the world. Along with the groupId, the artifactId fully defines the artifact's living quarters within the repository

**version**: This is the last piece of the naming puzzle. groupId:artifactId denote a single project but they cannot delineate which incarnation of that project we are talking about. Do we want the junit:junit of today (version 4), or of four years ago (version 2)? In short: code changes, those changes should be versioned, and this element keeps those versions in line. It is also used within an artifact's repository to separate versions from each other.

The three elements given above point to a specific version of a project letting Maven knows who we are dealing with, and when in its software lifecycle we want them.

Video Url: <https://www.youtube.com/watch?v=XuoC_N3wvFc>

Web Ref: <https://maven.apache.org/pom.html>

**Adding Boot Strap**

Bootstrap is one of the most popular front-end framework for developing responsive web design. It includes buttons, form inputs, links, columns, and tons of other pre-formatted page objects. The mixt of Bootstrap and Spring MVC gives a powerfull toolbox to develop a web application running in both desktop and mobile devices.

* You can add Bootstrap to your project by downloading a zip file from the project web site <http://getbootstrap.com> , then unzip the content and copy it to the webapp resources directory.

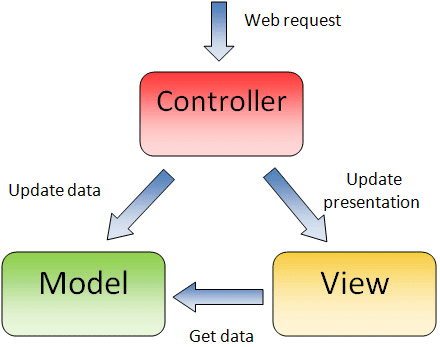
**Add a bootstrap reference to the web page**

<script type="text/javascript" src="webjars/jquery/2.1.1/jquery.min.js"></script>

<script type="text/javascript" src="webjars/bootstrap/3.2.0/js/bootstrap.min.js"></script>

**MAVEN MVC**

[**Model-view-controller (MVC)**](http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller) is a well-known [design pattern](http://howtodoinjava.com/category/design-patterns/) for designing UI based applications. It mainly decouples business logic from UIs by separating the roles of **model, view, and controller** in an application. Usually, models are responsible for encapsulating application data for views to present. Views should only present this data, without including any business logic. And controllers are responsible for receiving requests from users and invoking back-end services (manager or Dao) for business logic processing. After processing, back-end services may return some data for views to present. Controllers collect this data and prepare models for views to present. The core idea of the MVC pattern is to separate business logic from UIs to allow them to change independently without affecting each other.

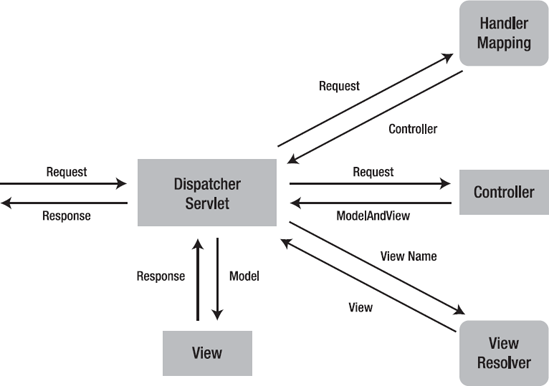
****

In a Spring MVC application, models usually consist of POJO objects that are processed by the service layer and persisted by the persistence layer. Views are usually JSP templates written with [Java Standard Tag Library (JSTL)](https://jstl.java.net/). Controller part is played by dispatcher servlet which we will learn about in this tutorial in more detail.

**Dispatcher Servlet (Spring Controller)**

In the simplest Spring MVC application, a controller is the only servlet you need to configure in a Java web deployment descriptor (i.e., the web.xml file). A Spring MVC controller—often referred to as a [Dispatcher Servlet](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/DispatcherServlet.html) implements [front controller](https://en.wikipedia.org/wiki/Front_Controller_pattern) design pattern and every web request must go through it so that it can manage the entire request life cycle.

When a web request is sent to a Spring MVC application, dispatcher servlet first receives the request. Then it organizes the different components configured in Spring’s web application context (e.g. actual request handler controller and view resolvers) or annotations present in the controller itself, all needed to handle the request.



**Spring Dispatcher Servlet**

To define a controller class in Spring 3.0, a class has to be marked with the @Controller annotation. When a @Controller annotated controller receives a request, it looks for an appropriate handler method to handle the request. This requires that a controller class map each request to a handler method by one or more handler mappings. In order to do so, a controller class’s methods are decorated with the @Request Mapping annotation, making them handler methods.

**Views and Resolving them**

All MVC frameworks for web applications provide a way to address views. Spring provides view resolvers, which enable you to render models in a browser without tying you to a specific view technology.

**Resolving views - the View Resolver interface**

As an example, when using JSP for a view technology you can use the Url Based View Resolver. This view resolver translates a view name to a URL and hands the request over to the Request Dispatcher to render the view.

