**Project On**

**EShop.com E-Commerce Project**

**Developed Under**

**NIIT DIGINXT Training**

**@NIIT Saheed Nagar Bhubaneswar**

**Developed By**

**Mr. Rahul Thakur**

**Guided By**

**Miss Sweta Sahoo**

**ABSTARCT**

Electronic Commerce is process of doing business through computer networks. A person sitting on his chair in front of a computer can access all the facilities of the Internet to buy or sell the products.

Unlike traditional commerce that is carried out physically with effort of a person to go & get products, ecommerce has made it easier for human to reduce physical work and to save time.    E-Commerce which was started in early 1990’s has taken a great leap in the world of computers, but the fact that has hindered the growth of e-commerce is security. Security is the challenge facing e-commerce today & there is still a lot of advancement made in the field of security.

The main advantage of e-commerce over traditional commerce is the user can browse online shops, compare prices and order merchandise sitting at home on their PC.

For increasing the use of e-commerce in developing countries the B2B e-commerce is implemented for improving access to global markets for firms in developing countries. For a developing country advancement in the field of e-commerce is essential. The research strategy shows the importance of the e-commerce in developing countries for business applications.

Contents

1. [Introduction 4](#_Toc450894923)
2. [Scope 4](#_Toc450894924)
3. [Objective 4](#_Toc450894925)
4. [Methodology 5](#_Toc450894926)
5. [Hardware and Software Requirements 5](#_Toc450894927)
6. [Software Installation 6](#_Toc450894928)

* [Getting Started with Eclipse IDE 6](#_Toc450894929)
* [Getting Started with H2 Database 8](#_Toc450894930)
* [Getting Started with Maven and Configuring Tomcat Server 9](#_Toc450894931)
* [Getting Started with GITHUB 10](#_Toc450894933)
* [Adding Dependencies in Maven 21](#_Toc450894935)

1. [Adding Boot Strap Front End Framework 23](#_Toc450894936)
2. [Spring MVC PATTERN 25](#_Toc450894937)

* [Architecture 25](#_Toc450894938)
* [Dispatcher Servlet 27](#_Toc450894939)
* [Spring Application Context 30](#_Toc450894941)
* [Handler Mapping 31](#_Toc450894942)
* Controller………………………………………………………………………………………….35
* ModelAndView and ViewResolver…………………………………………………….37
* Spring MVC Form Model Implementation [32](#_Toc450894943)

1. [CRUD Operations using Spring-Hibernate Integration 34](#_Toc450894944)

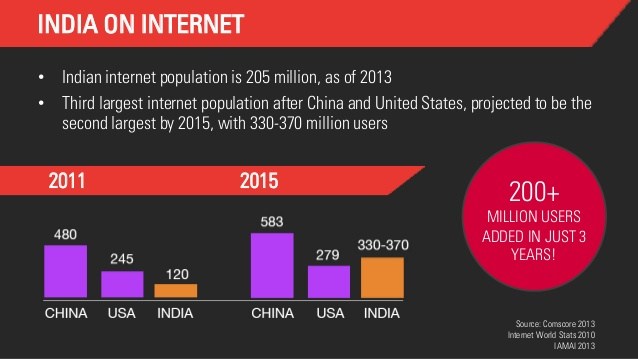
* [Create DAO Class 41](#_Toc450894947)
* [Implementing Session Factory 41](#_Toc450894948)
* [Implementing Hibernate Configuration File 42](#_Toc450894949)
* [Implementing Transaction Manager 45](#_Toc450894950)

***Introduction :***

Electronic commerce or ecommerce is a term for any type of business, or commercial transaction, that involves the transfer of information across the Internet. The project titled **EShop.com** covers a range of different types of businesses, from consumer based retail sites, through auction or music sites, to business exchanges trading goods and services between corporations. It is currently one of the most important aspects of the Internet to emerge.

***Scope :***

**E-Commerce, or the Electronic Commerce** is more than just buying and selling products online. It also includes the entire online process of developing, marketing, selling, delivering, servicing and paying for products and services. India has shown tremendous growth in the E-commerce segment. With an **internet user base of over 300 million**, India has third largest internet population after US & China (see info-graphic below).

****

***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 various JavaScript frameworks such as Bootstrap & JQuery.
* Build persistent and loosely coupled Web apps using Hibernate and Spring technologies.
* Build high quality Web/enterprise apps by using DevOps platform.

