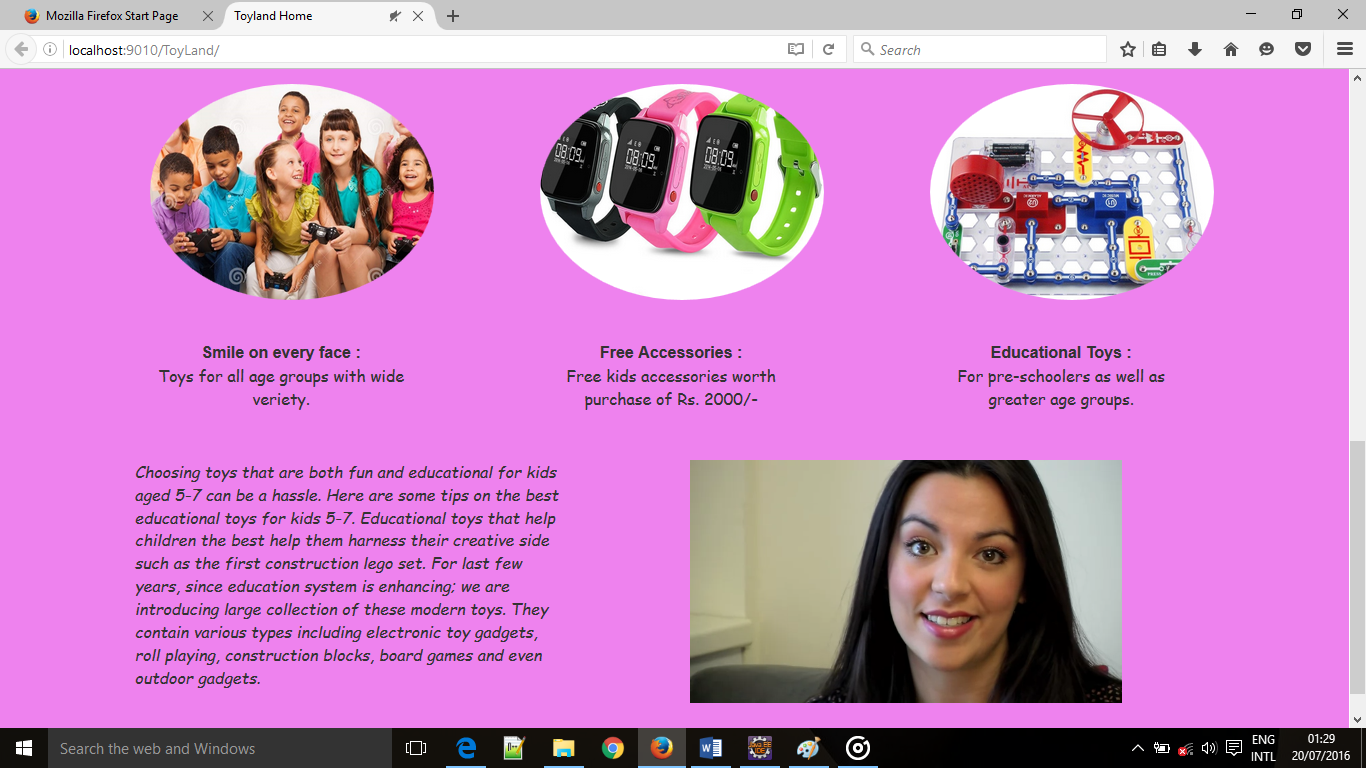
Project On

Online Toyshop E-Commerce Website

Submitted towards Digital Transformation Training

@ NIIT Mumbai





**INTRODUCTION**

The project titled “toyland.com” is online toy shopping web application developed in rapid pace as more and more businesses are moving towards adapting E-commerce. This I why, software development organizations are looking for professionals who can develop such diverse applications.

**Scope**

Being online shopping web application toyland.com enables learner to implement various aspects of developing E-Commerce applications. Any well-functioning e-commerce site needs to have various key features like :

* Attractive as well as responsive web pages
* Web pages which give details of products
* Light-weight or Fast-download webpages
* Security
* Proper web flow
* Maintain Software Versioning and Control

**Objective**

The project learning, implementation and methodology is centered on Project Based Learning will encompass the various aspects of enterprise app development and covers the following objectives:

* Understand 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 JavaScript frameworks like Bootstrap
* Build persistent and loosely coupled Web apps using Hibernate and Spring technologies
* Build high quality Web/enterprise apps by using DevOps platform

**Methodology**

CODING, Design Thinking, Code Review and Refactoring are the Methodology components of the program. The Program using the Project Based Learning Methodology will focus on building coding skills of the programmer.

**Hardware and Software Requirements**

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

Hardware Requirements:

* Intel Core i3 processor minimum
* 2 GB RAM minimum (Recommended 4GB)
* System type : 64bit
* Hard Disk Space Required 10GB Min
* Internet Connection with Minimum 512kbps speed (1 MBPS Recommended)

Software Requirements:

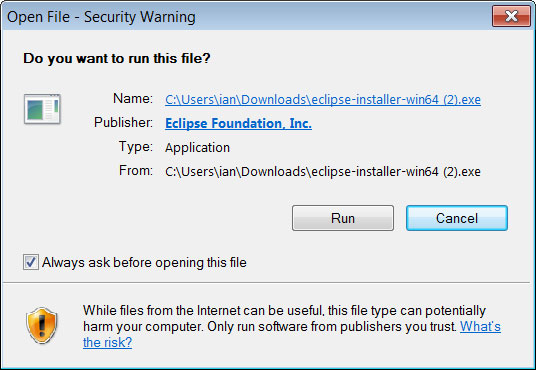
* Operating System Windows 7, 8 or 10
* Eclipse Mars IDE
* Apache tomcat 8 server
* Java Development Kit version 8
* H2 Database
* Bootstrap Scripts
* Angular JS Scripts
* Github Desktop (optional)

**Software installations**

**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**.



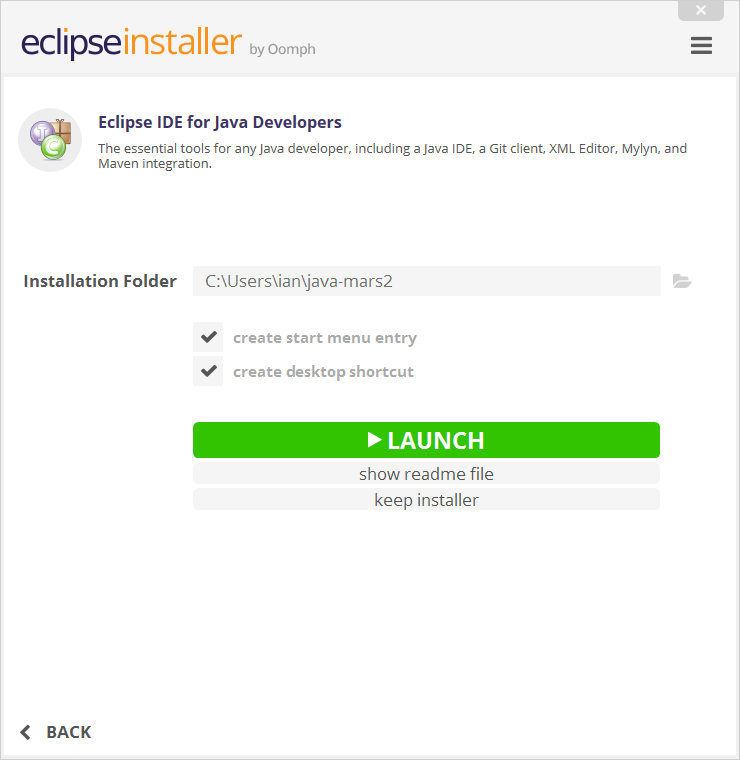
* Select Eclipse IDE for JAVA EE Developers (Eclipse Mars).



* Select your installation folder, Specify the folder where you want Eclipse to be installed. The default folder will be in your User directory. Select the ‘Install’ button to begin the installation.



* Launch Eclipse, Once the installation is complete you can now launch Eclipse.



**Getting Started 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>

**Apache Maven**

**Prerequisites**

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

**Installation**

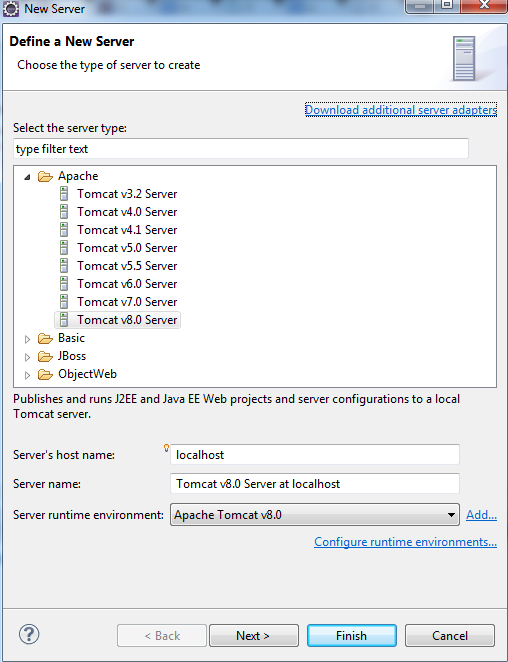
Maven is a Java tool, so you must have Java installed in order to proceed.

**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>.

**Tomcat8 server (integration with 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.
* Check http://localhost:8080 (provided you installed Tomcat 8 and set Http port as 8080)
* Interestingly, you would NOT see the welcome page, but the 404 error page.

To get rid of that, Double click on ”Tomcat v8.0 Server at localhost”. In the window that opens up, select “Use Tomcat installation” and, change deploy path from wtpwebapps to webapps. Look at the figure below.



* Restart the server and access http://localhost:8080 .

**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.

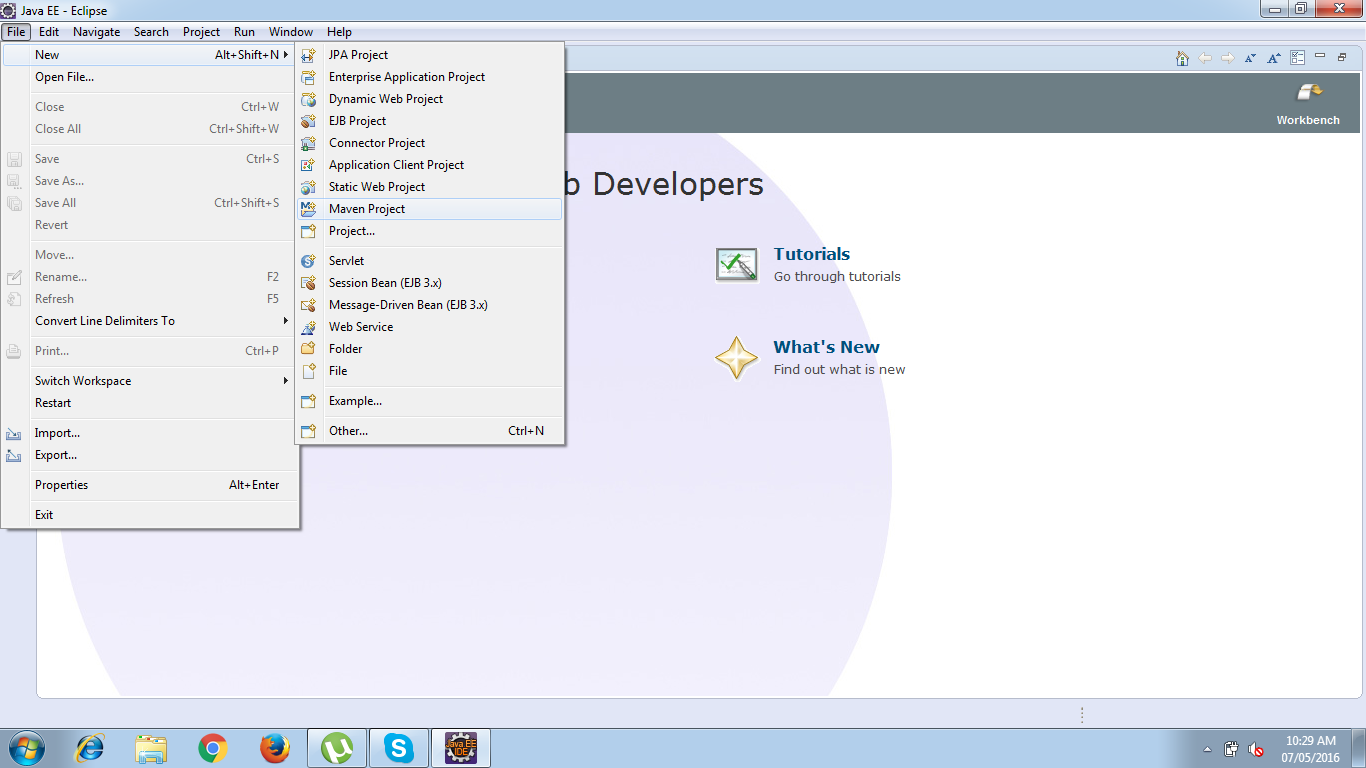
**Creating Maven Project on Eclipse**

**Prerequisite**

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

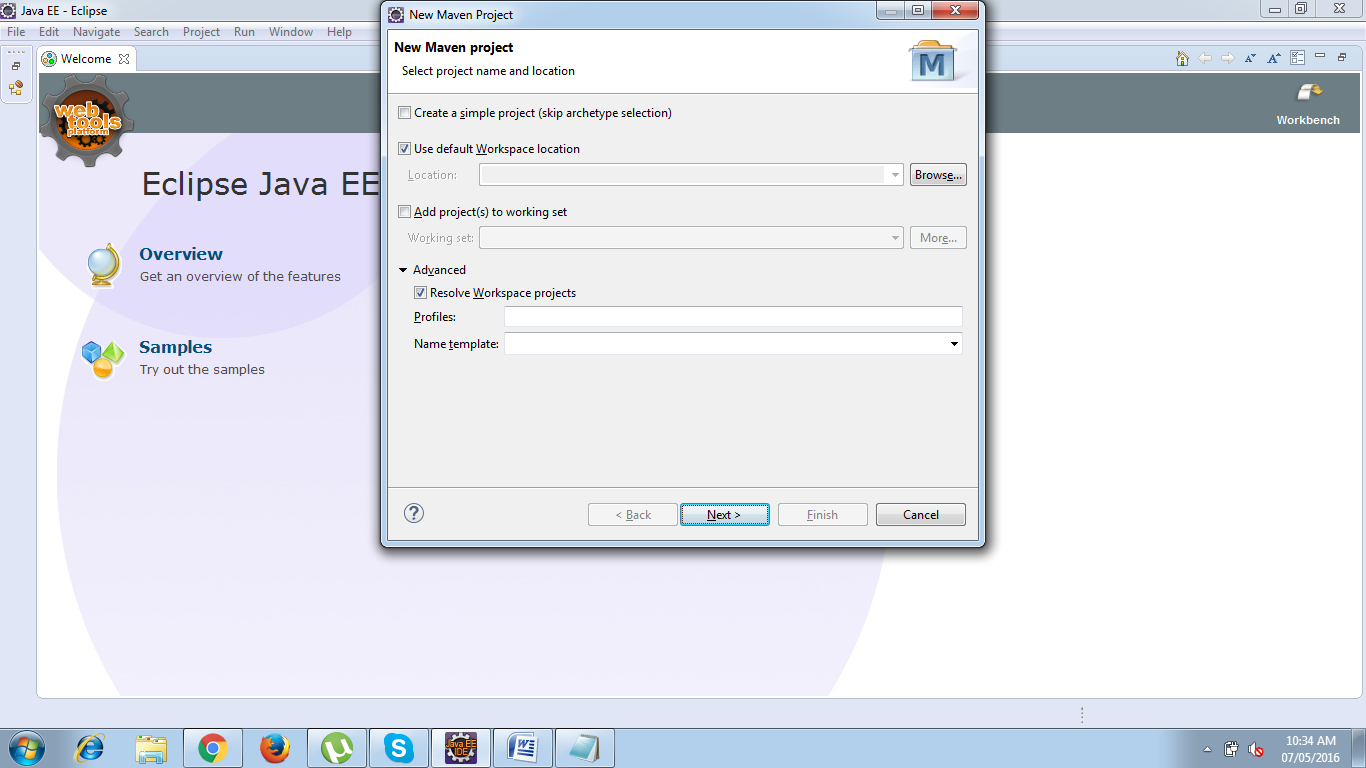
**Step 1.** Create a New Maven Project

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

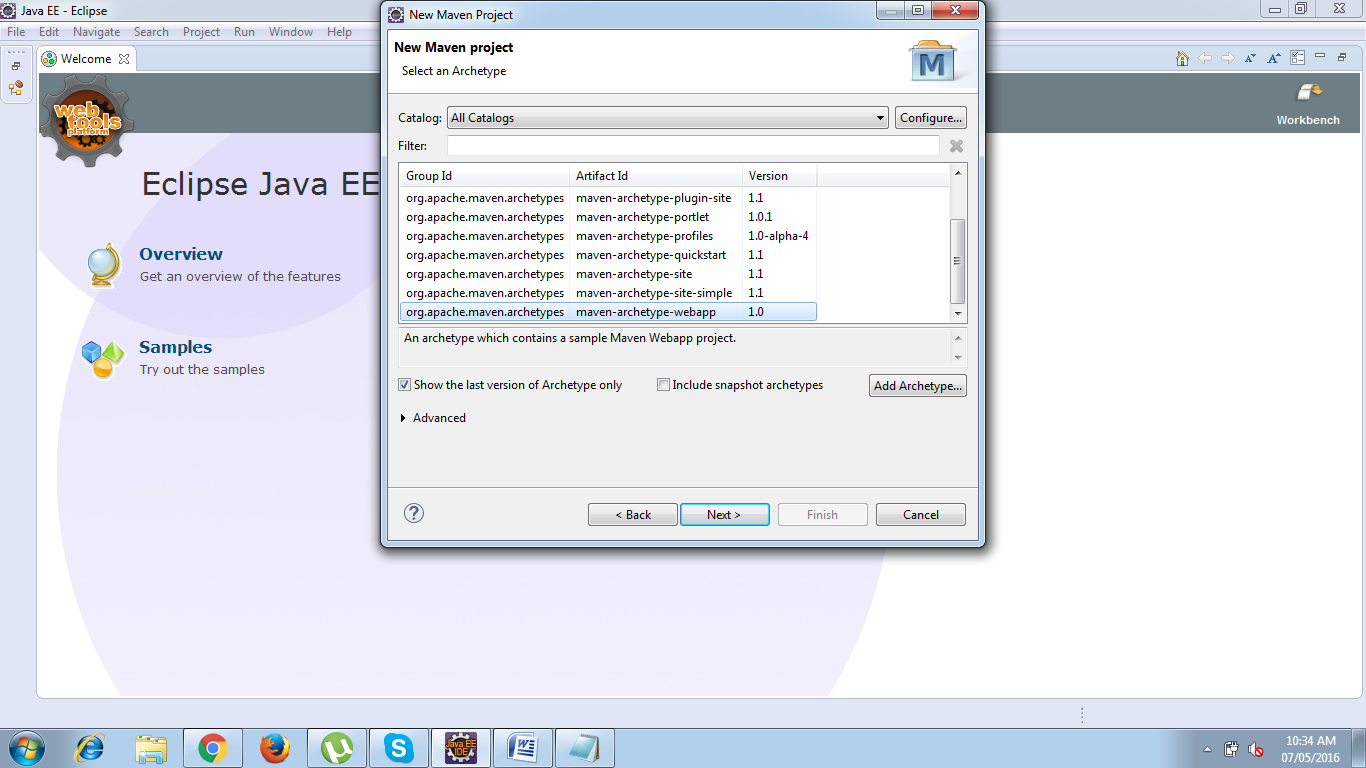


**Step 2.** Select project name and location.

Use the default Workspace location or specify the location if necessary.