<mvc:resources location=*"/resources/"* mapping=*"/resources/\*\*"* />

<context:component-scan base-package=*"com.\*"* />

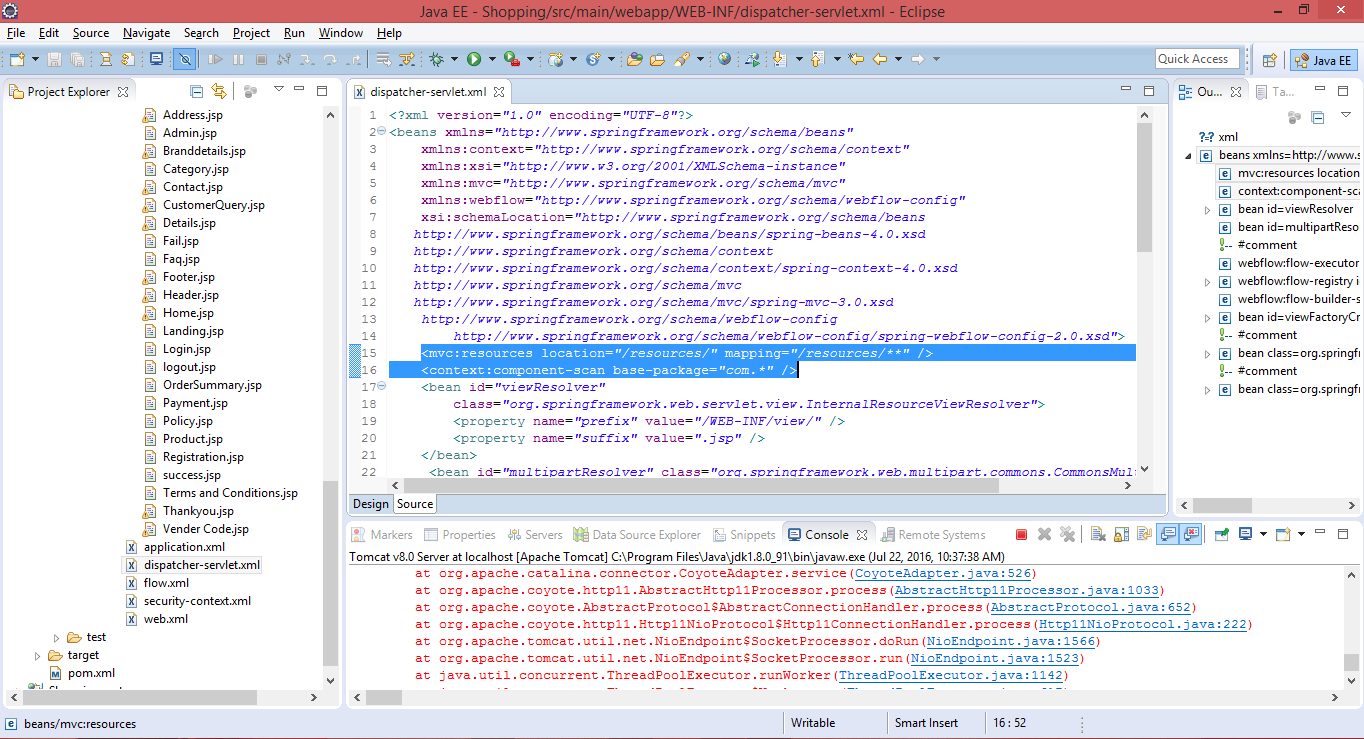
<bean id=*"viewResolver"*

class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<property name=*"prefix"* value=*"/WEB-INF/view/"* />

<property name=*"suffix"* value=*".jsp"* />

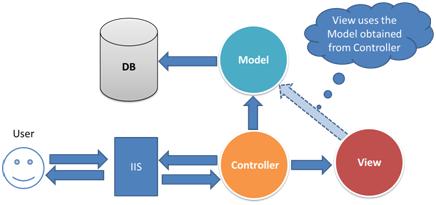
</bean>



**Adding the Front Controller and SPRING MVC Framework**

Model view controller is a software architecture design pattern. It provides solution to layer an application by separating three concerns business, presentation and control flow.

* The Model can be some DAO layer or some Service Layers which give some information about request or requested information or Model can be a POJO which encapsulates the application data given by the controller.
* The View is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
* The Controller is responsible for processing user requests and building appropriate model and passes it to the view for rendering.

****

**Basic Flow of SPRING MVC**

**Advantages of Spring MVC Framework-**

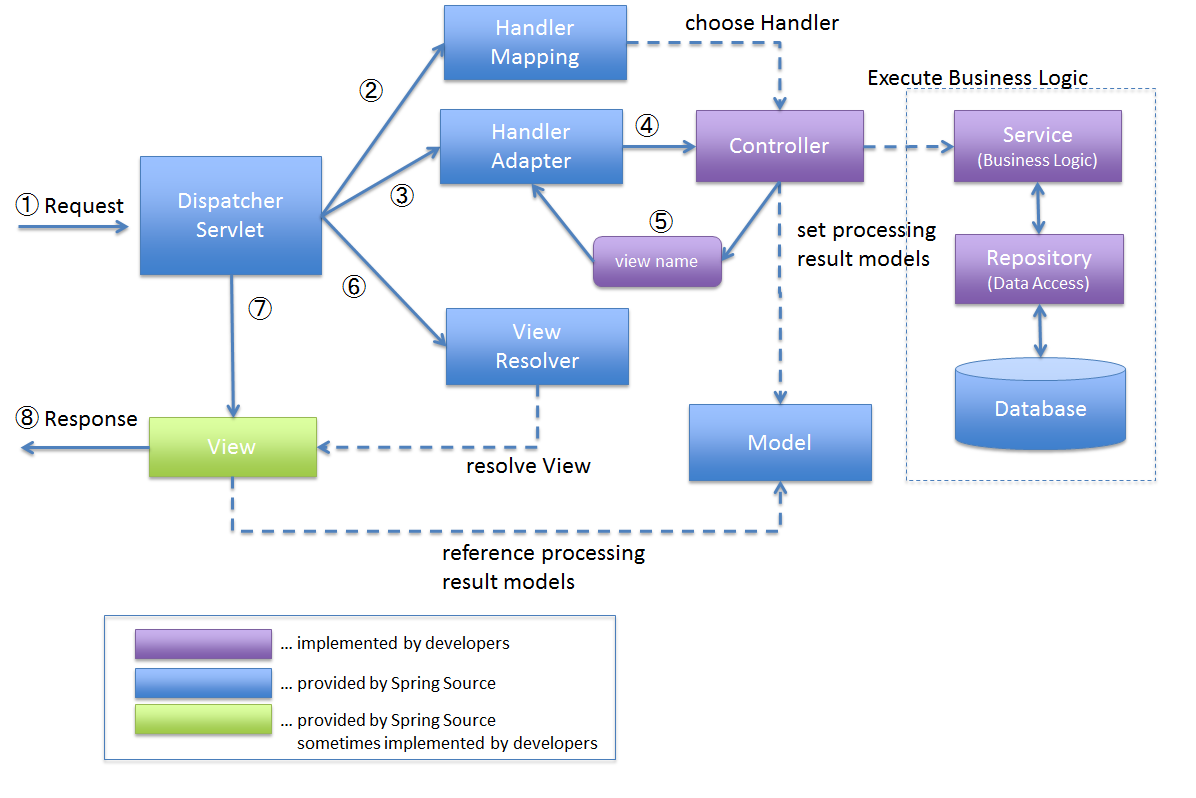
* Supports RESTful URLs.
* Annotation based configuration (i.e. you may reduce the metadata file or less of configuration).
* Supports to plug with other MVC frameworks like Struts, Struts2 and Web Works etc.
* Flexible in supporting different view types like JSP, velocity, XML, PDF, Tiles etc.