***Methodology :***

**Scrum** is an iterative and incremental [agile software development](https://en.wikipedia.org/wiki/Agile_software_development) framework for managing product development.It defines "a flexible, [holistic](https://en.wikipedia.org/wiki/Holism) product development strategy where a development team works as a unit to reach a common goal",challenges assumptions of the "traditional, sequential approach"to product development, and enables teams to self-organize by encouraging physical [co-location](https://en.wikipedia.org/wiki/Colocation_(business)) or close online collaboration of all team members, as well as daily face-to-face communication among all team members and disciplines involved.

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 student and prepare for Developer role. The students are expected to spend at least 70% of their time on coding for the project work during the program.

***Hardware & Software Requirements :***

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

Hardware Requirements:

* Intel Core i3 processer minimum
* 2 GB RAM minimum (Recommended 4GB)
* System type : 64bit
* Hard Disk Space Required 20GB Min
* Internet Connection with Minimum 512kbps (1 MBPS Recommended)

Software Requirements:

* Operating System Windows 7, 8, 10 or above
* Eclipse Mars IDE
* Apache tomcat 8 server
* Github Repository(For daily uploading)
* Java Development Kit version 8
* H2 Database
* Bootstrap Scripts
* JQuery Scripts

***Software Installation :***

* **Getting Started With Eclipse Mars IDE**

### Downloading

Click [Eclipse](http://www.eclipse.org/downloads/packages/release/Luna/R)

The following page will appear in your browser. In this handout we will download **Eclipse Standard 4.4** for Windows 32 Bit; if your computer uses Windows, continue below; otherwise look for the pull-down list showing **Windows** and instead choose either **Mac OS X (Cocoa)** or **Linux** and then continue below.

It is critical that Java, Python, and Eclipse are either all 32 Bit or are all 64 Bit (and only if your Machine/OS supports 64 Bit): I think it easiest to use 32 Bit for everything.

Click the **Windows 32 Bit** Operating System for your machine, under the heading **Eclipse Standard 4.4** (right under **Package Solutions**).

You will see the following page (don't worry about the name of the institution to the right of the big downward pointing arrow).

Click the orange **DOWNLOAD** button. The site named here, in orange to the right of the button: **United States - Columbia University (http)** is the random one chosen by the download page this time; yours may differ.

This file should start downloading in your standard download folder, while showing a splash screen about donating to Eclipse. This file is about 200 Mb so it might take a while to download fully if you are on a slow internet connection (it took me about 5 minutes over a cable modem). Don't worry about the exact time as long as the download continues to make steady progress. In Chrome progress is shown on the bottom-left of the window, via the icon

https://www.ics.uci.edu/~pattis/common/handouts/pythoneclipsejava/images/eclipse/downloadprogress.jpg

The file should appear as

https://www.ics.uci.edu/~pattis/common/handouts/pythoneclipsejava/images/eclipse/downloadfile.jpg

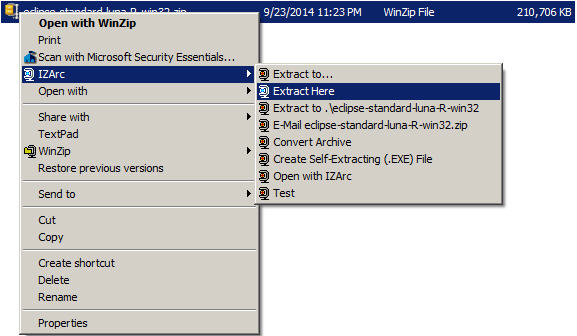
Terminate the window browsing the Eclipse download.

1. Move this file to a more permanent location, so that you can install Eclipse (and reinstall it later, if necessary).
2. Start the **Installing** instructions directly below.

### Installing (for Python)

**Python and Java should be installed before installing Eclipse**

1. Unzip **eclipse-standard-luna-R-win32.zip**, the file that you just downloaded and moved.   
   On my machine (running Windows 7), I can
   * Right-click the file.
   * Hover over the **IZArc** command from the menu of options.
   * Click **Extract Here**



If you do not have IZArc or an equivalent unzipping program, here is the web site to download a free copy of [IZarc](http://izarc.org/download.html).

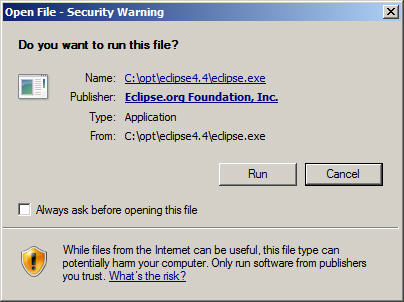
Unzipping this file creates a folder named **eclipse**; unzipping 200 MB can take a few minutes. You can leave this folder here or move it elsewhere on your hard disk. I recommend putting the downloaded file and resulting folder in the **C:\Program Files\** directory.