**Step 3.** Select an archetype



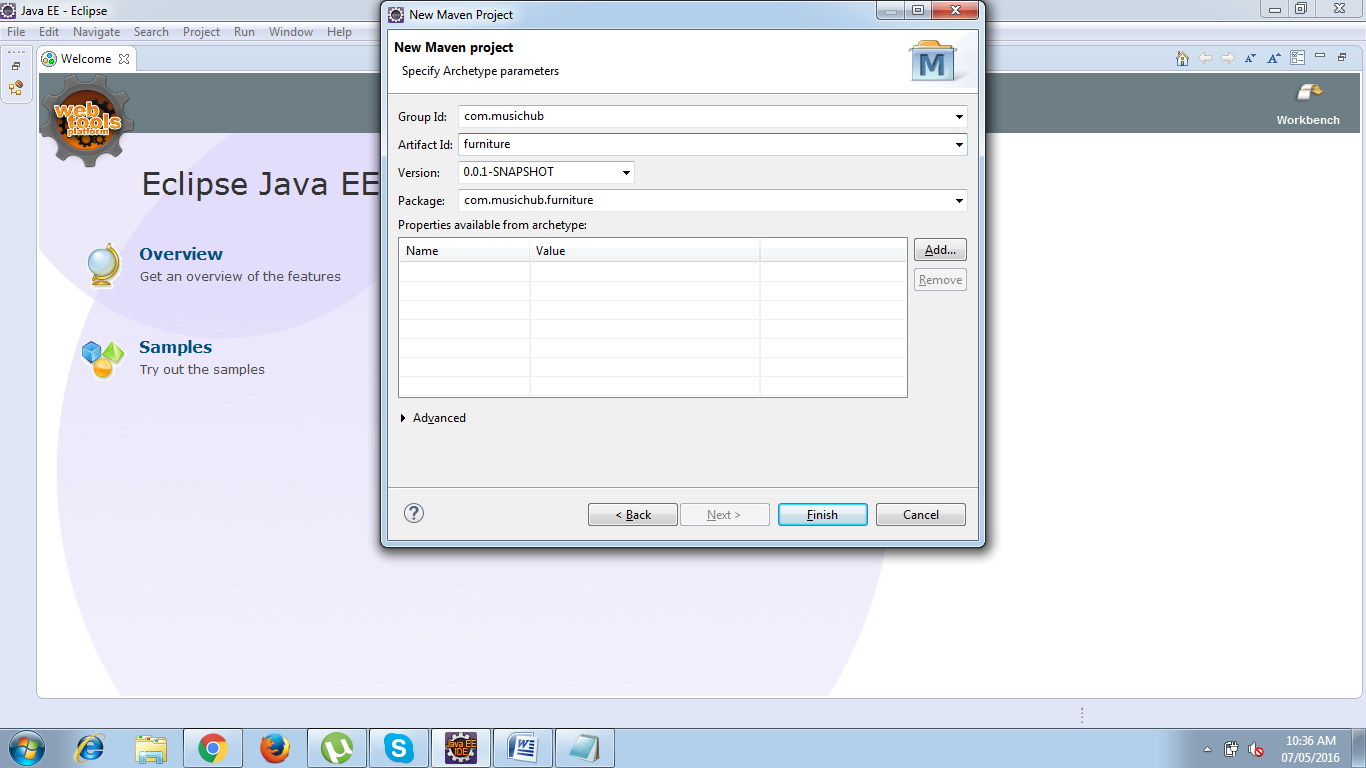
**Step 4.** Enter the Group id and Artifact Id

Enter the Group Id and Artifact Id and click 'Finish'

groupId will identify your project uniquely across all projects, so we need to enforce a naming schema. It has to follow the package name rules, what means that has to be at least as a domain name you control

artifactId is the name of the jar without version. If you created it then you can choose whatever name you want with lowercase letters and no strange symbols.

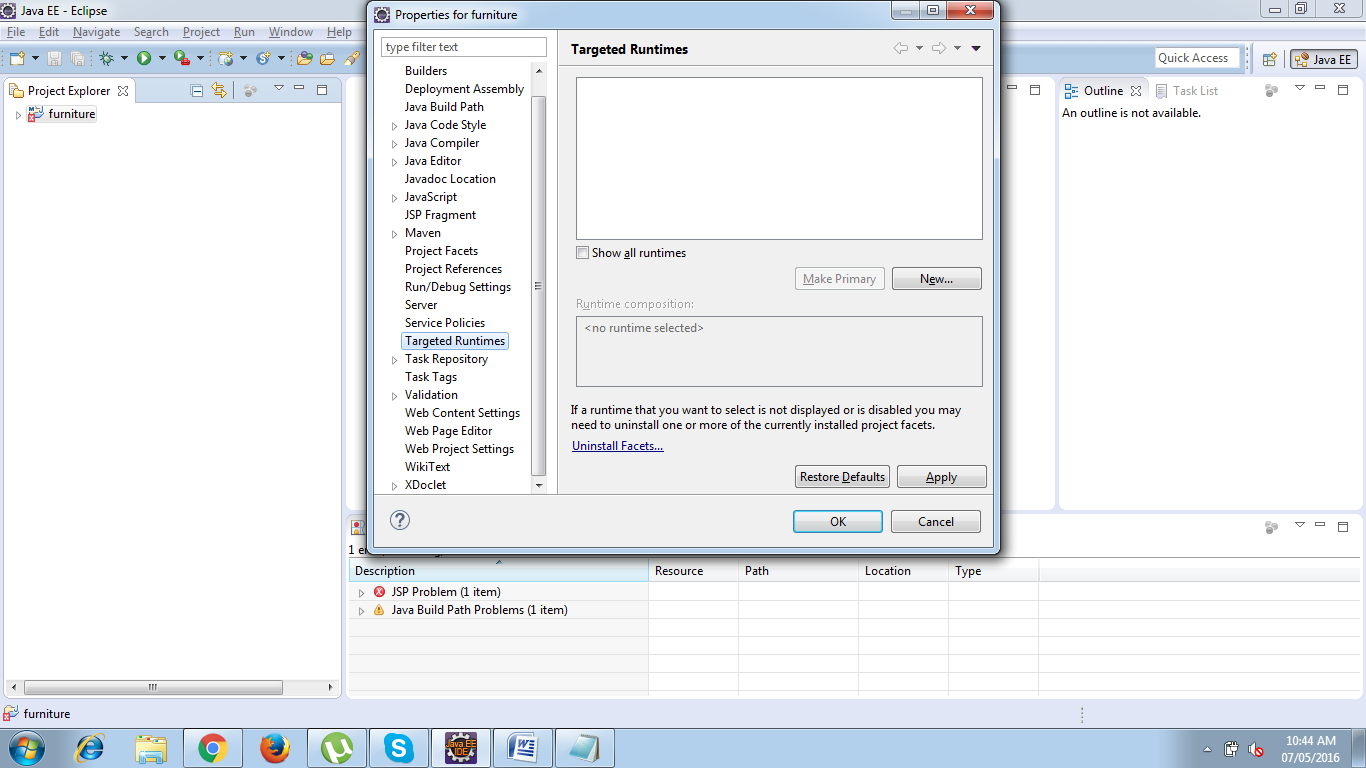
version if you distribute it then you can choose any typical version with numbers and dots (1.0, 1.1, 1.0.1, ...). Don't use dates as they are usually associated with SNAPSHOT (nightly) builds. If it's a third party artifact, you have to use their version number whatever it is, and as strange as it can look.



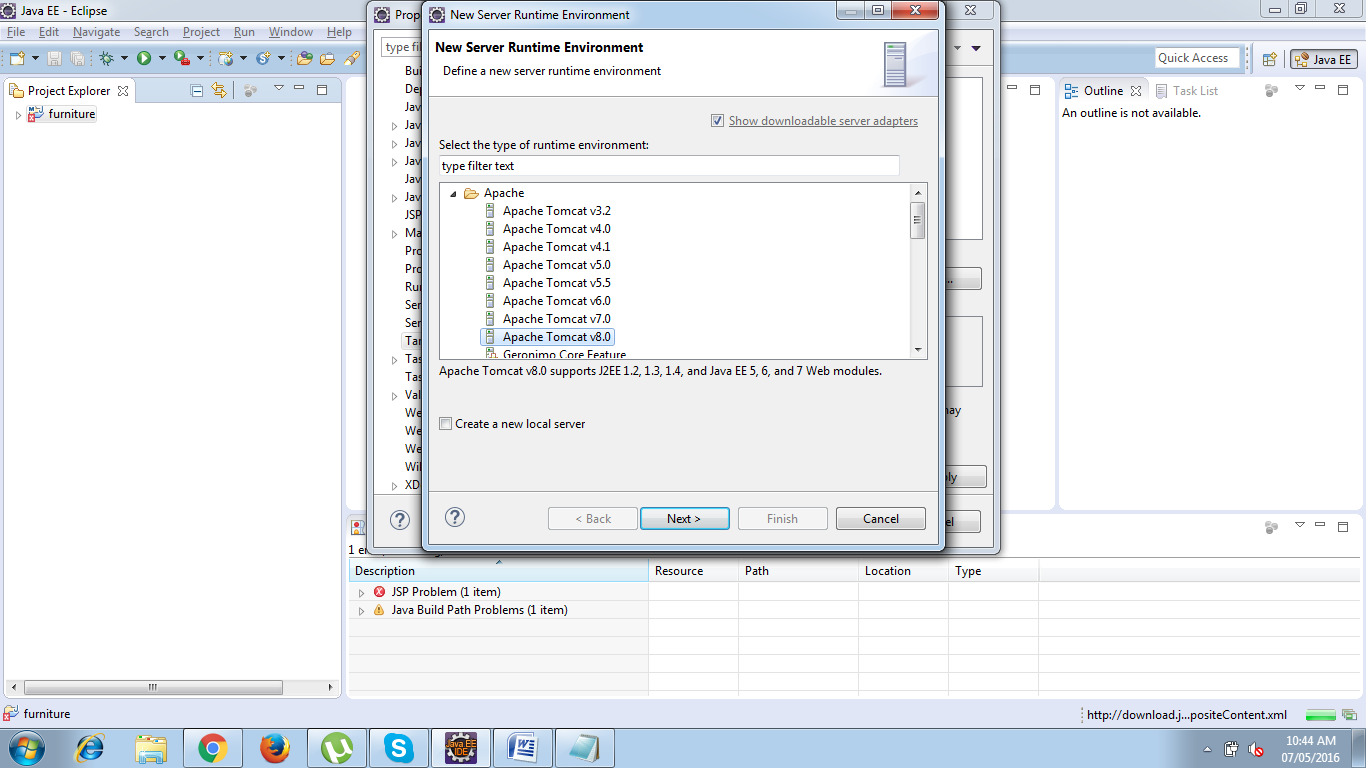
Your project will be ready but with an error due to runtime is not configured for the newly created Workspace.

Updating Target Runtime Server’s :

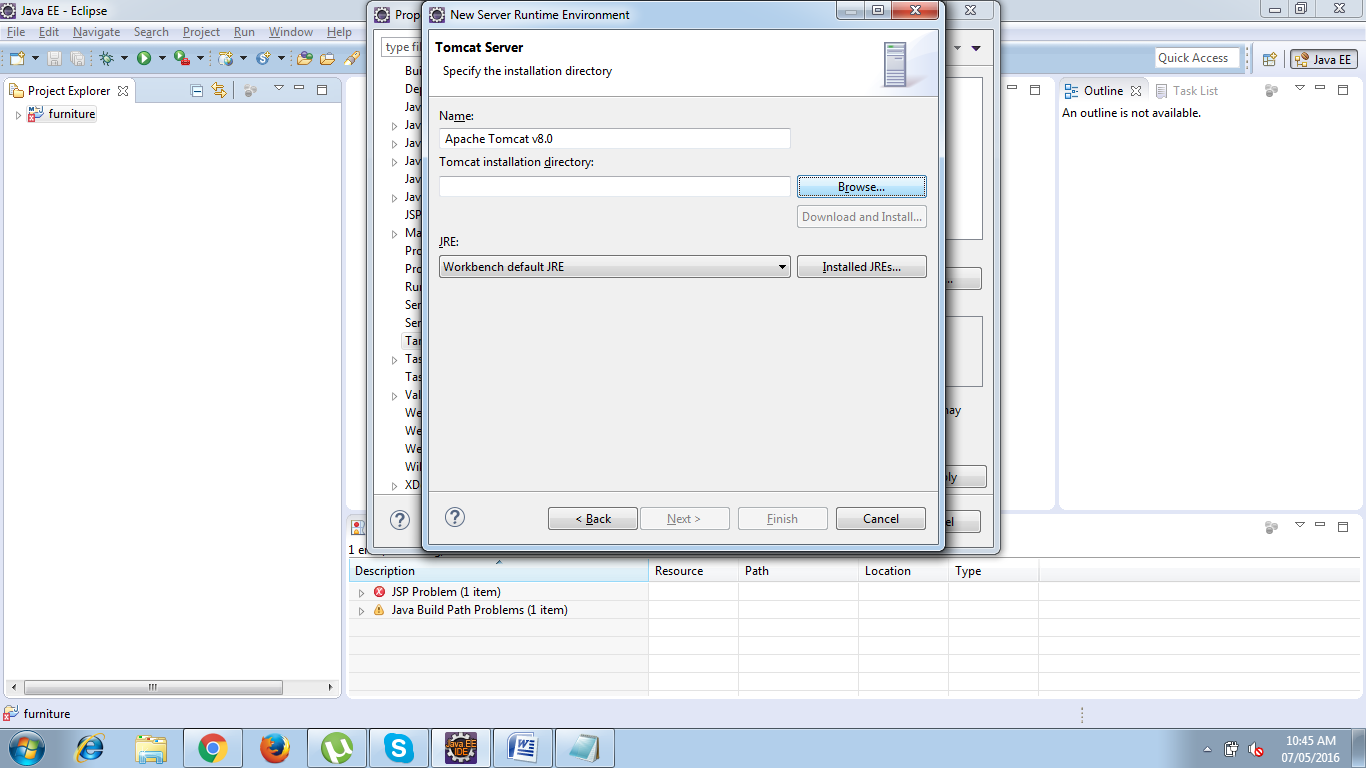
Right Click Project root folder and choose properties :