**Front Controller:** Front Controller is very important component one which route the all the requests into framework control that means whenever requests land on different controllers it queues that request to the controller of framework without this MVC framework will not may be able to take control of the request at landing at the application. So front controller is not only capture the request but also the following responsibility-

* It initialize the framework to cater to the requests.
* Load the map of all URLs and the components responsible to handle the request.
* Prepare the map for the views.

**Spring 3 MVC Basic Architecture**

The spring web MVC framework provides model-view-controller architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.



**Spring 3 MVC Architecture Diagram**

**Spring 3 MVC Request Flow**

1. Request lands to Front Controller i.e. Dispatcher Servlet

2. Capture the Request Locale i.e. use for internationalization i.e. Read .properties files

3. Check for multipart-file (MIME type header or not) upload data from distributed application

4. Consult with Handler Mapping for which Controller to be invoked

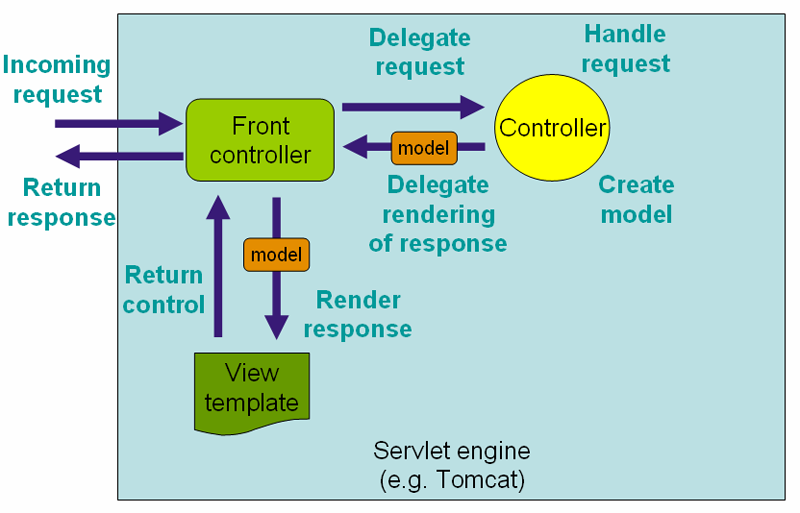
5. And then responsibility is given to the Handler Chain

6. This Handler Chain is responsible to be invoked some of the interceptors that needs to be invoked before of a controller and after the controller that means interceptors are here like very too similar to the filters that help to separate the pre-process logic and post-process logic.

7. After process of pre-process interceptor return to the controller process the post-process logic.

8. Then return to the view resolver prepared the view based on your configuration decide the configuration (JSP, Velocity, PDF etc.) to be invoked.

9. After choosing view technology prepare the view and return the response back to the client.

****

**Required Spring 3 MVC Configuration:**

You need to map requests that you want the Dispatcher Servlet to handle, by using a URL mapping in the web.xml file. The following is an example to show declaration and mapping for spring3 Dispatcher Servlet example:

**Step 1** Configure the web.xml with Dispatcher Servlet and details of the application context file location.

<web-app id="WebApp\_ID" version="2.4" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/j2ee" xsi:schemalocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd">

<servlet>

<servlet-name>spring3</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>spring3</servlet-name>

<url-pattern>\*.\*</url-pattern>

</servlet-mapping>

**Step 2** Configure the contextConfigLocation for the application context to be loaded.

<web-app id="WebApp\_ID" version="2.4" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/j2ee" xsi:schemalocation="http://java.sun.com/xml/ns/j2ee

http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd">

<display-name>Spring MVC Application</display-name>

<servlet>

<servlet-name>Dispatcher</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>default</servlet-name>

<url-pattern>\*.\*</url-pattern>

</servlet-mapping>

<context-param>

<param-name>contextConfigLocation</param-name><param-value>/WEB-INF/spring3-servlet.xml</param-value></context-param>

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

</web-app>

**Step 3:** Configure the spring3-servlet.xml

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

xmlns:webflow=*"http://www.springframework.org/schema/webflow-config"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-4.0.xsd*