1. Create a shortcut on your desktop to the **eclipse.exe** file in this **eclipse** folder: https://www.ics.uci.edu/~pattis/common/handouts/pythoneclipsejava/images/eclipse/eclipseexe.jpg   
   On most Windows machines, you can
   * Right-press the file **eclipse.exe**
   * Drag it to the desktop.
   * Release the right button.
   * Click **Create shortcut here**

Now you are ready to perform a **one-time only** setup of Eclipse on your machine.

1. Double-click the shortcut to Eclipse that you just created above.

If the following **Security Warning** window appears

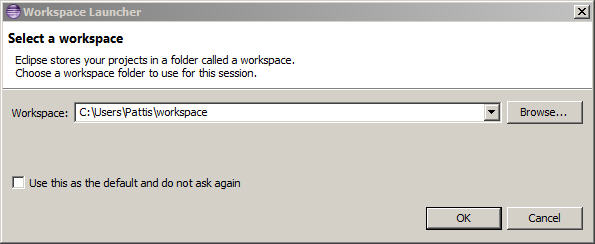


* + Uncheck the **Always ask before opening this file** checkbox
  + Click **Run**

The following splash screen will appear



and then a **Workspace Launcher** pop-up window will appear.



In the **Workspace** text box, your name should appear between **C:\Users\** and **\workspace**, instead of **Pattis**.

Leave **unchecked** the **Use this as the default and do not ask again** box. Although you will use this same workspace for the entire quarter (checking projects in and out of it), it is best to see this **Workspace Launcher** pop-up window each time you start Eclipse, to remind you where your workspace is located.

In fact, it is a good idea to create on your desktop a shortcut to your workspace folder; but you must click **OK** (see below) before Eclipse creates this folder and you can create a shortcut to it.

1. Click **OK**.

Progress bars will appear as Eclipse loads.

Eventually the Eclipse workbench will appear with a **Welcome** tab covering it.



1. Terminate (click **X** on) the **Welcome** tab.

You will not see the **Welcome** tab when you start Eclipse again, after this first time.

1. Click **Help** (on the far right of the line below this window's blue title **Java - Eclipse**) and then click **Install New Software...** in its pull-down menu, as shown below.

* **Getting Started With H2 Database**

## Embedding H2 in an Application

This database can be used in embedded mode, or in server mode. To use it in embedded mode, you need to:

* Add the h2\*.jar to the classpath (H2 does not have any dependencies)
* Use the JDBC driver class: org.h2.Driver
* The database URL jdbc:h2:~/test opens the database test in your user home directory
* A new database is automatically created

## 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 or Vista, Mac OS X, or Linux
* Sun Java 6 or newer
* Recommended Windows file system: NTFS (FAT32 only supports files up to 4 GB)

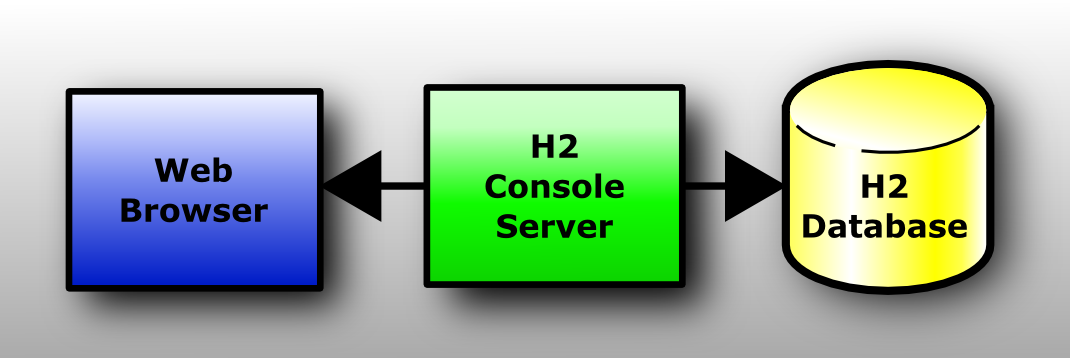
### H2 Console

* Mozilla Firefox

## Supported Platforms

As this database is written in Java, it can run on many different platforms. It is tested with Java 6 and 7. Currently, the database is developed and tested on Windows 8 and Mac OS X using Java 6, but it also works in many other operating systems and using other Java runtime environments. All major operating systems (Windows XP, Windows Vista, Windows 7) are supported.