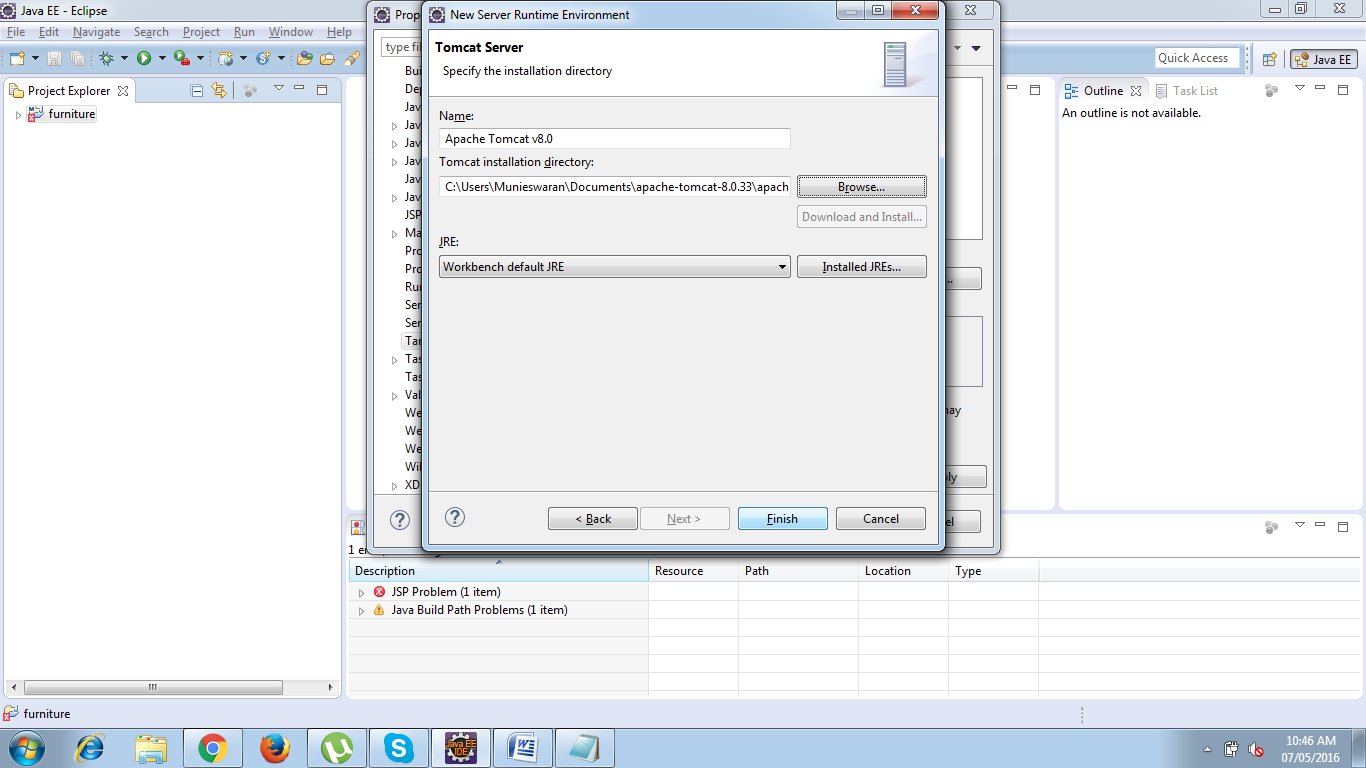


Choose Targeted Runtime which will open New Server Runtime Environment window :



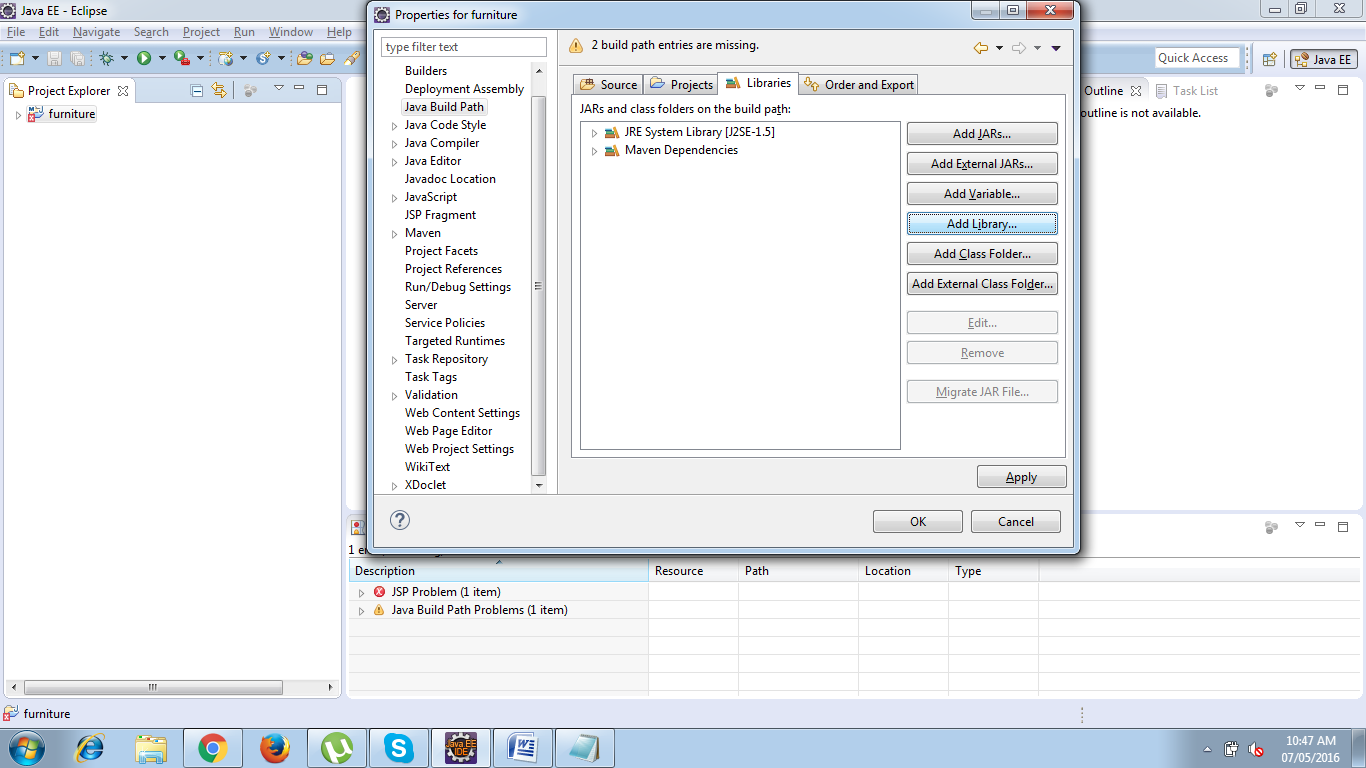
Select Apache Tomcat v8.0 and Click Next -> Choose Tom cat Installation Directory



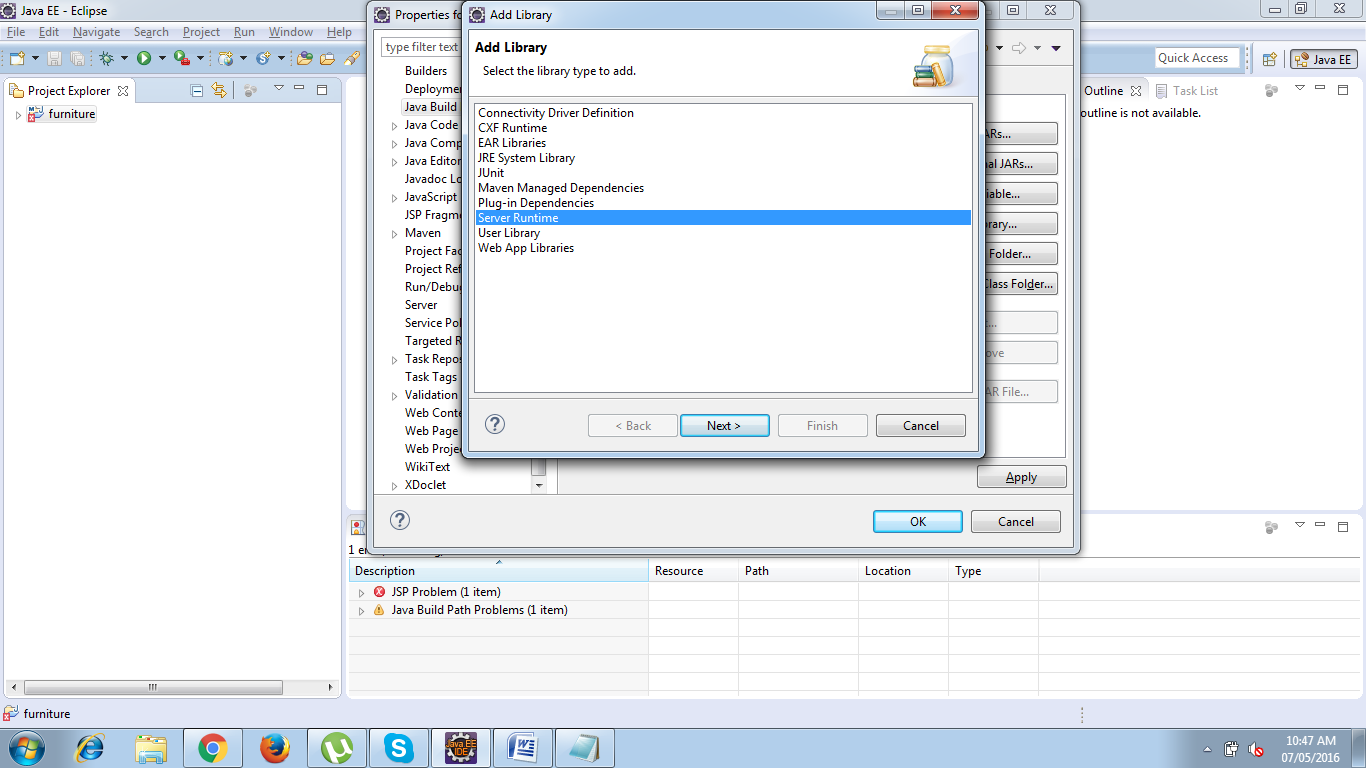


**Click Finish.**

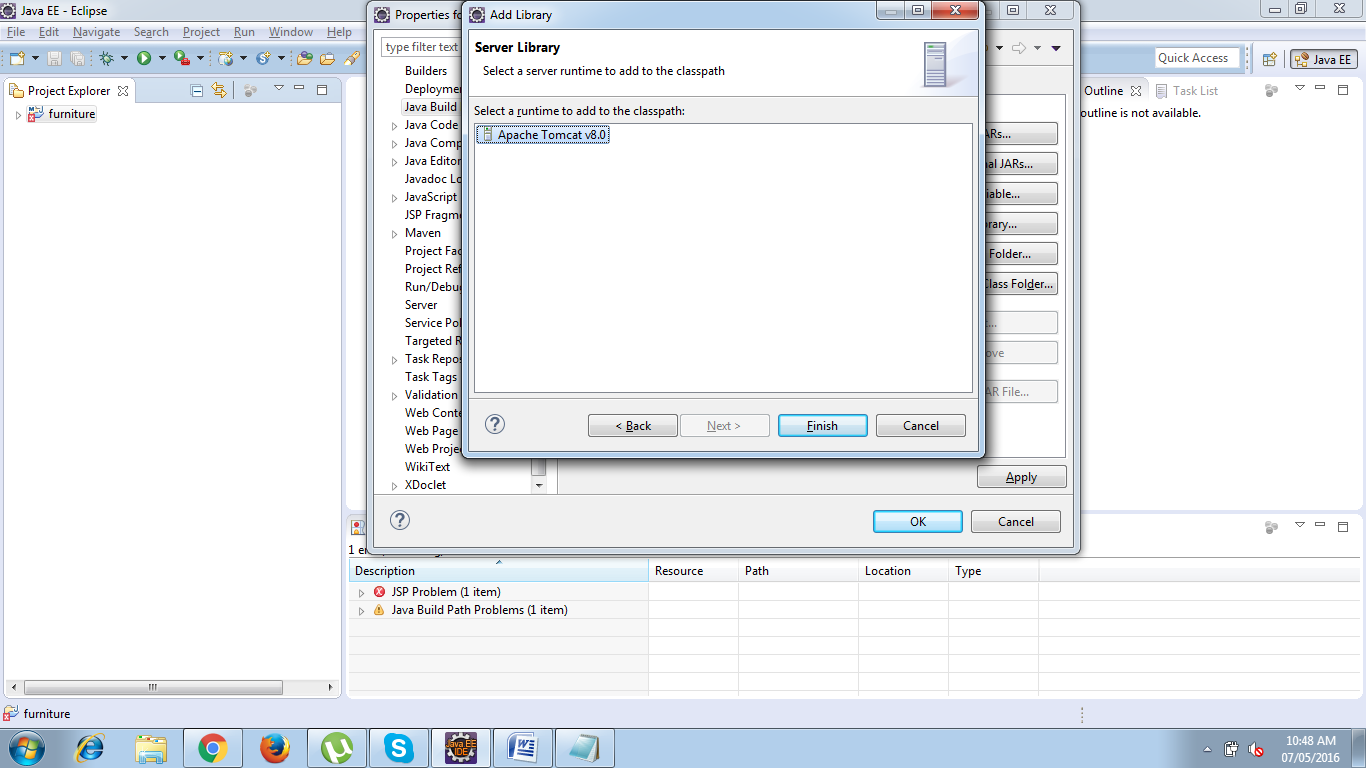
Adding build path Right click project and choose properties -> Select Java Build path -> Choose Libraries -> Select Add libraries.



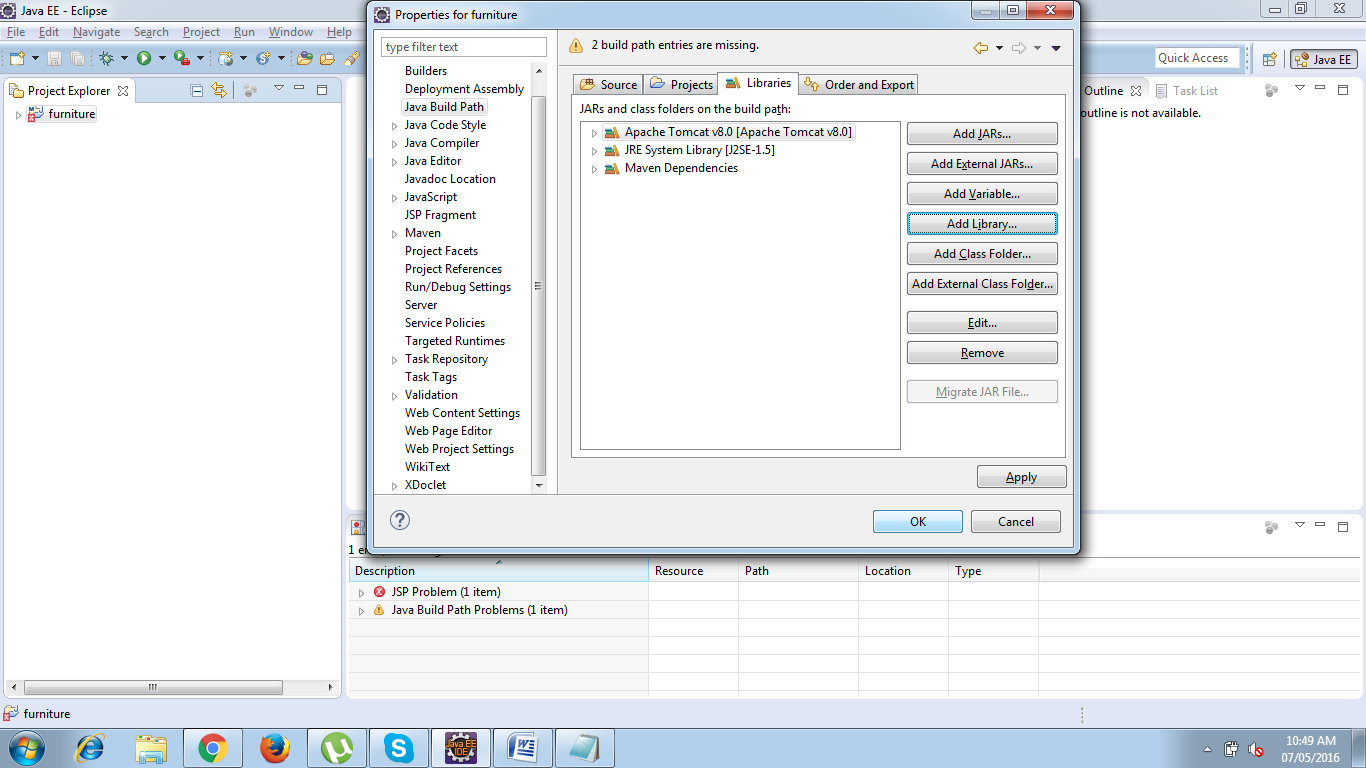
Choose Server Runtime -> Click Next



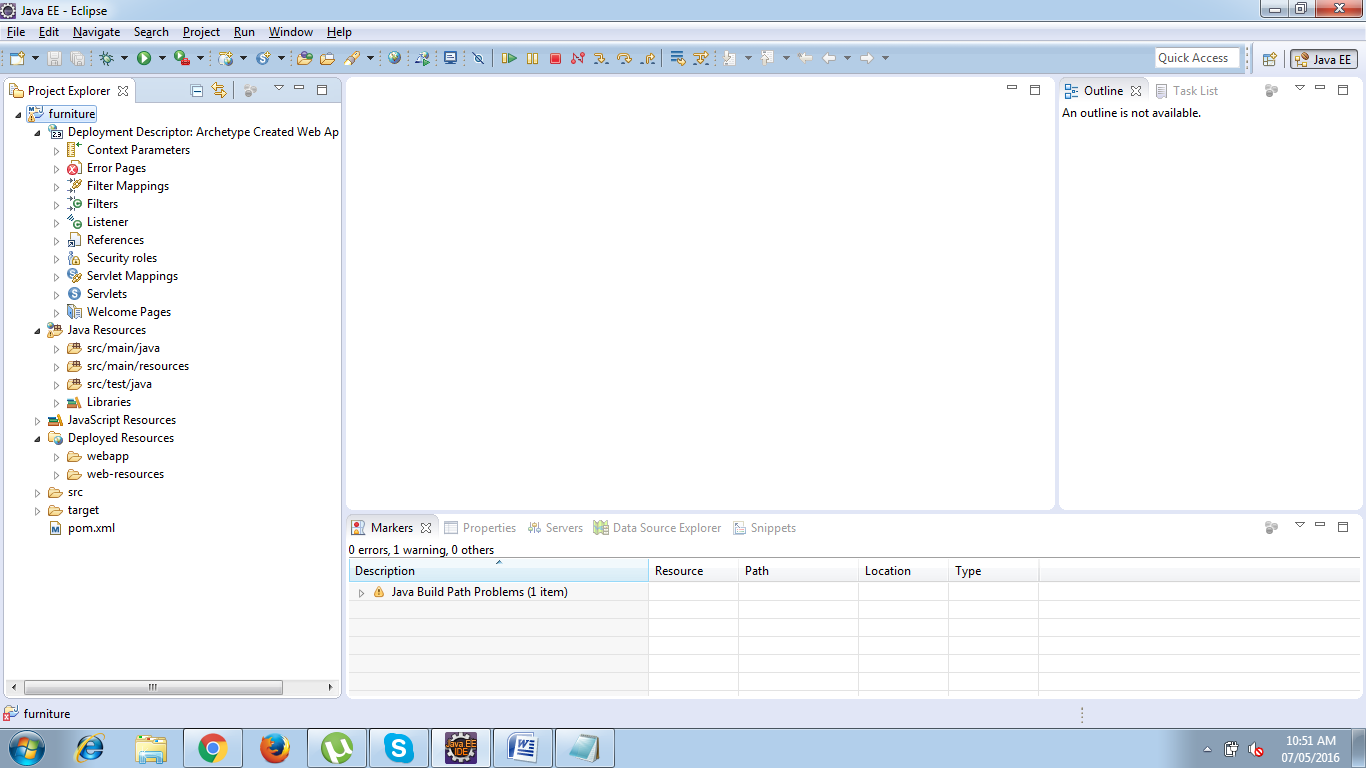
Select a runtime to add to the class path ->Apache Tomcat V8.0



Click OK in the properties window.