*http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd*

*http://www.springframework.org/schema/webflow-config*

*http://www.springframework.org/schema/webflow-config/spring-webflow-config-2.0.xsd"*>

<mvc:resources location=*"/resources/"* mapping=*"/resources/\*\*"* />

<context:component-scan base-package=*"com.\*"* />

<bean id=*"viewResolver"*

class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<property name=*"prefix"* value=*"/WEB-INF/view/"* />

<property name=*"suffix"* value=*".jsp"* />

</bean>

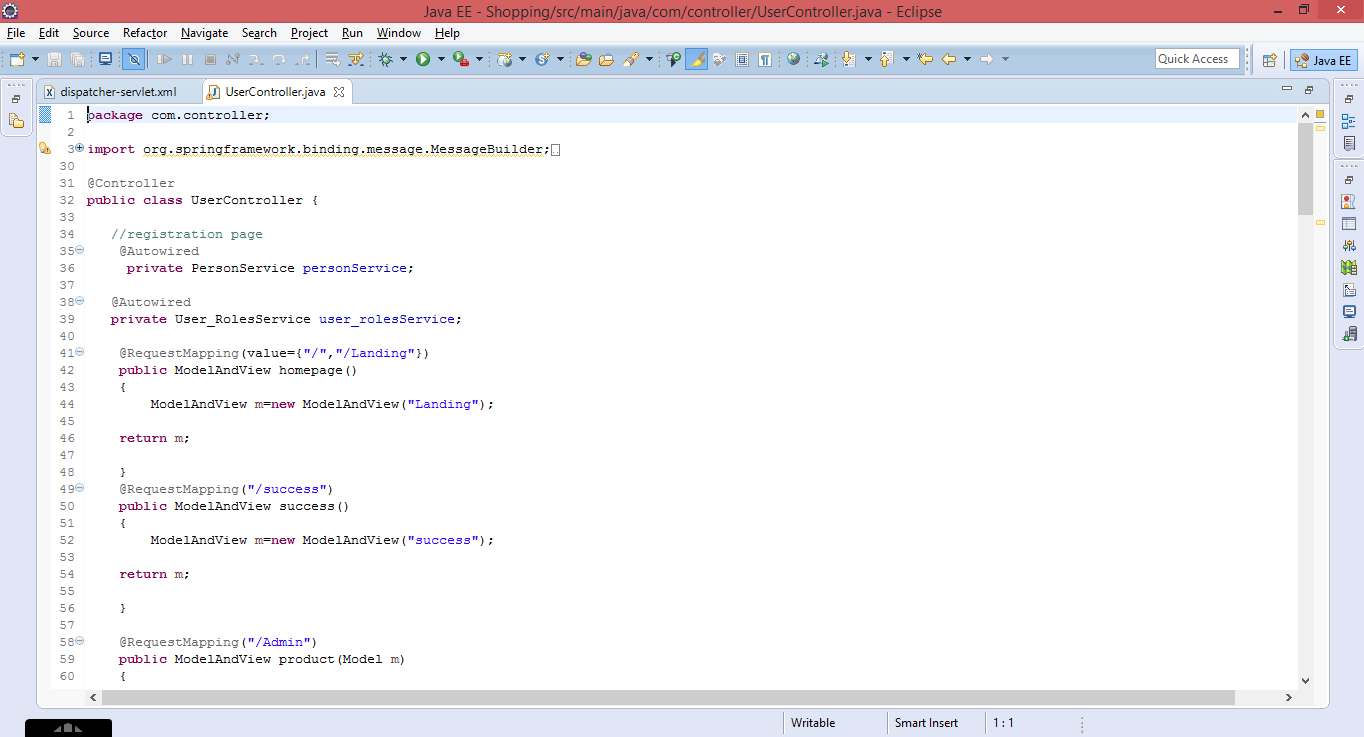
</beans>

**Following are the important points about spring3-servlet.xml file:**

* The **[servlet-name]-servlet.xml** file will be used to create the beans defined, overriding the definitions of any beans defined with the same name in the global scope.
* The **<context:component-scan...>** tag will be use to activate Spring MVC annotation scanning capability which allows to make use of annotations like @Controller and @RequestMapping etc.
* The **InternalResourceViewResolver** will have rules defined to resolve the view names. As per the above defined rule, a logical view named hello is delegated to a view implementation located at /WEB-INF/jsp/Landing.jsp .

**Defining a Controller**

DispatcherServlet delegates the request to the controllers to execute the functionality specific to it. The @Controller annotation indicates that a particular class serves the role of a controller. The @RequestMapping annotation is used to map a URL to either an entire class or a particular handler method.