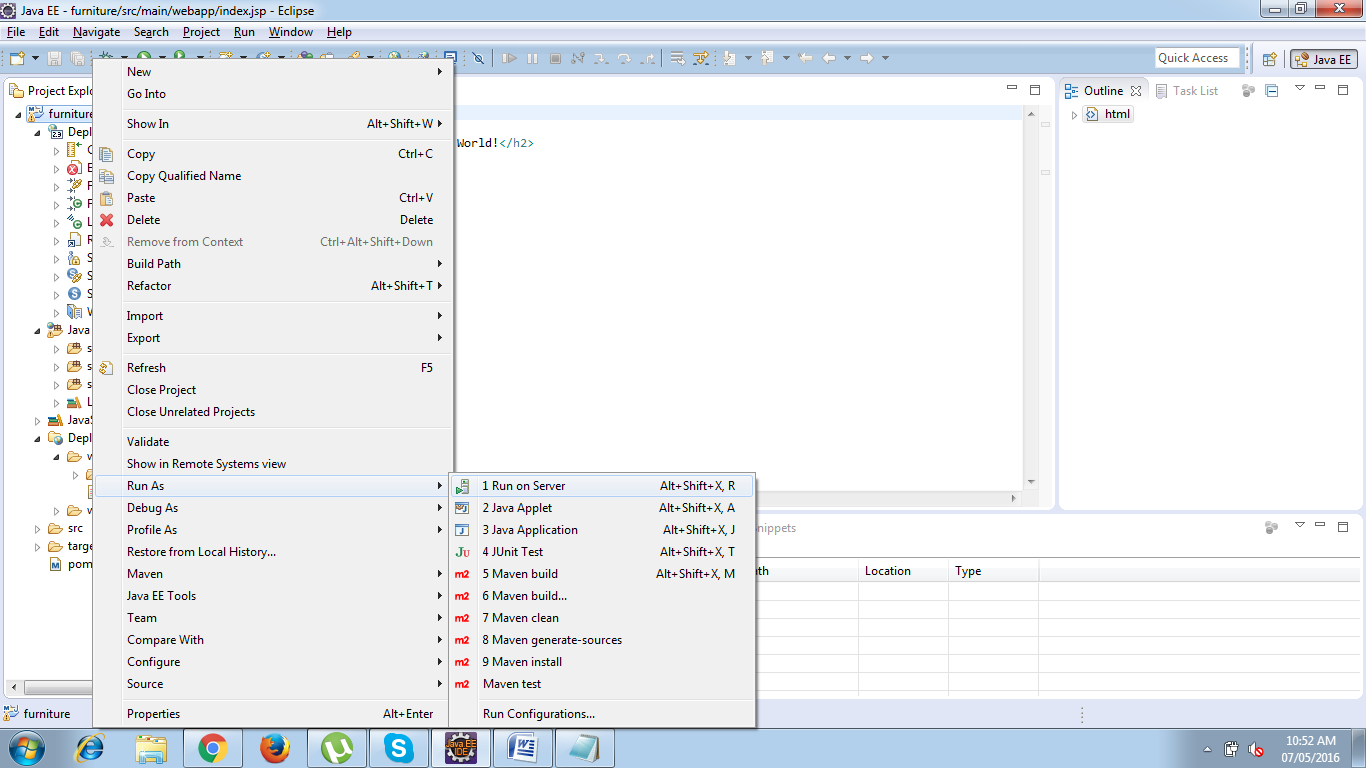


Finally your project is now ready to develop.

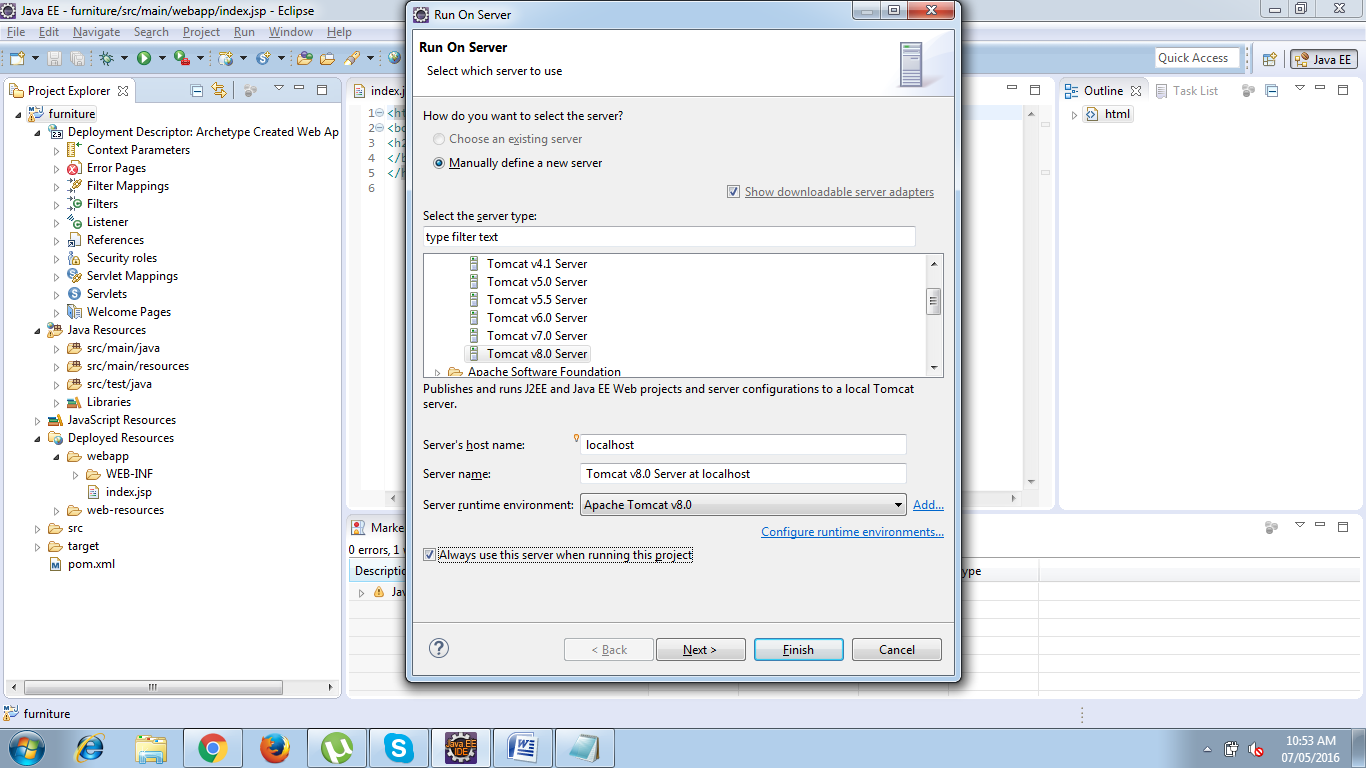


**Testing the Environment**

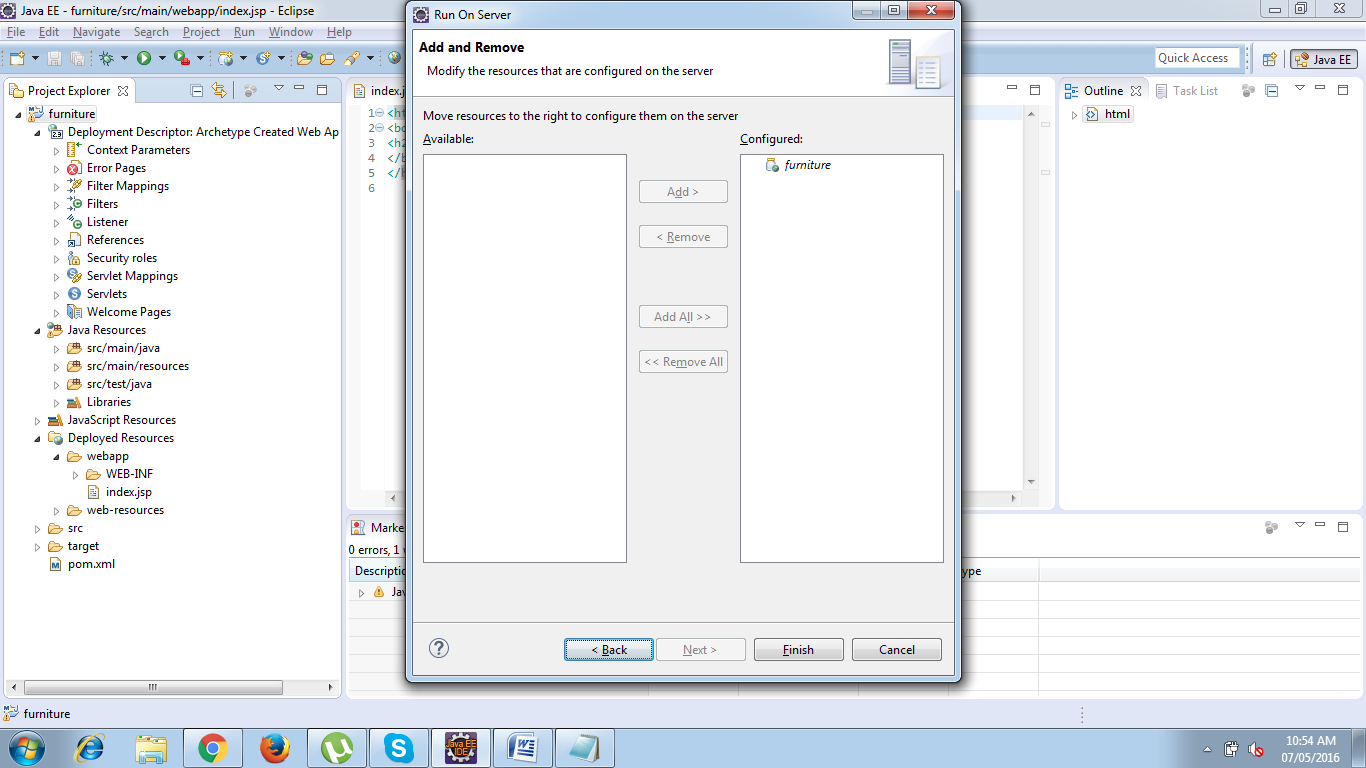
Right Click the project and choose RUN AS -> Run on Server option

****

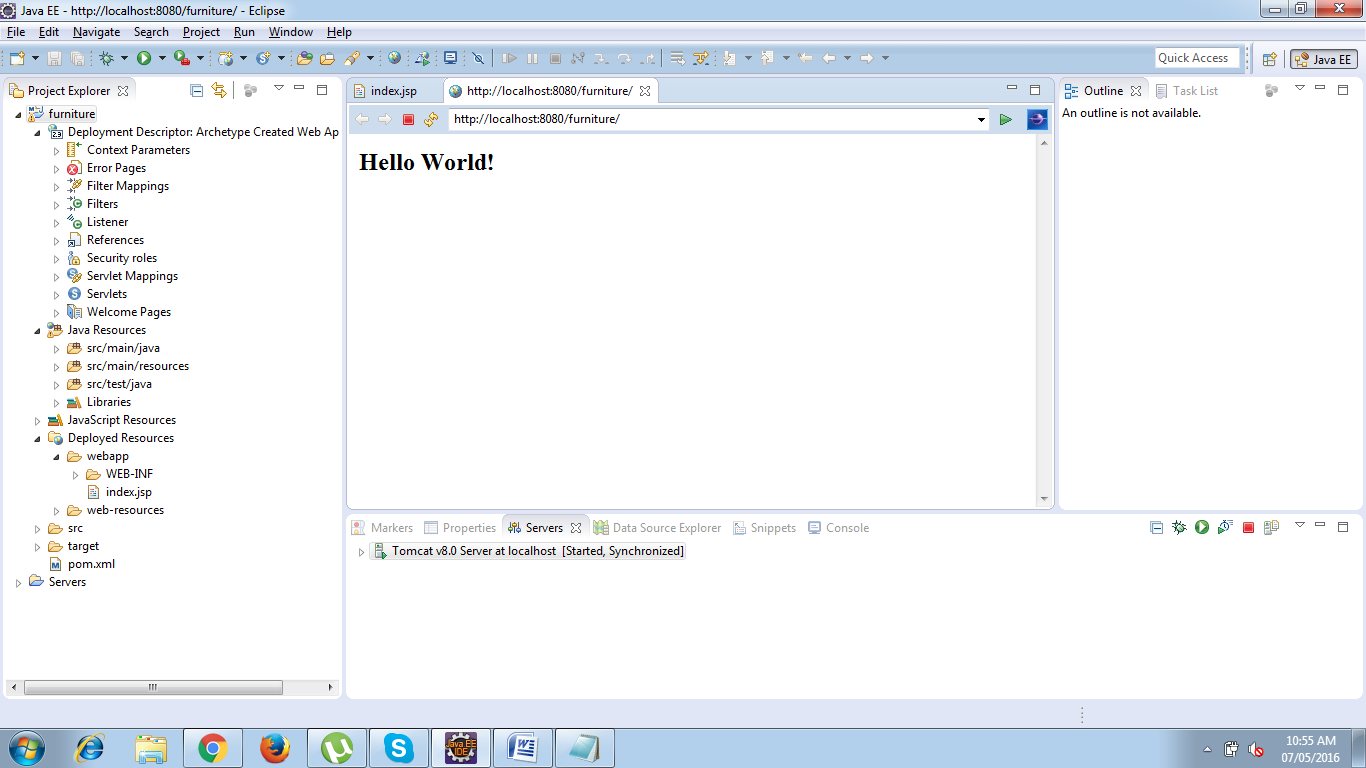
Run on Server window will be now opened , Select the Server to use and select always use this server when running the project check box.



Click Next -> Move the resources to be configured and click finish

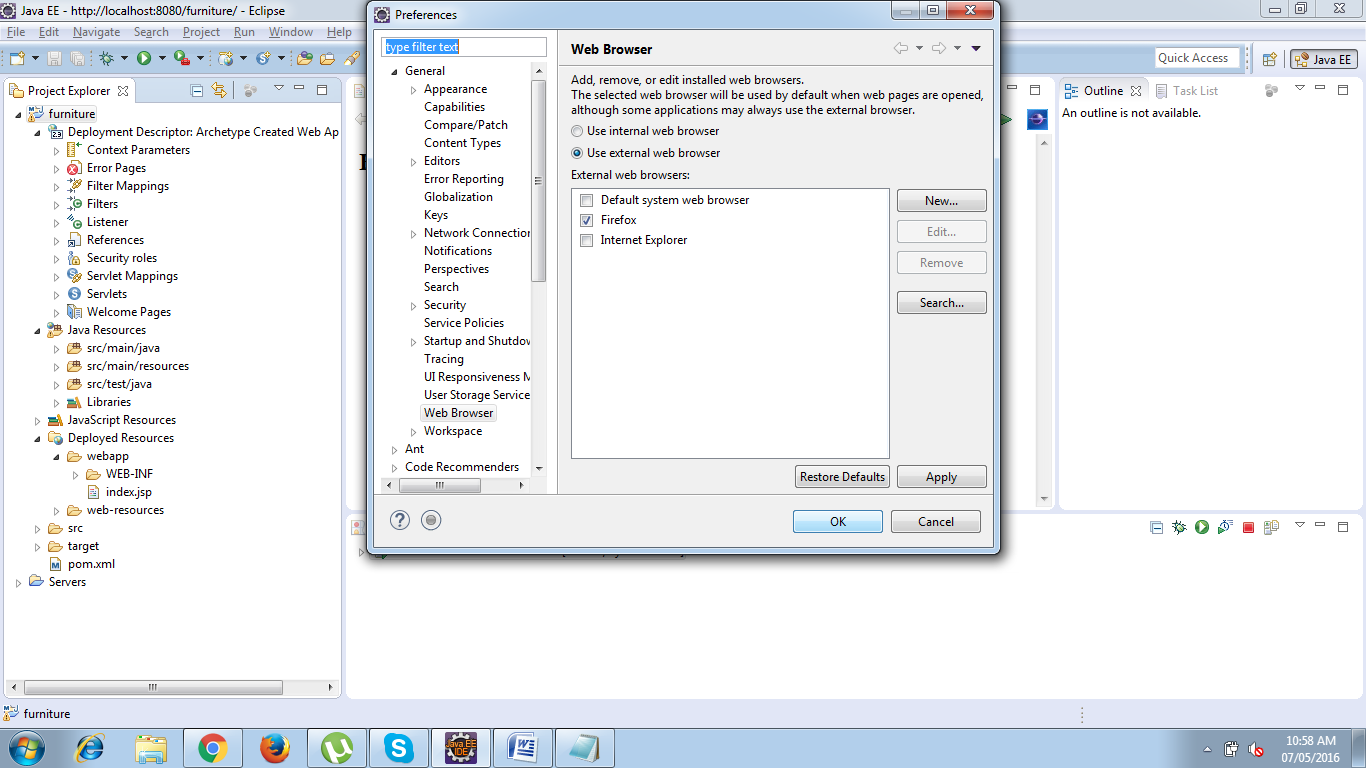


Now your server will start up and your project will be deployed into Tomcat Server and you will see the index page on your default embedded browser.



**To check the Browser choose the following option.**

* Goto Window Menu and Select Preferences :
* In the preferences Window select General->Select sub-type Web Browser ->
* Choose ***Internal***(embedded browser) or ***external*** browser and click ok.



**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

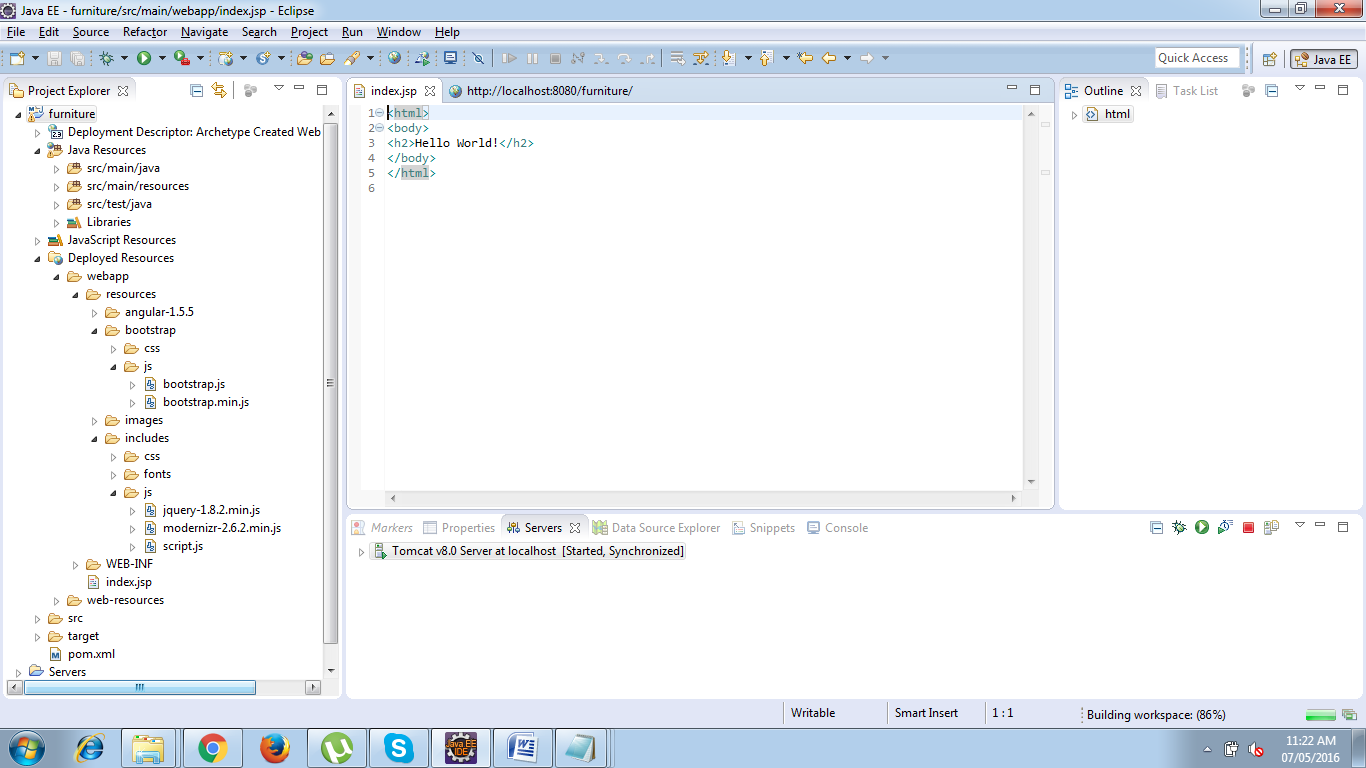
**version**: This is the last piece of the naming puzzle. groupId:artifactId denote a single project but they cannot delineate which incarnation of that project we are talking about. Do we want the junit:junit of today (version 4), or of four years ago (version 2)? In short: code changes, those changes 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.

**Adding Boot Strap Front End Framework**

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.



* You can also add Bootstrap dependency to your project dependency

<dependency>

    <groupId>org.webjars</groupId>

    <artifactId>jquery</artifactId>

    <version>2.1.1</version>

</dependency>

<dependency>

    <groupId>org.webjars</groupId>

    <artifactId>bootstrap</artifactId>

    <version>3.2.0</version>

    <exclusions>

        <exclusion>

            <groupId>org.webjars</groupId>

            <artifactId>jquery</artifactId>

        </exclusion>

    </exclusions>

</dependency>

Bootstrap javascript components needs JQuery. That’s why Bootstrap has a dependency to JQuery. You can manage the version by defining a dependency management to the project pom.xml

Web Ref : <http://www.w3schools.com/bootstrap/>

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

<script src="<spring:url value="/resources/js/bootstrap.js"/>"></script>

<script src="<spring:url value="/resources/jquery/jquery-2.2.4.min.js" />"></script>

Bootstrap download url : <http://v4-alpha.getbootstrap.com/getting-started/download>

**MAVEN MVC PATTERN**

**Implementing the VIEW COMPONENT**

* Move your jsp to /WEB-INF/view/index.jsp
* Update your app context so '/WEB-INF/view/' is the prefix, and '.jsp' is the suffix.
* Add the context loader listener to your web xml, and set the contextConfigLocation path relative to the app context root. For example, it could be /WEB-INF/Dispatcher-Servelt.xml
* make a request to localhost:8080/toyland/

For the paths, it might help to think of things step by step :

1. You request the url from your browser, The browser looks at the protocol, host, and port, and uses a DNS to find the appropriate IP address to connect with.
2. A connection is established between your browser and the host. The host looks for a process running on the port you specified, and if a TCP connection is allowed by any security systems in place, the request is streamed to the process running on that port, the web server.
3. The web server makes decisions based on what's after the port, specifically, it determines what the web application context is by looking at the path it was given. Once it determines the application context root, it knows which web application should handle that request. The decision is based on how you configure the web server, you could have a web application handle requests with no context root, or a specific context root. For example, if you requested localhost:8080/CtxtRoot/jsp/, you could have one web application on the server whose context root is "CtxtRoot", and it would handle that request. Alternatively, you could have an application that has "" for a context, and it could handle that request. It depends on how you configure the server, by default Tomcat will use the war name as a context root.
4. The web application receives the request. While it knows the full URL requested, it only makes decisions based on everything after the context root. So for example, with the request to localhost:8080/CtxtRoot/jsp/, the web application would route things based on 'jsp' as the path.
5. The web application has a filter chain that it submits the request to first. If a filter's pattern matches the request, that filter can evaluate the request. It might block the request, handle the request, or pass it on. I won't say much more because your question doesn't involve filters.
6. The web app looks for a resource whose pattern matches the request, it considers servlets first, and then static resources. The url part that comes AFTER the context is what it tries to match, so if the request was for localhost:8080/CtxtRoot/jsp/, and the context root was 'CtxtRoot', then the web application is comparing '/jsp/' to all of the servlet mappings. Requests for static resources in WEB-INF will always be refused, but servlets and filters can and do return data from WEB-INF.
7. I'm going to proceed assuming the request was sent to the spring DispatcherServlet, it receives the request, and considers everything after the servlet path. Spring's DispatcherServlet looks for a Controller whose path matches the path after the servlet path. The servlet path is basically what you put in the servlet mapping in your web xml. Let me give an example, let's say you have a web app whose context is 'app', and it has a spring MVC servlet whose servlet mapping is '/mvc', and a controller that handles the path 'sales', then you could reach that controller with http://localhost:8080/app/mvc/sales.
8. If the DispatcherServlet cannot find a Controller, I believe it treats the incoming request as if it was returned by a controller, so if the sub-path is 'sales', then it would pass that as an argument to the view resolver. If it can't find it, then the server returns a not found error.
9. Typically the Controller returns a string when it's done, which is the path to a resource. It could return 'popular' as a string. Spring then forwards this to the ViewResolver, and I will assume you're using the InternalResourceViewResolver. It will look at the prefix and suffix, and basically wrap those around what it was given. So if the prefix is '/WEB-INF/views/', the suffix is '.jsp', and the argument is 'popular', then it will look for a resource at '/WEB-INF/views/popular.jsp'. It is literally just concatenating those strings to make a path. The path is ALWAYS relative to the web application root here. If the path produced is a jsp file, it will be interpreted before being returned.
10. Then it is finally returned to the user.

From your example you were requesting localhost:8080/ContextRoot/jsp/fileName, so it looks like 'CtxRoot' is the context root, your servlet's path is '/', so it should pass whatever is after that to a controller. By the time the DispatcherServlet receives the request, it is searching for a controller that handles 'jsp' as a path. Since you had none, it decided to treat that as a resource path. It used the view resolver and formed the path /WEB-INF/jsp/jsp/fileName.jsp, which obviously does not exist.

Let's assume that you had instead requested localhost:8080/ContextRoot/fileName, the request would reach the DispatcherServlet, it would find no Controller that handles 'fileName' as a path, and hence would treat it as a resource. It would form the path /WEB-INF/jsp/fileName.jsp, and that would return the result.

However, your web xml was not configured to initialize spring. So your web application was actually treating every one of your requests as if they were for a resource relative to the web application root. I believe that if you had made that request with Spring properly initialized, it might have worked.

**Add your Dispatcher Servlet class name to implement Front Controller Pattern in WEB.XML**

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<!-- Url should be the url of the dispatcher.xml -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/Dispatcher-servlet.xml </param-value>

</context-param> :

<servlet>

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

<servlet-class> org.springframework.web.servlet.DispatcherServlet</servlet-class>

</servlet>

<servlet-mapping>

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

<url-pattern>/</url-pattern>

</servlet-mapping>

**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 ViewResolver interface**

As an example, when using JSP for a view technology you can use the UrlBasedViewResolver. This view resolver translates a view name to a URL and hands the request over to the RequestDispatcher to render the view.

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

<property name=*"prefix"*>

<value>/WEB-INF/views/</value>

</property>

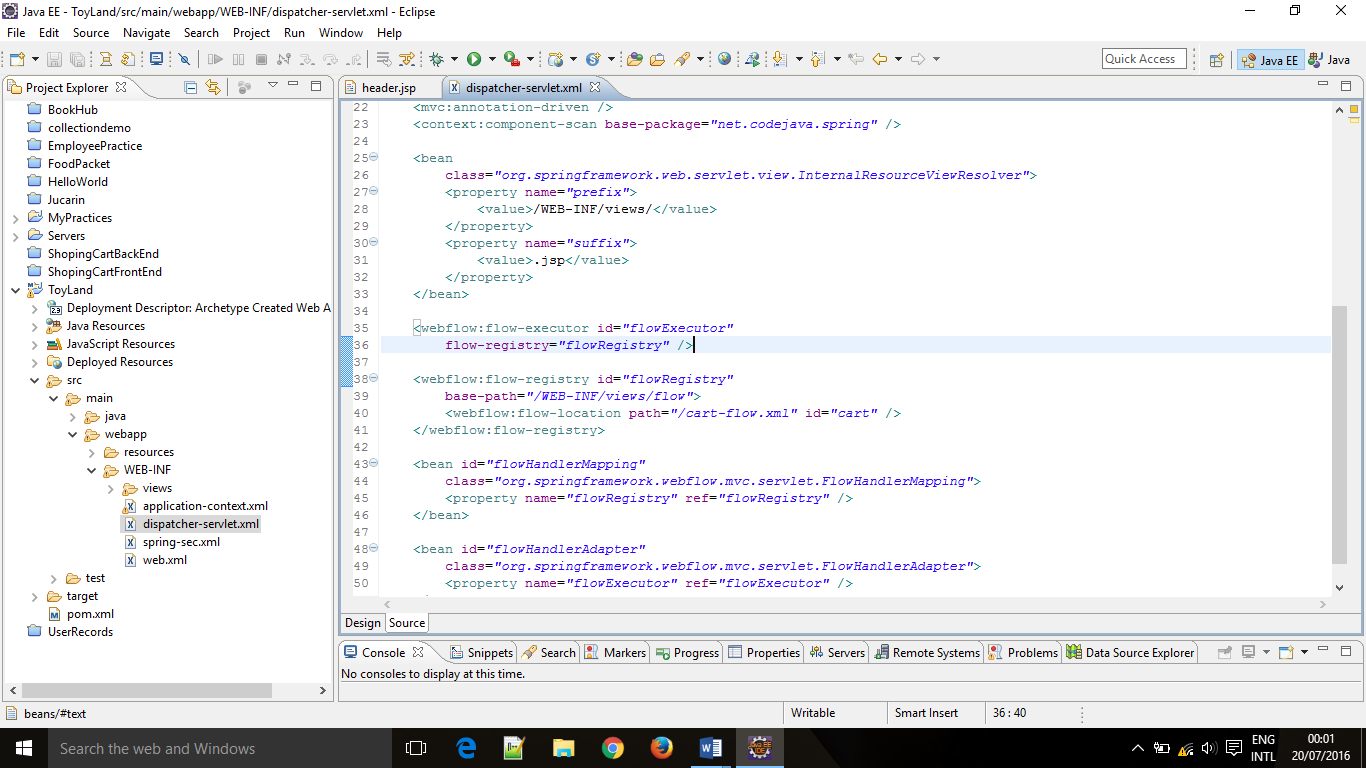
<property name=*"suffix"*>

<value>.jsp</value>

</property>

</bean>

When returning index as a viewname, this view resolver will hand the request over to the RequestDispatcher that will send the request to /WEB-INF/view/index.jsp.



**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.

**Advantages of Spring MVC Framework-**

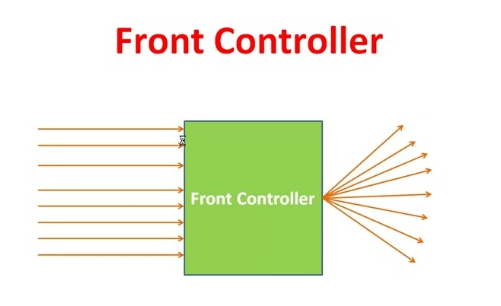
* Annotation based configuration

(i.e. you may reduce the metadata file or less of configuration).

* Supports to plug with other MVC frameworks like Struts, Struts2, WebWorks etc.
* Flexible in supporting different view types like JSP, velocity, XML, PDF, Tiles etc.

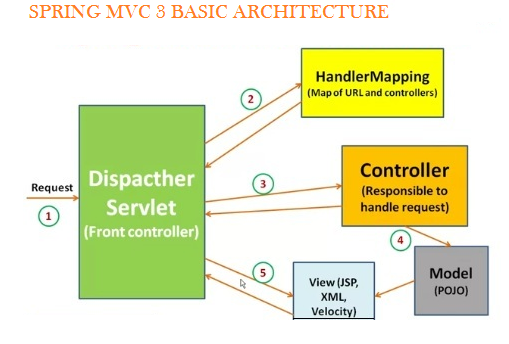
**Front Controller:** Front Controller is very important component one which route the all the requests into framework control that means whenever requests land on different controllers it queues that request to the controller of framework without this MVC framework will not may be able to take control 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.



In Spring 3 MVC framework Dispatcher Servlet access Front Controller which handles all coming requests and queues for forward to the different controller.

1. Whenever request lands the dispatcher servlet consult with HandlerMapping

(HandlerMapping- is a component which have the map of URL and Controller which need to be invoked for that particular request which lands with URL)

2. then Dispatcher servlet has information about which is controller need to be invoked

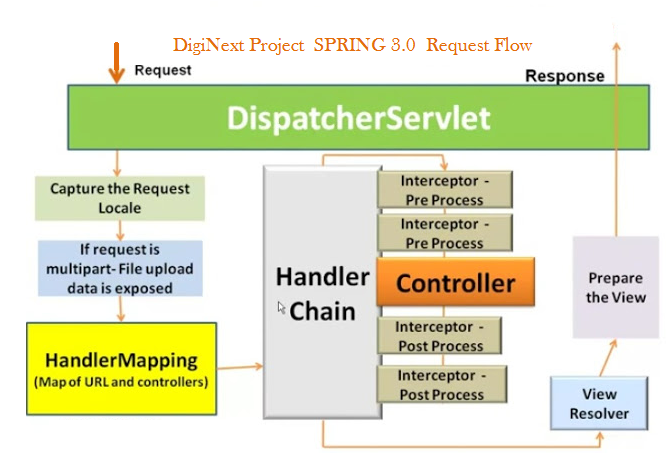
3. then that controller will be invoked

4. and Controller can request the model for some information (about some DAO, Service layer or Data in POJO, or data in database using business logic)

5. once process has been done then dispatcher servlet get the response then dispatcher servlet will get view resolver to build the view and view resolver look out what view has being configured it has been JSP, Velocity, XML etc. based this configuratin view has been prepared and the information from model i.e. POJO it will be put on the view and response will be send back to browser.

**Spring 3 MVC Request Flow:**

1. Request lands to Front Controller i.e. DispatcherServlet
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 HandlerMapping for which Controller to be invoked and then responsibility is given to the Handler Chain
5. 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 to similar to the filters that help to separate the pre-process logic and post-process logic.
6. After process of pre-process interceptor return to the controller process the post-process logic.
7. Then return to the view resolver prepared the view based on your configuration decide the which configuration (JSP, Velocity, PDF etc.) to be invoked.
8. After choosing view technology prepare the view and return the response back to the client.