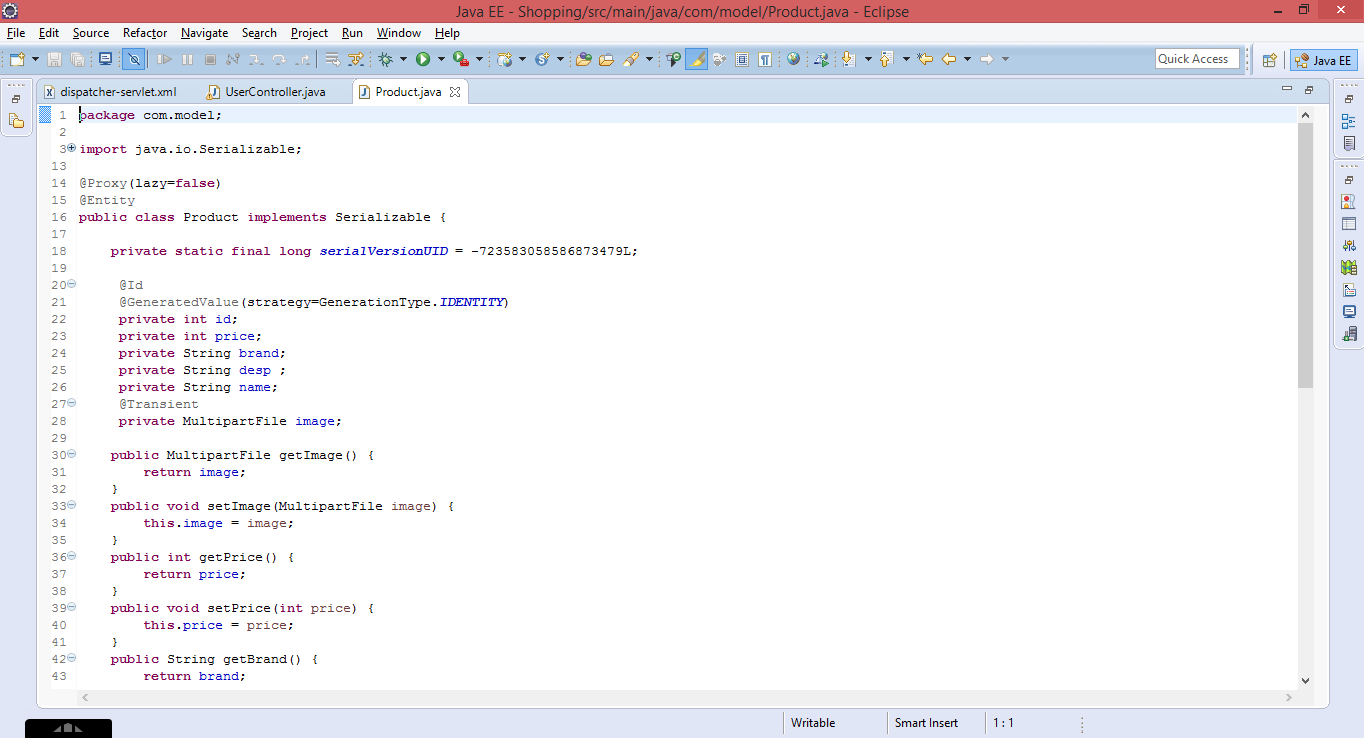


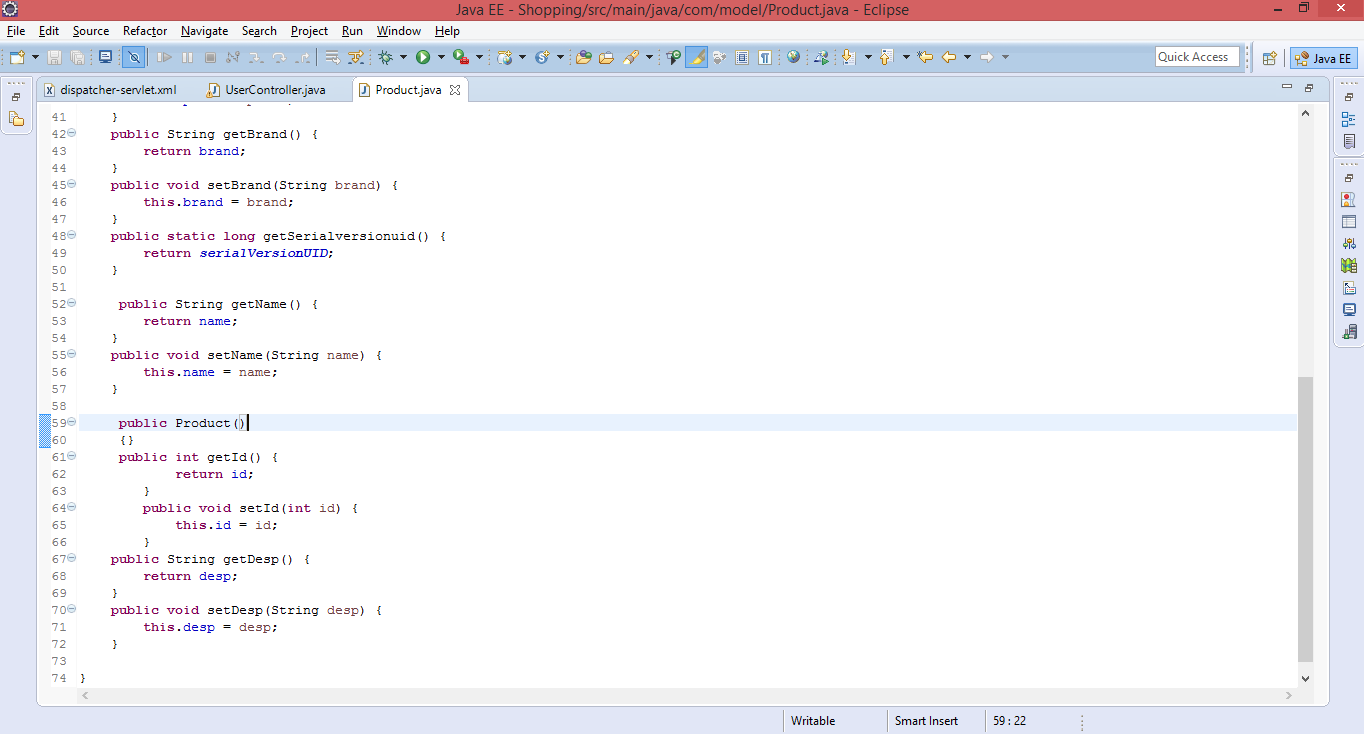
**Spring MVC Form Model Implementation**

Create a Java class Product and under the com.model package respectively. Model classes refer to data-centric classes which encapsulate closely related items.

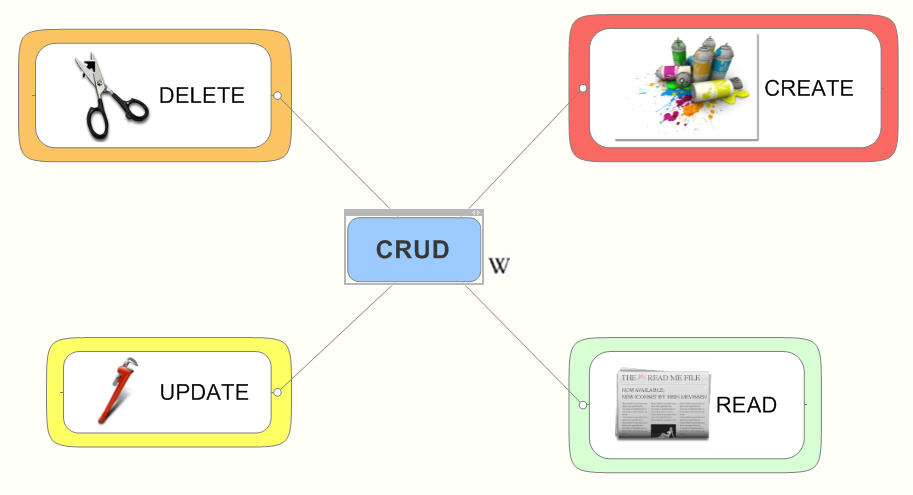
Model Objects:

* are very common, and are used in almost all applications
* are often central to an application, since they usually model problem domain objects
* often map roughly to the records of a corresponding database table
* are often used as return values for Data Access Object methods
* are easily tested using JUnit (or a similar tool)
* can be used to implement the Model in a Model-View-Controller pattern





**CRUD Operations Spring 3 Hibernate Integration**

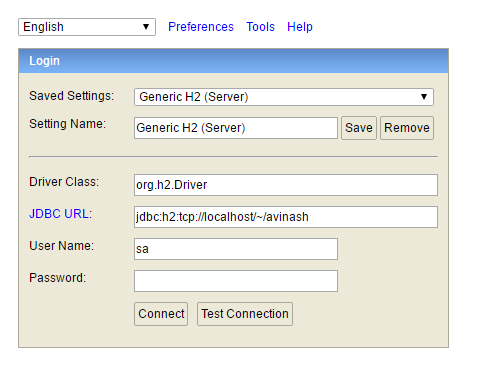
****

**Files used**

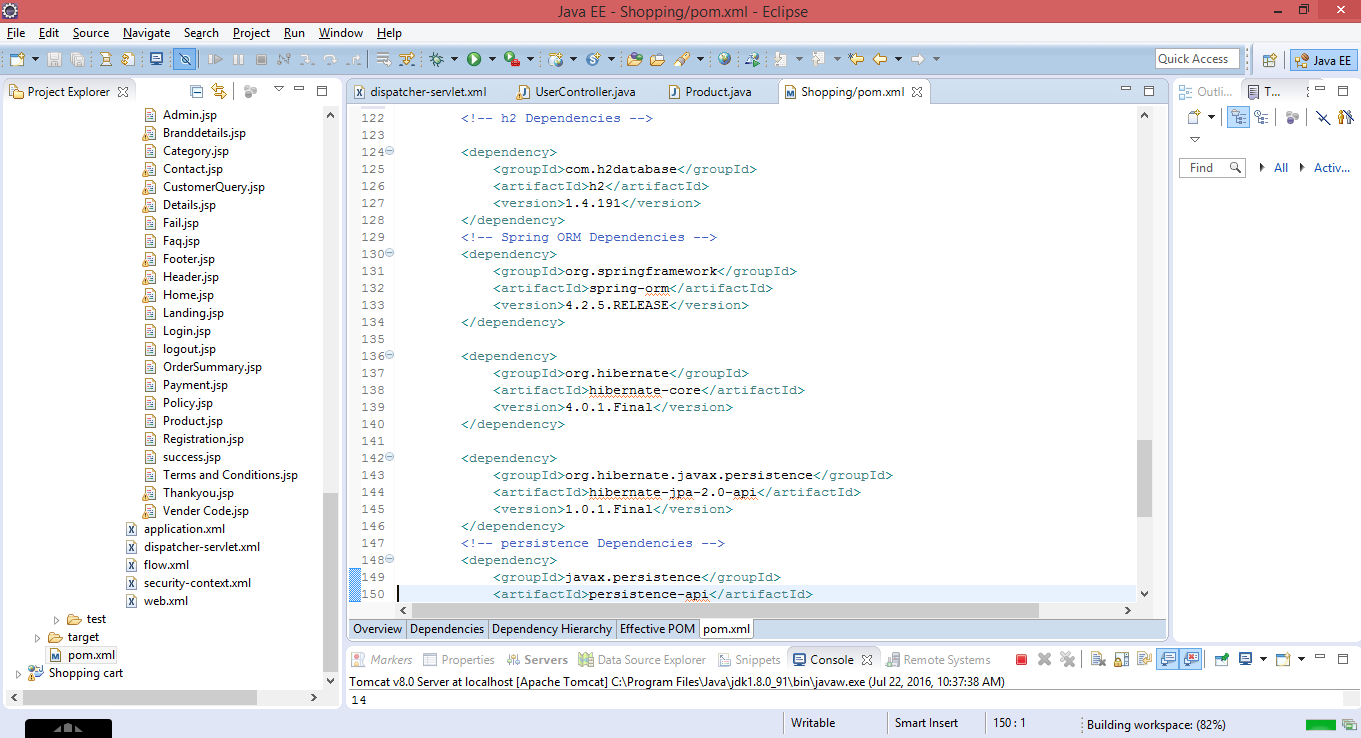
* The Spring Form Validator
* The Spring MVC Controller
* The jsp page and JavaScript files used to perform UI creation
* And the last Spring MVC configuration file

Step1: Ensure the H2 RDBMS is running on the back ground.