**Required Spring 3.0 MVC Configuration:**

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

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

<web-app>

<display-name>Archetype Created Web Application</display-name>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>

/WEB-INF/dispatcher-servlet.xml

/WEB-INF/application-context.xml

/WEB-INF/spring-sec.xml

</param-value>

</context-param>

<filter>

<filter-name>springSecurityFilterChain</filter-name>

<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>

</filter>

<filter-mapping>

<filter-name>springSecurityFilterChain</filter-name>

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

</filter-mapping>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<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>dispatcher</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

**Step 2:** Configure the spring3-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"*

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

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

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

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

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

*http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-3.0.xsd "*>

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*>

<property name=*"driverClassName"* value=*"org.h2.Driver"*></property>

<property name=*"url"* value=*"jdbc:h2:tcp://localhost/~/toysdb"*></property>

<property name=*"username"* value=*"player"*></property>

<property name=*"password"* value=*"password"*></property>

</bean>

**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/hello.jsp .

**@Controller:**

* Used at the class level
* Tells the spring framework that the marked class acts as a controller.

@Controller

public class EmployeeController{

}

**@RequestMapping:**

* Can be used at the class level and method level in controllers.
* Arguments
* URL[]
* HTTP Methods[]-GET, POST, DELETE, TRACE, OPTIONS, HEAD, PUTS. Defaults method supported is GET
* params[]-used to check if a request parameter matches with a value and only if the conditions passes the method or controller processes the request. (eg. @RequestMapping params="myName=guest" )
* headers[]-used to check if a request header matches with a value and only if the condition passes the method or controller processes the request (eg. @RequestMapping headers="myheader=guestHadder")

**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.

@Controller

public class ToyController{

@RequestMapping("add", method = RequestMethod.POST,value=/product)

public String create Product(){}//appcontext/employee/add

@RequestMapping("delete", method = RequestMethod.GET,value=/product)

public String deleteProduct(){}

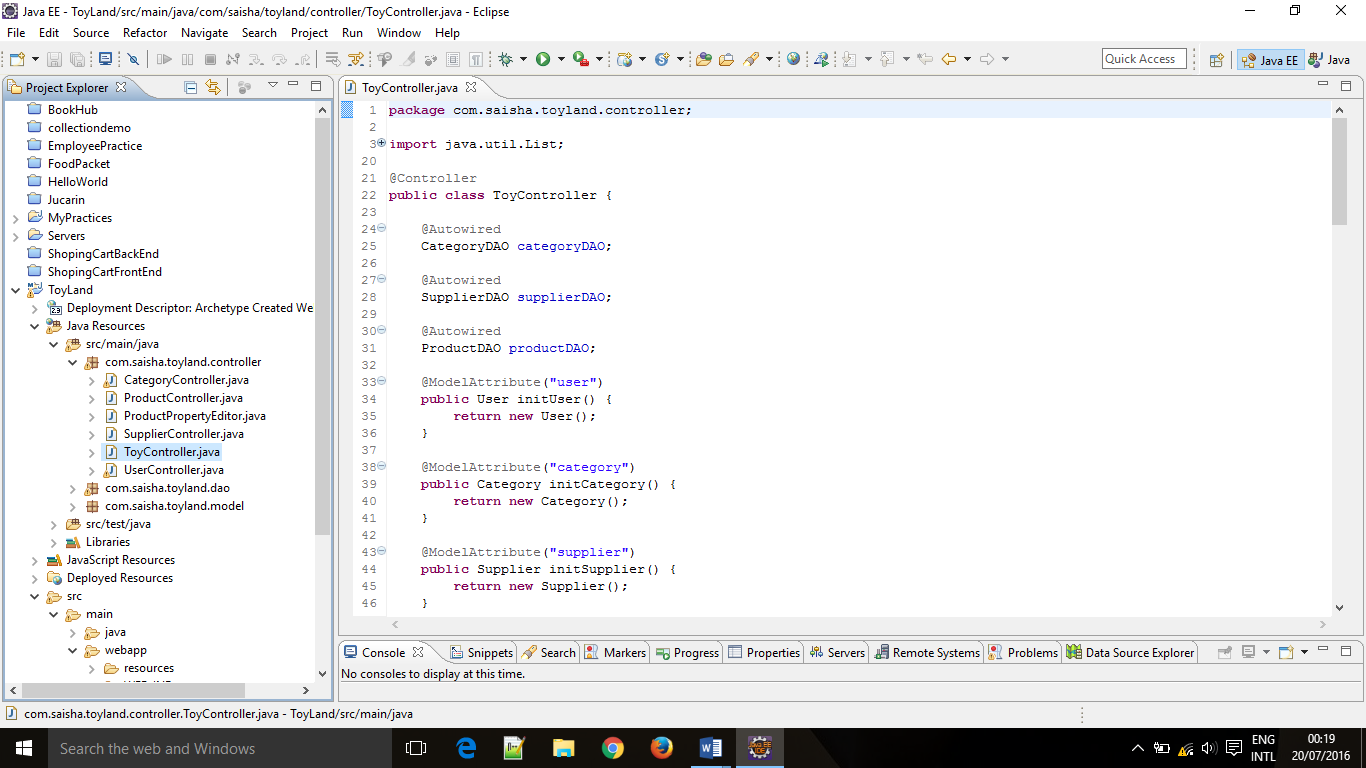
@RequestMapping("details",value=/employee )

public String getEmployeeDetails(){}

}

The value attribute indicates the URL to which the handler method is mapped and the method attribute defines the service method to handle HTTP GET request. There are following important points to be noted about the controller defined above:

* You will defined required business logic inside a service method. You can call another methods inside this method as per requirement.
* Based on the business logic defined, you will create a model within this method. You can setter different model attributes and these attributes will be accessed by the view to present the final result. This example creates a model with its attribute "message".
* A defined service method can return a String which contains the name of the view to be used to render the model.



Web Ref : <https://docs.spring.io/docs/Spring-MVC-step-by-step/part2.html>

**Spring MVC Form Model Implementation**

Create a Java class Product and under the com.saisha.toyland.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

package com.saisha.toyland.model;

import java.io.Serializable;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.FetchType;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.ManyToOne;

import javax.persistence.Transient;

import org.springframework.web.multipart.MultipartFile;

@Entity

public class Product implements Serializable{

@Id

@Column(name = "Product\_ID")

@GeneratedValue(strategy = GenerationType.AUTO)

private int productId;

@Column(name = "Product\_Name")

private String productName;

@Column(name = "Product\_Description")

private String productDescription;

@Column(name = "Price")

private double productPrice;

private String manufacturer;

@ManyToOne(fetch=FetchType.LAZY)

private Category category;

@ManyToOne(fetch=FetchType.LAZY)

private Supplier supplier;

private static final long serialVersionUID = 1L;

@Transient

private MultipartFile img;

public Product() {

}

public Supplier getSupplier() {

return supplier;

}

public void setSupplier(Supplier supplier) {

this.supplier = supplier;

}

public Category getCategory() {

return category;

}

public void setCategory(Category category) {

this.category = category;

}

public int getProductId() {

return productId;

}

public void setProductId(int productId) {

this.productId = productId;

}

public String getProductName() {

return productName;

}

public void setProductName(String productName) {

this.productName = productName;

}

public String getProductDescription() {

return productDescription;

}

public void setProductDescription(String productDescription) {

this.productDescription = productDescription;

}

public double getProductPrice() {

return productPrice;

}

public void setProductPrice(double productPrice) {

this.productPrice = productPrice;

}

public String getManufacturer() {

return manufacturer;

}

public void setManufacturer(String manufacturer) {

this.manufacturer = manufacturer;

}

public MultipartFile getImg() {

return img;

}

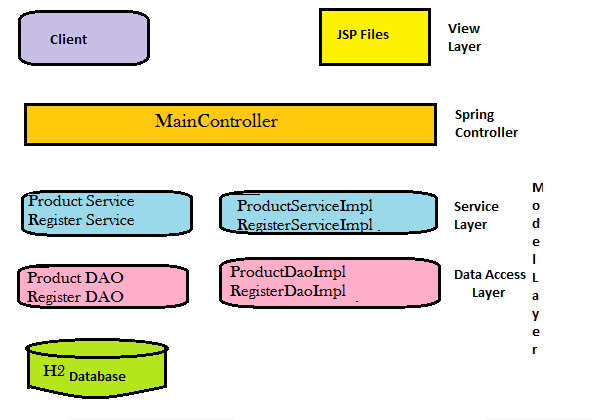
public void setImg(MultipartFile img) {

this.img = img;

}

}

**CRUD Operations Spring 3 Hibernate Integration**

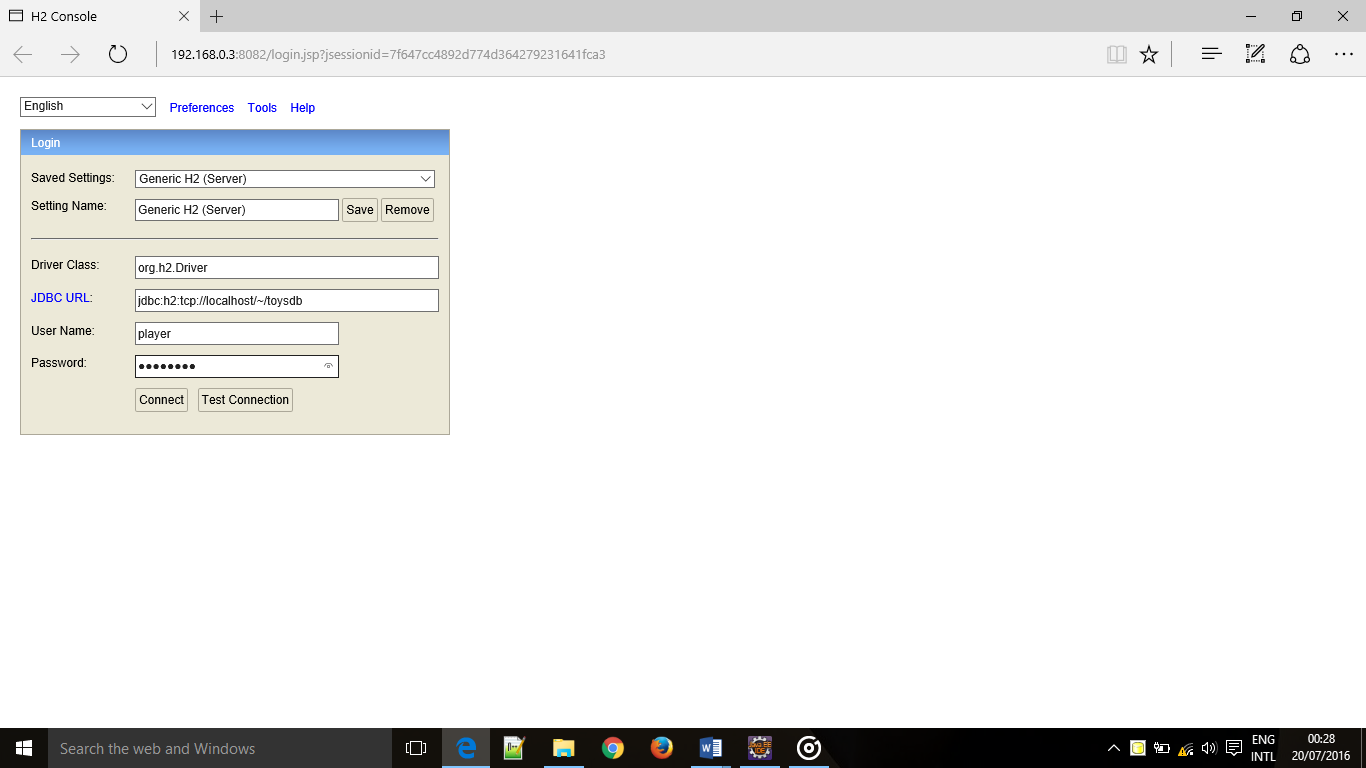


**Files used**

* 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 logined into Server and not Client



Step 3 : The following Dependency to be added

<!-- h2 dependency -->

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<version>1.4.192</version>

</dependency>

<!-- /h2 dependency -->

<!-- hibernate dependency -->

<dependency>

<groupId>org.hibernate.javax.persistence</groupId>

<artifactId>hibernate-jpa-2.1-api</artifactId>

<version>1.0.0.Final</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>4.0.1.Final</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-validator</artifactId>

<version>5.0.1.Final</version>

</dependency>

<dependency>

<groupId>commons-dbcp</groupId>

<artifactId>commons-dbcp</artifactId>

<version>1.2.2</version>

</dependency>

<!-- /hibernate dependency -->

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.
* nullable 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.

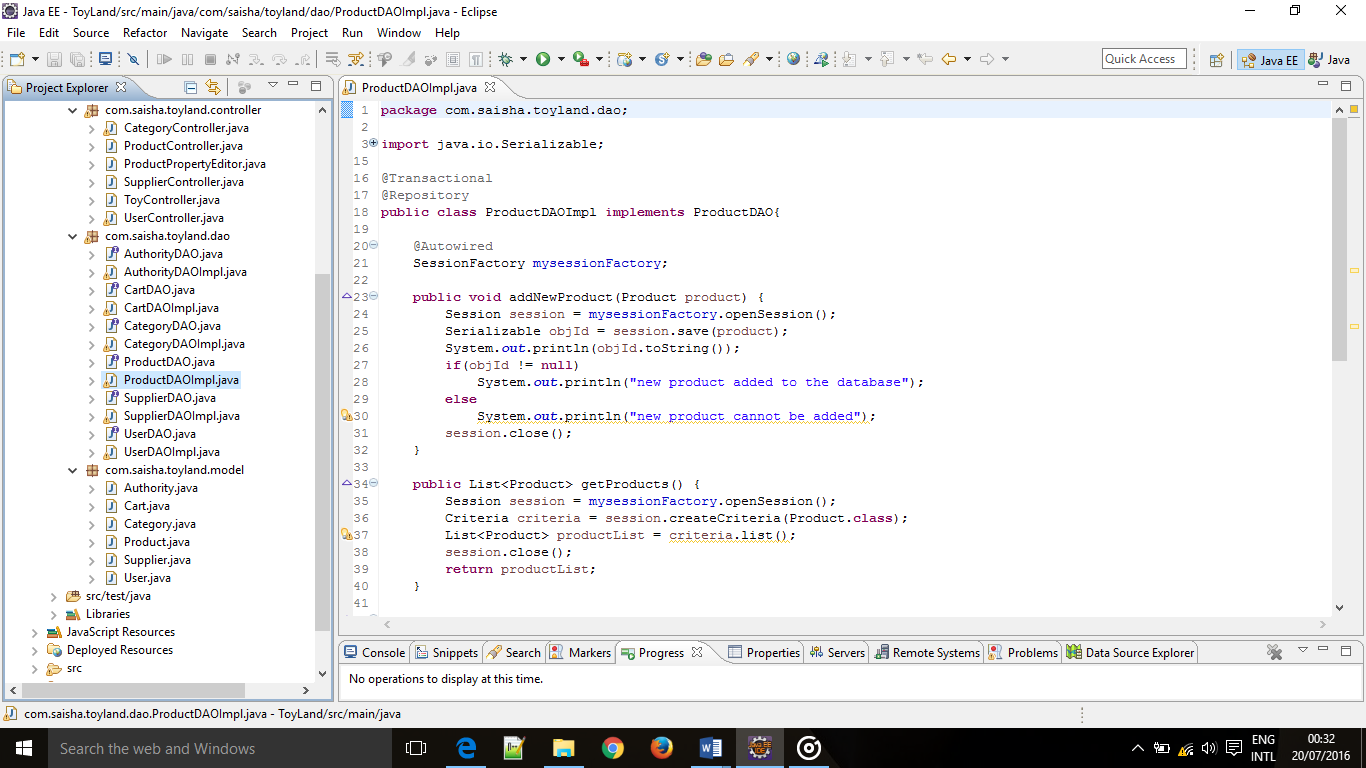
**Create DAO Class**

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.