Step2: Ensure all database properties are correct which can be retrieved from the home screen of H2 DB. Make sure you are login into Server and not Client

****

Step 3: The following Dependency to be added



Step 4: Add Annotation to the Model classes Annotations is the powerful way to provide the metadata for the Object and Relational Table mapping. All the metadata is clubbed into the POJO java file along with the code this helps the user to understand the table structure and POJO simultaneously during the development.

**@Entity Annotation:**

The EJB 3 standard annotations are contained in the javax.persistence package, so we import this package as the first step. Second we used the @Entity annotation to the Employee class which marks this class as an entity bean, so it must have a no-argument constructor that is visible with at least protected scope.

**@Table Annotation:**

The @Table annotation allows you to specify the details of the table that will be used to persist the entity in the database. The @Table annotation provides four attributes, allowing you to override the name of the table, its catalogue, and its schema, and enforce unique constraints on columns in the table. For now we are using just table name which is EMPLOYEE.

**@Id and @GeneratedValue Annotations**:

Each entity bean will have a primary key, which you annotate on the class with the @Id annotation. The primary key can be a single field or a combination of multiple fields depending on your table structure.

By default, the @Id annotation will automatically determine the most appropriate primary key generation strategy to be used but you can override this by applying the @GeneratedValue annotation which takes two parameters strategy and generator which I'm not going to discuss here, so let us use only default the default key generation strategy. Letting Hibernate determine which generator type to use makes your code portable between different databases.

**@Column Annotation:**

The @Column annotation is used to specify the details of the column to which a field or property will be mapped. You can use column annotation with the following most commonly used attributes:

* Name attribute permits the name of the column to be explicitly specified.
* Length attribute permits the size of the column used to map a value particularly for a String value.
* Null able attribute permits the column to be marked NOT NULL when the schema is generated.
* Unique attribute permits the column to be marked as containing only unique values.

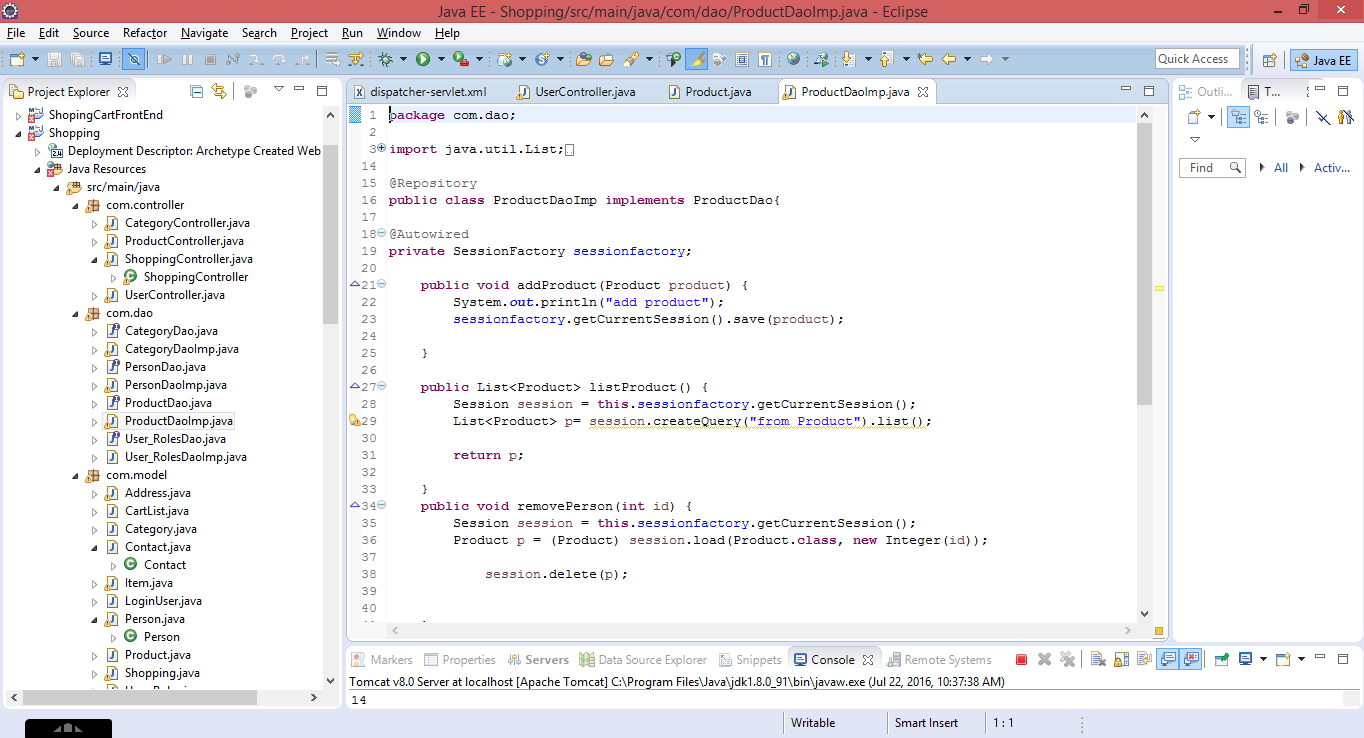
**Create DAO Class**

The Data Access Object (DAO) support in Spring is aimed at making it easy to work with data access technologies like JDBC, Hibernate, JPA in a consistent way. This allows one to switch between the aforementioned persistence technologies fairly easily and it also allows one to code without worrying about catching exceptions that are specific to each technology.

@Repository – Indicates DAO component in the persistence layer

@Controller – Indicates a controller component in the presentation layer.

@Autowired annotation provides more fine-grained control over where and how autowiring should be accomplished. The @Autowired annotation can be used to autowire bean on the setter method just like @Required annotation, constructor, a property or methods with arbitrary names and/or multiple arguments.



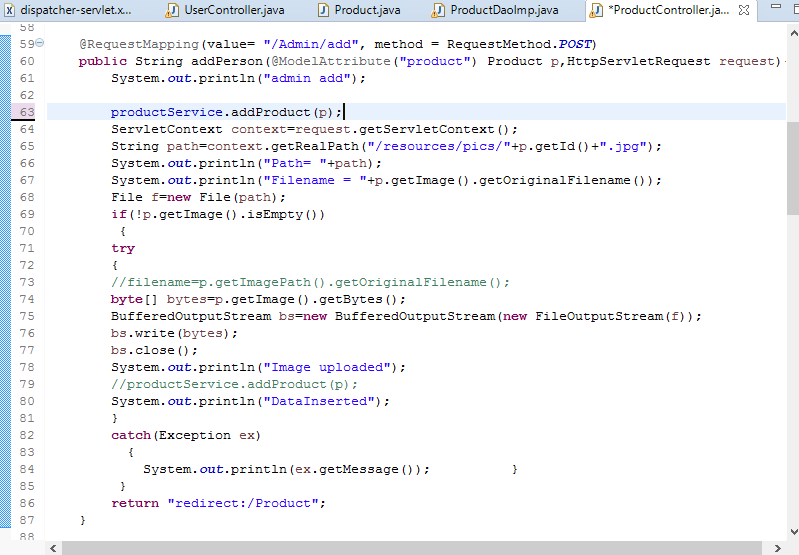
**Added Multi-Part**

In Spring MVC, a controller is used to handle file upload requests. The following code provides the web app with the ability to upload files.

**Adding Model**



**Adding controller**



**Spring Security**

Spring Security is a framework that focuses on providing both authentication and authorization to Java applications. Like all Spring projects, the real power of Spring Security is found in how easily it can be extended to meet custom requirements.

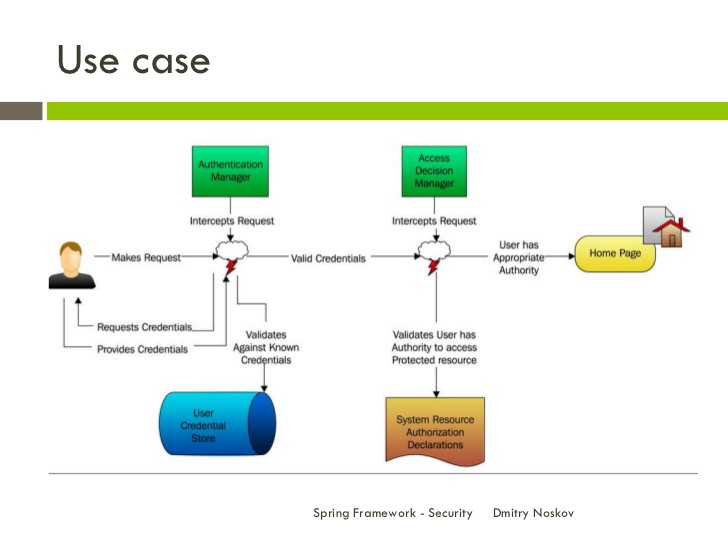
**Features**

• Comprehensive and extensible support for both Authentication and Authorization

• Protection against attacks like session fixation, clickjacking, cross site request forgery, etc

• Servlet API integration

• Optional integration with Spring Web MVC



**Web-Flow**

Spring Web Flow provides a declarative flow definition language for authoring flows on a higher level of abstraction. It allows it to be integrated into a wide range of applications without any changes (to the flow programming model) including Spring MVC, JSF, and even Portlet web applications. The following are common issues observed in stateful web applications with navigation requirements:

• Visualizing the flow is very difficult.

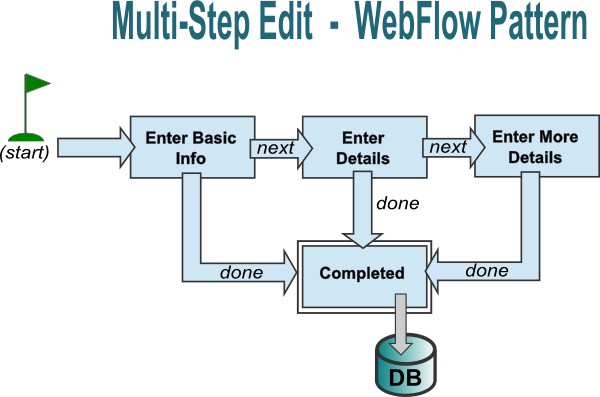
• The application has a lot of code accessing the HTTP session.

• Enforcing controlled navigation is important but not possible.

• Proper browser back button support seems unattainable.

• Browser and server get out of sync with "Back" button use.

• Multiple browser tabs causes concurrency issues with HTTP session data



**Screen Shot of Project:-**