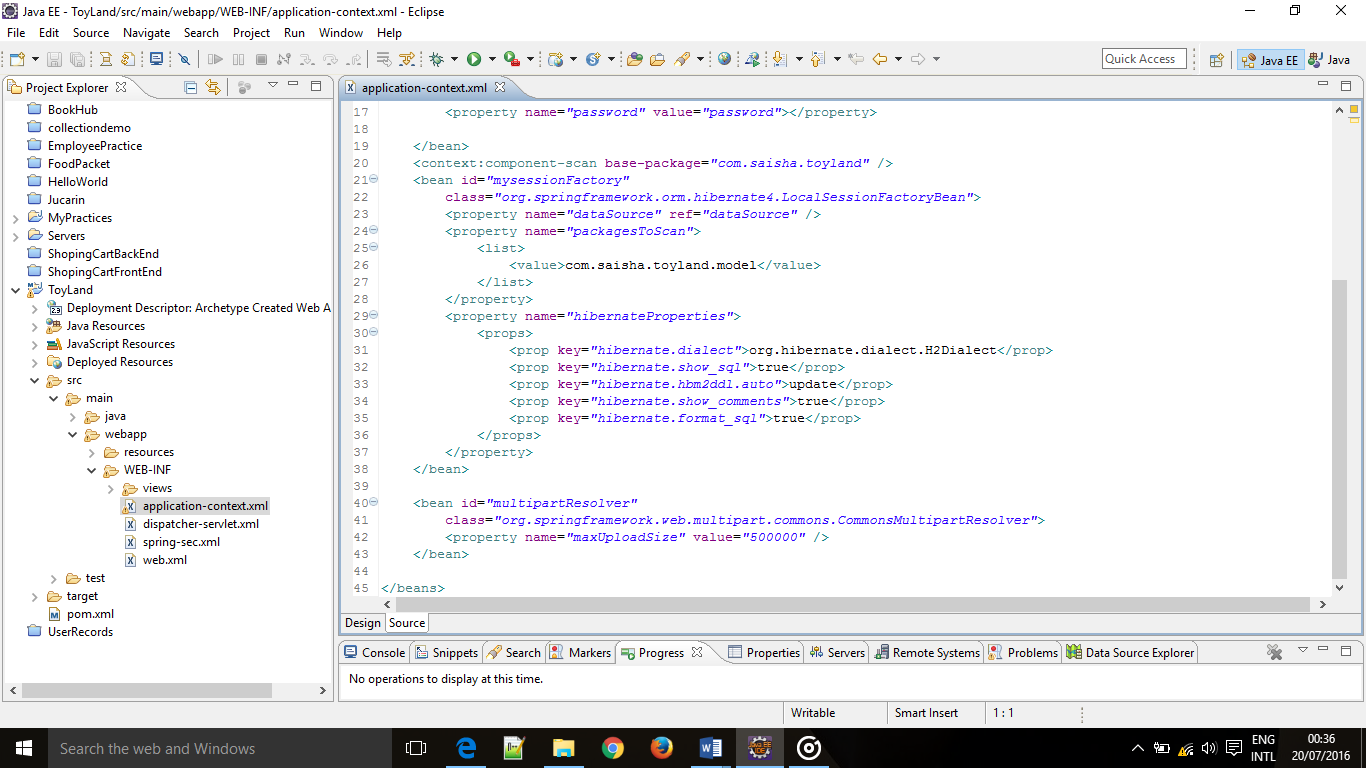


**SessionFactory setup in a Spring container**

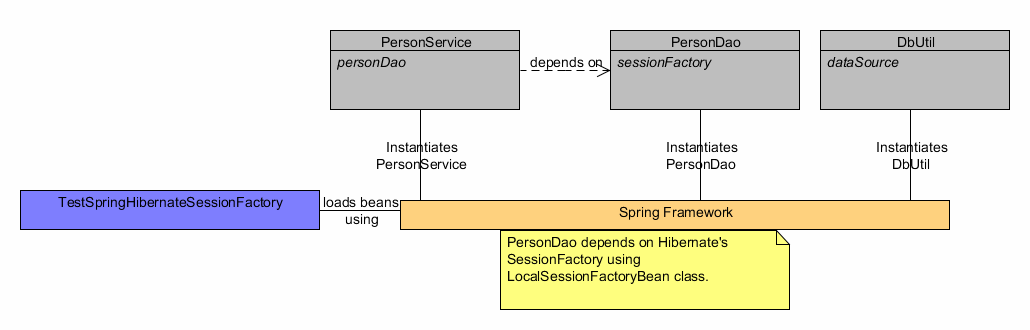
We can directly obtain a Hibernate session using the getSession() method of the Hibernate SessionFactory. Wire a LocalSessionFactoryBean into the DAO as a SessionFactory object and then use the getSession method of this object to obtain the session.

To avoid tying application objects to hard-coded resource lookups, you can define resources such as a JDBC DataSource or a Hibernate SessionFactory as beans in the Spring container. Application objects that need to access resources receive references to such predefined instances through bean references

The following excerpt from an XML application context definition shows how to set up a JDBC DataSource and a Hibernate SessionFactory on top of it:



**Implementing DAOs based on plain Hibernate API**



Hibernate has a feature called contextual sessions, wherein Hibernate itself manages one current Session per transaction. This is roughly equivalent to Spring’s synchronization of one Hibernate Session per transaction. A corresponding DAO implementation resembles the following example, based on the plain Hibernate API.

The product DAO follows the dependency injection pattern: it fits nicely into a Spring MVC container, just as it would if coded against Spring’s HibernateTemplate. Of course, such a DAO can also be set up in plain Java (for example, in unit tests). Simply instantiate it and call setSessionFactory(..) with the desired factory reference. As a Spring bean definition, the DAO would resemble the following:

The main advantage of this DAO style is that it depends on Hibernate API only; no import of any Spring class is required. This is of course appealing from a non-invasiveness perspective, and will no doubt feel more natural to Hibernate developers.

However, the DAO throws plain HibernateException (which is unchecked, so does not have to be declared or caught), which means that callers can only treat exceptions as generally fatal - unless they want to depend on Hibernate’s own exception hierarchy. Catching specific causes such as an optimistic locking failure is not possible without tying the caller to the implementation strategy. This trade off might be acceptable to applications that are strongly Hibernate-based and/or do not need any special exception treatment.

Fortunately, Spring’s LocalSessionFactoryBean supports Hibernate’s SessionFactory.getCurrentSession() method for any Spring transaction strategy, returning the current Spring-managed transactional Session even with HibernateTransactionManager. Of course, the standard behavior of that method remains the return of the current Session associated with the ongoing JTA transaction, if any. This behavior applies regardless of whether you are using Spring’s JtaTransactionManager, EJB container managed transactions (CMTs), or JTA.

In summary: you can implement DAOs based on the plain Hibernate API, while still being able to participate in Spring-managed transactions.

**Declarative transaction demarcation**

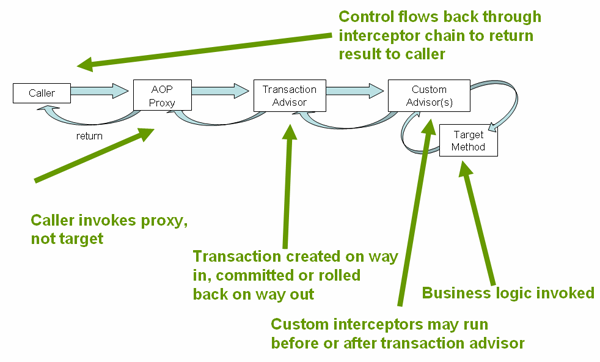
Spring’s declarative transaction support, which enables you to replace explicit transaction demarcation API calls in your Java code with an AOP transaction interceptor. This transaction interceptor can be configured in a Spring container using either Java annotations or XML. This declarative transaction capability allows you to keep business services free of repetitive transaction demarcation code and to focus on adding business logic, which is the real value of your application.

<bean id=*"mysessionFactory"*

class=*"org.springframework.orm.hibernate4.LocalSessionFactoryBean"*>

<property name=*"dataSource"* ref=*"dataSource"* />

<property name=*"packagesToScan"*>



Now we have to annotate the service layer with @Transactional annotations and instruct the Spring container to find these annotations and provide transactional semantics for these annotated methods.

**Spring Security Implementation**

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

**Update project dependencies in pom.xml**

To get start Add the following Maven dependency in POM.XML

**spring-security-core**

It contains core authentication and access-contol classes and interfaces.

**spring-security-web**

It contains filters and related web-security infrastructure code. It also enable URL based security which we are going to use in this demo.

**spring-security-config**

It contains the security namespace parsing code. You need it if you are using the Spring Security XML file for configuration.

**spring-security-taglibs**

It provides basic support for accessing security information and applying security constraints in JSPs.

So the pom.xml file will be updated with:

<!-- spring security dependencies -->

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-core</artifactId>

<version>4.0.0.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-config</artifactId>

<version>4.0.0.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-taglibs</artifactId>

<version>4.0.0.RELEASE</version>

</dependency>

<!-- /spring security dependencies -->

**The scope of Security in this project is to:**

* Only authorized user should be able to access edit employee screen.
* Unauthorized users should be presented with login screen.
* Successful credentials should forward to edit employee screen.
* Unsuccessful credentials should forward to access denied screen.
* There should be a link for logout of the application.

**Toyland complete web.xml\_\_**

<!DOCTYPE web-app PUBLIC

"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"

"http://java.sun.com/dtd/web-app\_2\_3.dtd" >

<web-app>

<display-name>Archetype Created Web Application</display-name>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>

/WEB-INF/dispatcher-servlet.xml

/WEB-INF/application-context.xml

/WEB-INF/spring-sec.xml

</param-value>

</context-param>

<filter>

<filter-name>springSecurityFilterChain</filter-name>

<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>

</filter>

<filter-mapping>

<filter-name>springSecurityFilterChain</filter-name>

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

</filter-mapping>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<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>

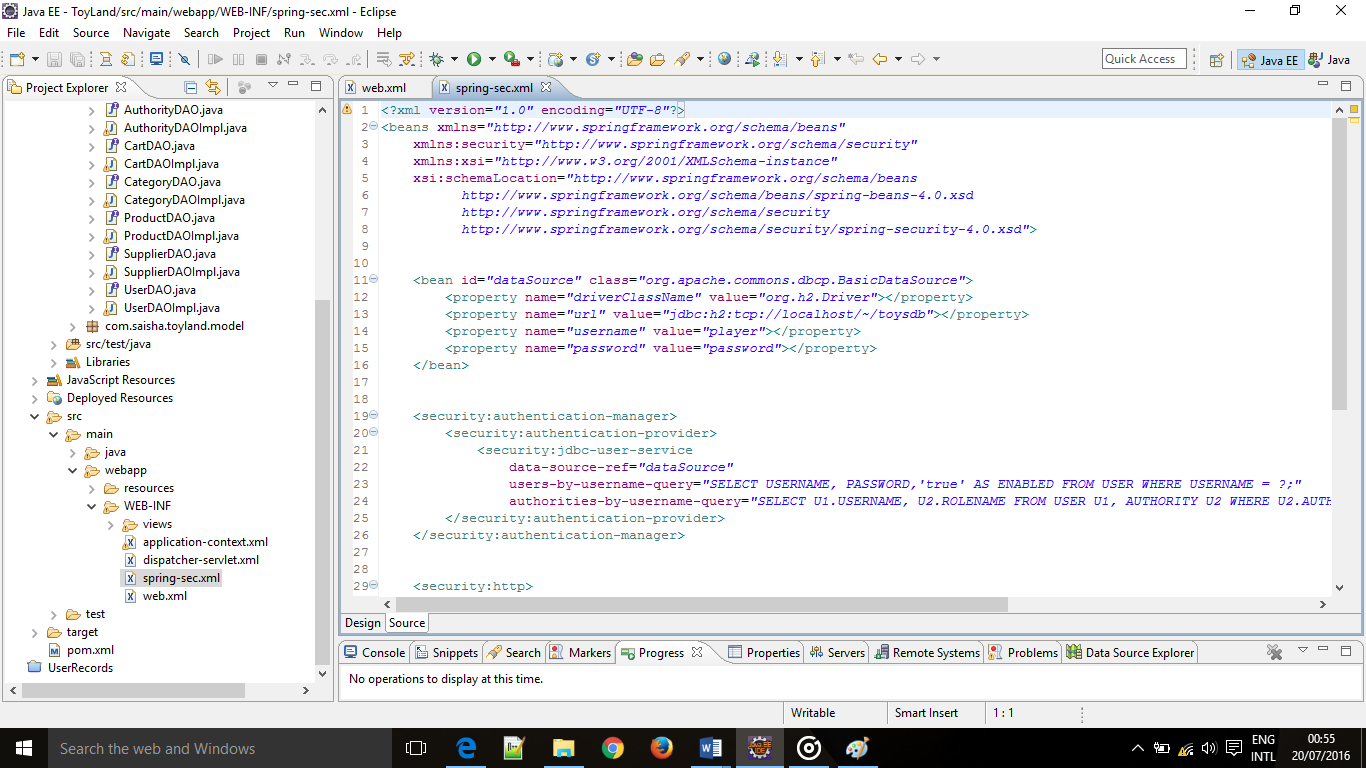
<servlet-name>dispatcher</servlet-name>

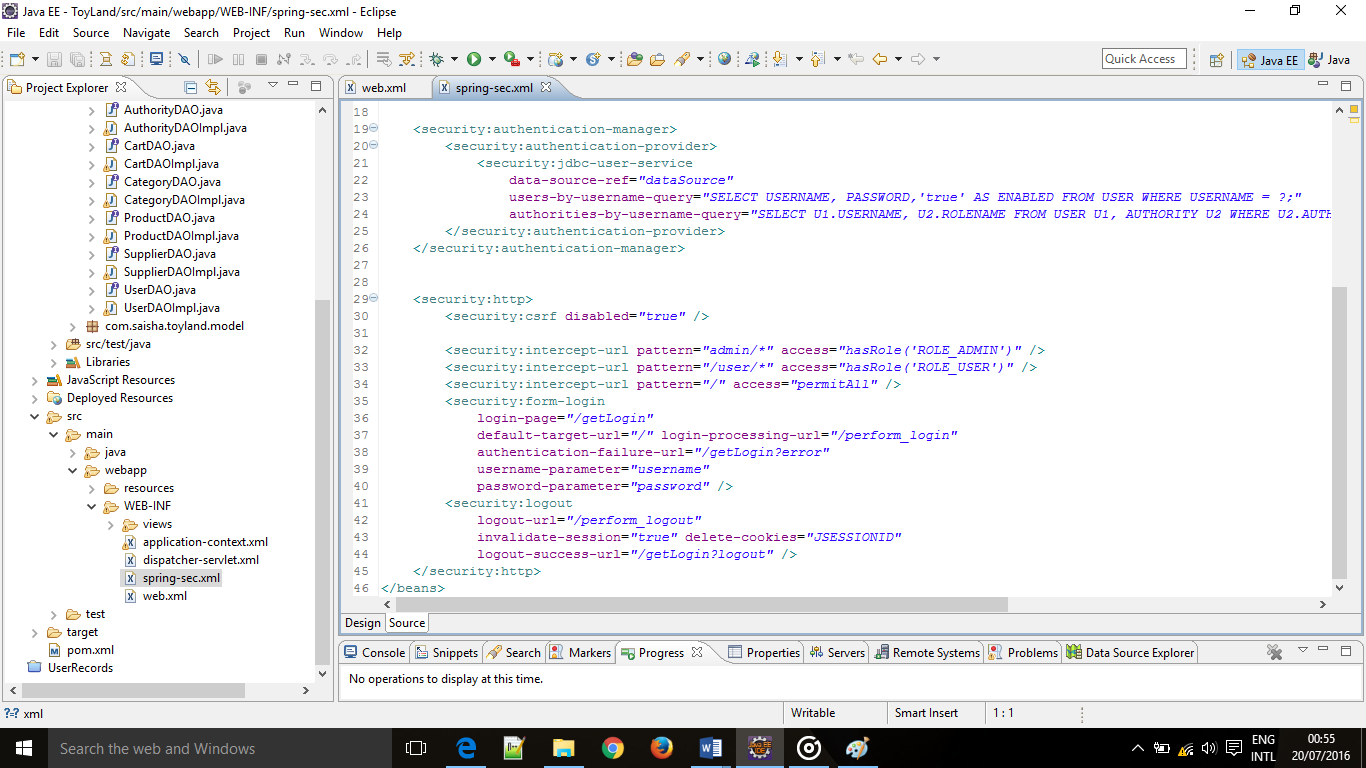
<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

**Add security configuration in spring-sec.xml**





**use-expressions:** It is here to use expressions to secure individual URLs. These expressions can be e.g. hasRole([role]), hasAnyRole([role1,role2]), permitAll, denyAll etc.

**intercept-url:** This will match the requested url pattern from request and will decide what action to take based on access value.

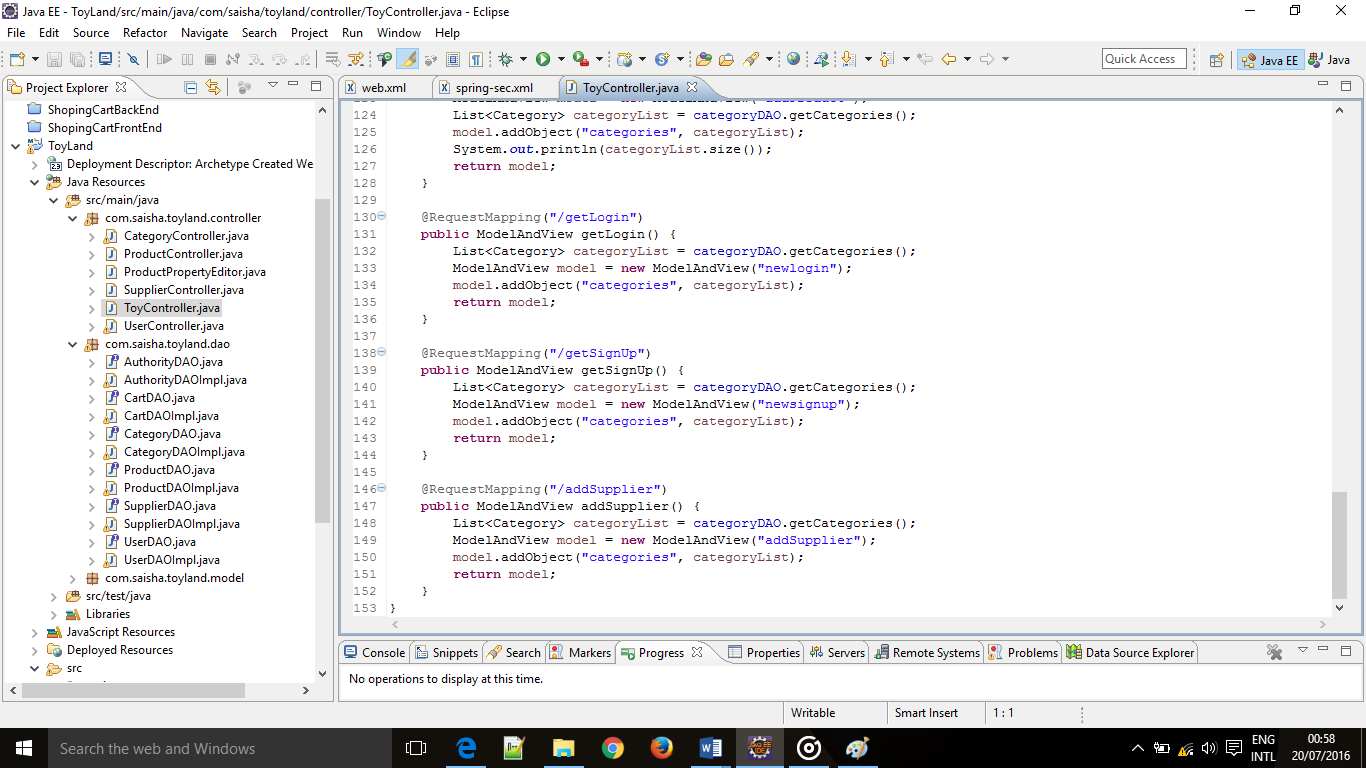
**form-login:** This will come into picture when user will try to access any secured URL. A login page mapped to “login-page” attribute will be served for authentication check. If not provided, spring will provide an inbuilt login page to user. It also contains attribute for default target if login success, or login failure due to invalid user/password match.

**logout:** This will help to find the next view if logout is called in application.

I am using XML based user service i.e. I will not go to database for password validation rather i have stored username/password combination in configuration file itself. To use this king of setup, authentication-manager is setup with inline in-built user details service. In more real time applications, this is going to be some user service fetching data from remote database.

**Update the controller**

The additional URLs are /login, /logout and /accessdenied has to be added into the ToyController



**Add related views**

We have now configured our application with security configuration and controller handlers. Its time to write the views which are essentially JSP files. Most important addition in jsp files is login.jsp file. This file have the form which contains text boxes for username and password field.

**Implementing Validation**

We have created Spring MVC application that take user input and checks the input using standard validation annotations. Now we have to display the error message on the screen so the user can re-enter a valid input.

Add annotation in the Bean Entity in our project it is Product.java and Register.java



Validating data is a common task that occurs throughout all application layers, from the presentation to the persistence layer. Often the same validation logic is implemented in each layer which is time consuming and error-prone. To avoid duplication of these validations, developers often bundle validation logic directly into the domain model, cluttering domain classes with validation code which is really metadata about the class itself.

**@NotNull**

The @NotNull Annotation is, actually, an explicit contract declaring the following:

* A method should not return null.
* A variable (like fields, local variables, and parameters) cannot hold null value.

@NotNull, @NotBlank or @NotEmpty they doesn't word please ensure you have added the following dependency

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-validator</artifactId>

<version>5.1.0.Final</version>

</dependency>

**Uploading Files with Multi-Part**

Now our project need image files to be uploaded into our site. For this we need to understand the process of creating a server application that can receive multi-part file uploads.

**Create a configuration class**

To upload files with Servlet 3.0 containers, you need to register a MultipartConfigElement class (which would be <multipart-config> in web.xml). Thanks to Spring Boot, that bean is already registered and available! All you need to get started with this application is the following, empty configuration setup.

src/main/java/hello/Application.java

package hello;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class Application {

}

You will soon add a Spring MVC controller, which is why you need @SpringBootApplication. Spring Boot automatically finds @Controller-marked classes and registers them with the application context.

As part of auto-configuring Spring MVC, Spring Boot will create a MultipartConfigElement bean and make itself ready for file uploads.

**Using a MultipartResolver with Servlet 3.0**

To use Servlet 3.0 based multipart parsing, you need to :

Mark the DispatcherServlet with a “multipart-config” section in web.xml or with javax.servlet.MultipartConfigElement in programmatic Servlet registration.

Next, when multipart parsing has been enabled in one of the above mentioned ways, then add the StandardServletMultipartResolver to your Spring configuration as follows:

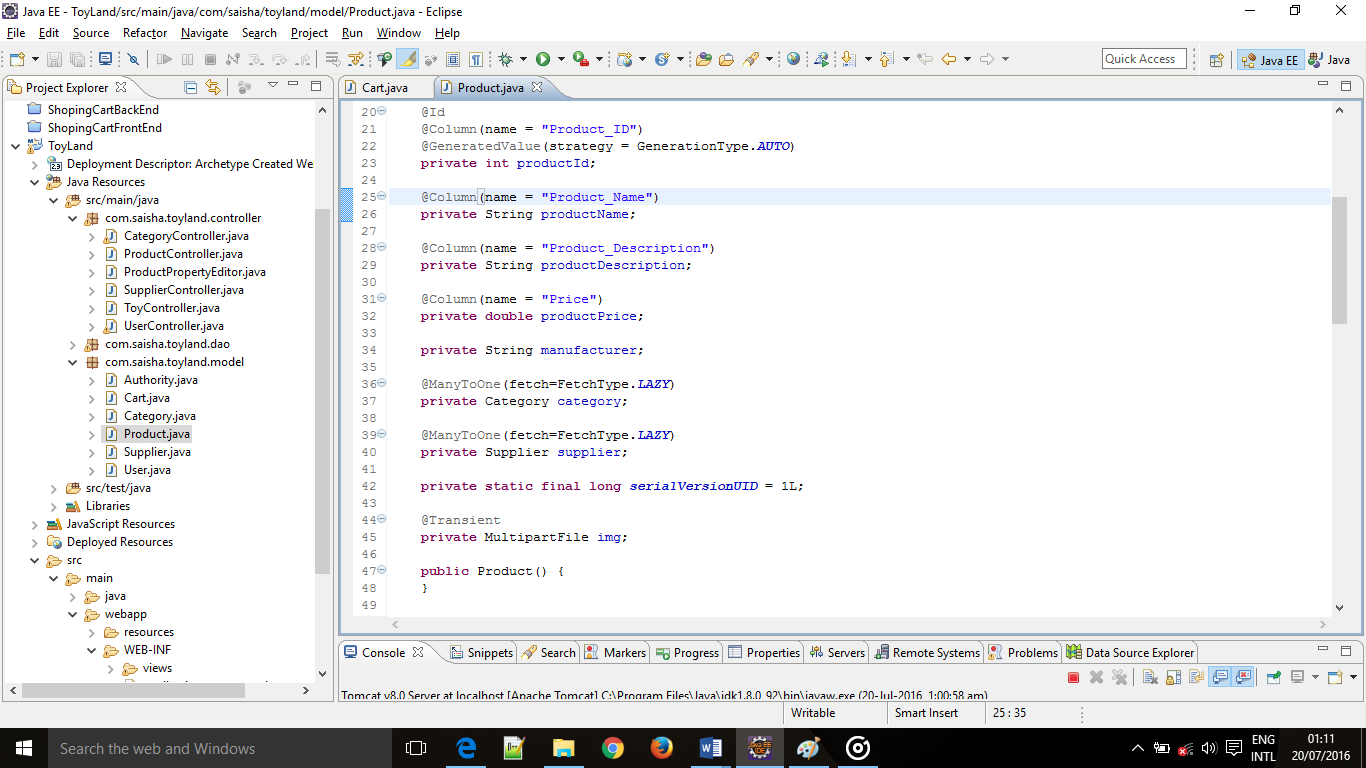
<bean id="multipartResolver"

class="org.springframework.web.multipart.support.StandardServletMultipartResolver">

</bean>

**Create a file upload controller**

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.



**Designing Front End :**

In the spring MVC form which already created update the following changes.

* <form:form action="add" enctype="multipart/form-data" method="post">

**Add multipart/form-data attribute** in Form tag which allows entire files to be included in the data.

* <td colspan="2">Upload Image : <input type="file" name="file" required="required" /> </td>

**Add input type File** to create a File Browser control.

Run the project and check….

**Hibernate Validator**

Welcome to Hibernate Validator. The following paragraphs will guide you through the initial steps required to integrate Hibernate Validator into your application.

**Project about:** In order to use Hibernate Validator within a Maven project, simply check whether you added the following dependency to your pom.xml:

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-validator</artifactId>

<version>5.0.1.Final</version>

</dependency>

**Steps**:

1. Load the view web form
2. Adding Constraints on domain object like @NotNull @Size etc.,
3. Add validation to handler Method
   1. @Valid annotation
   2. hasErrors method
4. Display back the errors on jsp, using spring form errors tag lib

**Data :** For this example we will use a bean in our project named User

**package** com.saisha.toyland.model;

**import** java.io.Serializable;

**import** javax.persistence.Entity;

**import** javax.persistence.FetchType;

**import** javax.persistence.GeneratedValue;

**import** javax.persistence.Id;

**import** javax.persistence.ManyToOne;

**import** javax.validation.constraints.Size;

**import** org.hibernate.validator.constraints.Email;

**import** org.hibernate.validator.constraints.NotEmpty;

**import** javax.persistence.GenerationType;

@Entity

**public** **class** User **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** String userid;

**private** String firstname;

**private** String lastname;

**private** String username;

**private** String floor;

**private** String buildingName;

**private** String landmark;

**private** String areaName;

**private** String city;

**private** String contactNumber;

**private** **int** pincode;

**private** String state;

@NotEmpty

@Email

**private** String email;

@NotEmpty

@Size(min = 6, max = 15, message = "password must contain atleast 6 characters and maximum 15 characters")

**private** String password;

@Id

@GeneratedValue (strategy=GenerationType.***AUTO***)

**public** String getUserid(){

**return** userid;

}

**private** Authority authority;

**public** **void** setUserid(String userid){

**this**.userid = userid;

}

**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 getUsername() {

**return** username;

}

**public** **void** setUsername(String username) {

**this**.username = username;

}

**public** String getEmail() {

**return** email;

}

**public** **void** setEmail(String email) {

**this**.email = email;

}

**public** String getPassword() {

**return** password;

}

**public** **void** setPassword(String password) {

**this**.password = password;

}

@ManyToOne(fetch=FetchType.***LAZY***)

**public** Authority getAuthority() {

**return** authority;

}

**public** **void** setAuthority(Authority authority) {

**this**.authority = authority;

}

**public** String getFloor() {

**return** floor;

}

**public** **void** setFloor(String floor) {

**this**.floor = floor;

}

**public** String getBuildingName() {

**return** buildingName;

}

**public** **void** setBuildingName(String buildingName) {

**this**.buildingName = buildingName;

}

**public** String getLandmark() {

**return** landmark;

}

**public** **void** setLandmark(String landmark) {

**this**.landmark = landmark;

}

**public** String getAreaName() {

**return** areaName;

}

**public** **void** setAreaName(String areaName) {

**this**.areaName = areaName;

}

**public** String getCity() {

**return** city;

}

**public** **void** setCity(String city) {

**this**.city = city;

}

**public** String getContactNumber(){

**return** contactNumber;

}

**public** **void** setContactNumber(String contactNumber){

**this**.contactNumber = contactNumber;

}

**public** **int** getPincode() {

**return** pincode;

}

**public** **void** setPincode(**int** pincode) {

**this**.pincode = pincode;

}

**public** String getState() {

**return** state;

}

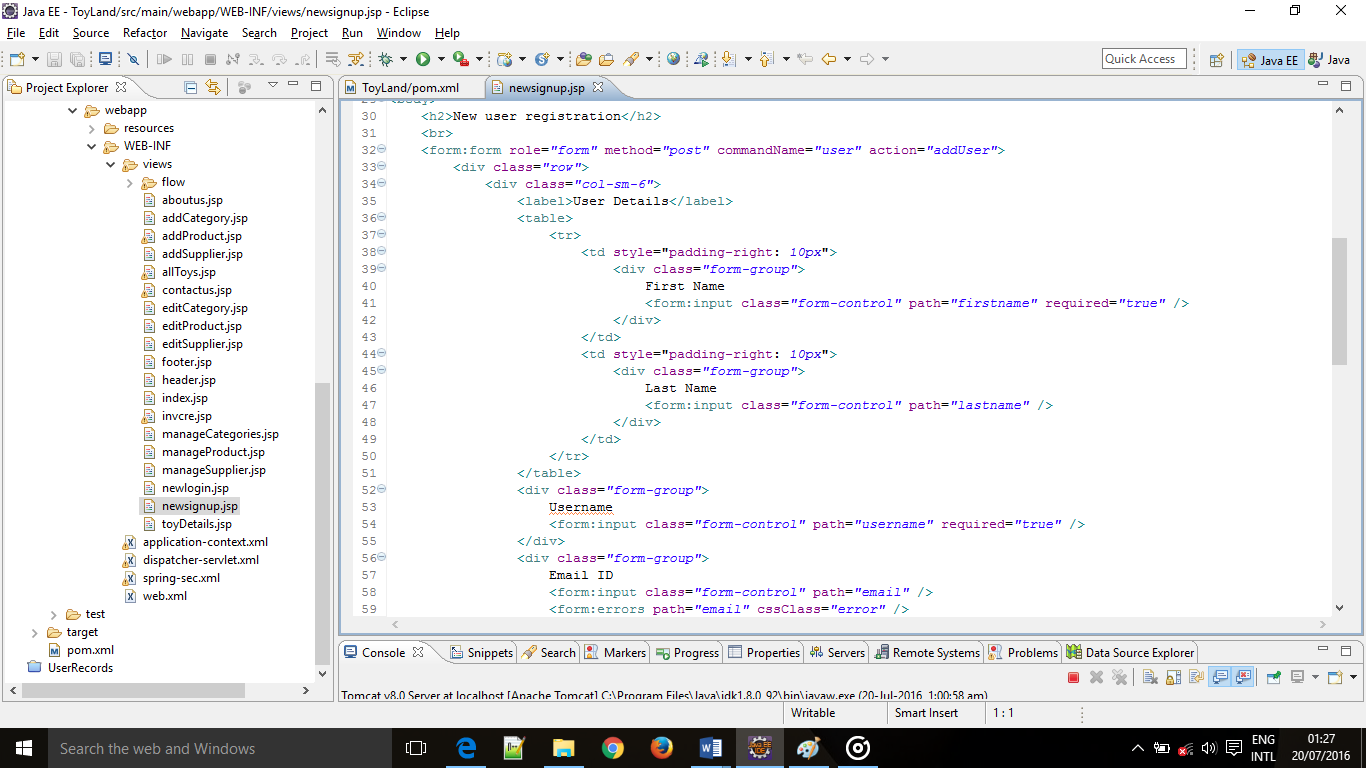
**public** **void** setState(String state) {

**this**.state = state;

}

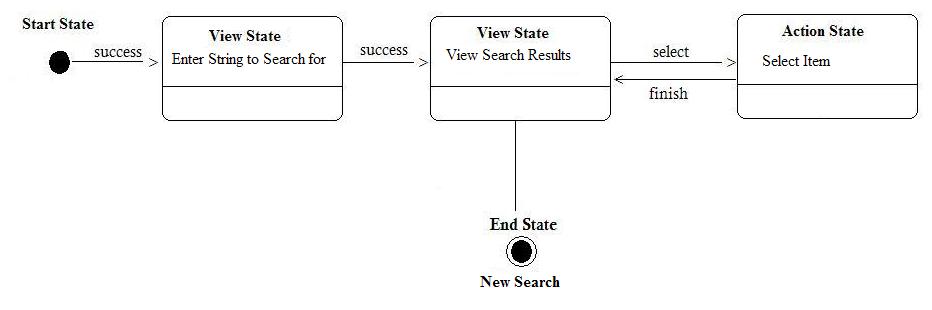
}

**Front End .jsp file design**



# Spring Web Flow

You must have found the web applications are getting more and more complex and to maintain them is getting even harder. You now have to write or understand and maintain a large and complex page flows (flow of the application). If you are bugged down with creating such a complex application the answer lies in using Spring Web Flow. Spring Web Flow (SWF) is a component of the Spring Framework’s web stack focused on the definition and execution of user interface (UI) flow within a web application. It is a module that allows you to make logical flows of your web application.



**Why Spring Web Flow**

We know that defining and understanding page flow of a complex web application is difficult no matter which framework it is based on. For a struts application you will have to write action to handle and process each request. This approach may be simple for simpler applications but image if you were to write an application with complex paths or process flows involving large number of views, the task would be difficult using the traditional approach. Maintenance of such an application is another tedious job, you will have to dig into the code to understand the process flow of the application. This problem gets considerably worse as a web application gets larger and more complex. So in a way these base Model 2 frameworks hide the higher-level flow which makes it difficult to track the lifecycle of the page flow.

The solution here is provided by the Spring Web Flow. Spring Web Flow allows you to represent the page flows of your application in a clear and simple way and the good part is that you can use it Spring Web Flow with other frameworks like Struts, Spring and JSF.

**Advantages of Spring Web Flow:**

Page flow of the application is visible just by looking at XML or java configuration. Web flows are designed to be self contained, and thus are reusable multiple of times. The technique to capture the page flow remains the same for all the cases and there are no specialized approaches for particular situations.

**How does Spring Web Flow Work?**

Spring Web Flow is composed of a set of states (Displaying a View or executing any Action etc.). Transition of the flow from one state to another is triggered by an event. This continues till the flow completes and enters the end-state. The important Spring Web Flow states are:

* The start-state, when a flow is created the initial state of the flow is defined by the start-state attribute in the Webflow.
* An action-state executes an action and returns a logical result on its completion. The next to which the flow will be transitioned to depends on the result of this state A view-state when entered pauses the flow and returns the control to the client/user and the flow is resumed on the user/client event which resumes the flow and triggers the transition to the state depending on the user/client input or decision.
* A decision-state is used to determine the next state in the dynamic way or at runtime. If our next state depends on some attributes or properties ( eg. If users are not logged then redirect them to login page ).
* A subflow-state is used to represent independent flows which are not dependent on the main flow. A subflow is created as child of main flow (parent flow). When a subflow is called the parent flow is suspended until the child flow completes. This helps to maintain the application as a set of sub-modules (Subflows) which can be used multiple times. The Subflow can be child of another Subflow or of the root flow.
* End-state signifies the end of the flow. When a flow enters the end-state the active flow session is terminated. If the end-state of the root flow is entered the resources associated with it are cleaned up automatically.