## The H2 Console Application

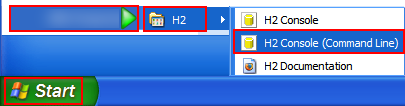
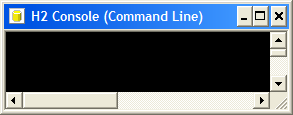
The Console lets you access a SQL database using a browser interface.   
   
If you don't have Windows XP, or if something does not work as expected, please see the detailed description in the internet

### Step-by-Step

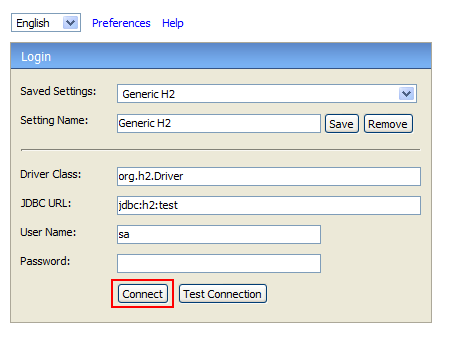
#### Installation

Install the software using the Windows Installer.

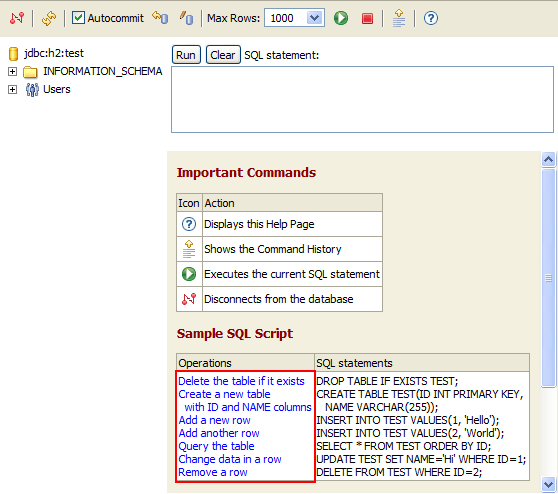
#### Start the Console

Click [Start], [All Programs], [H2], and [H2 Console (Command Line)]:  
  
A new console window appears:  
  
Also, a new browser page should open with the URL [http://localhost:8082](http://localhost:8082/). You may get a security warning from the firewall. If you don't want other computers in the network to access the database on your machine, you can let the firewall block these connections. Only local connections are required at this time.

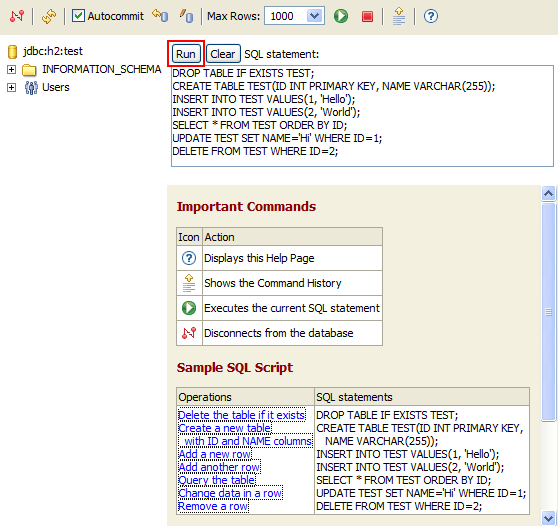
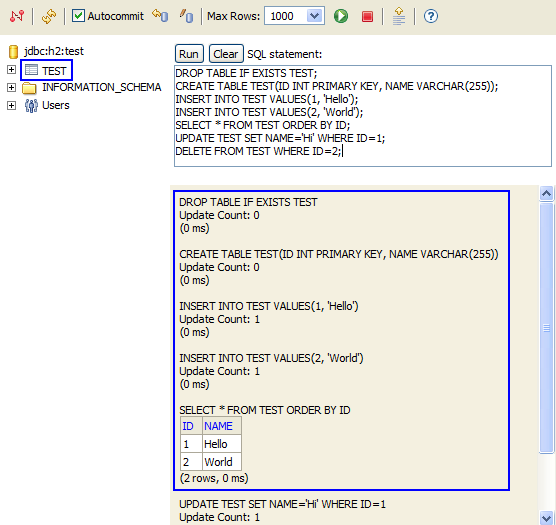
#### Login

Select [Generic H2] and click [Connect]:  
  
You are now logged in.

#### Sample

Click on the [Sample SQL Script]:  
  
The SQL commands appear in the command area.

#### Execute

Click [Run]  
  
On the left side, a new entry TEST is added below the database icon. The operations and results of the statements are shown below the script.  


#### Disconnect

Click on [Disconnect]:  
Disconnect icon  
to close the connection.

#### End

Close the console window.

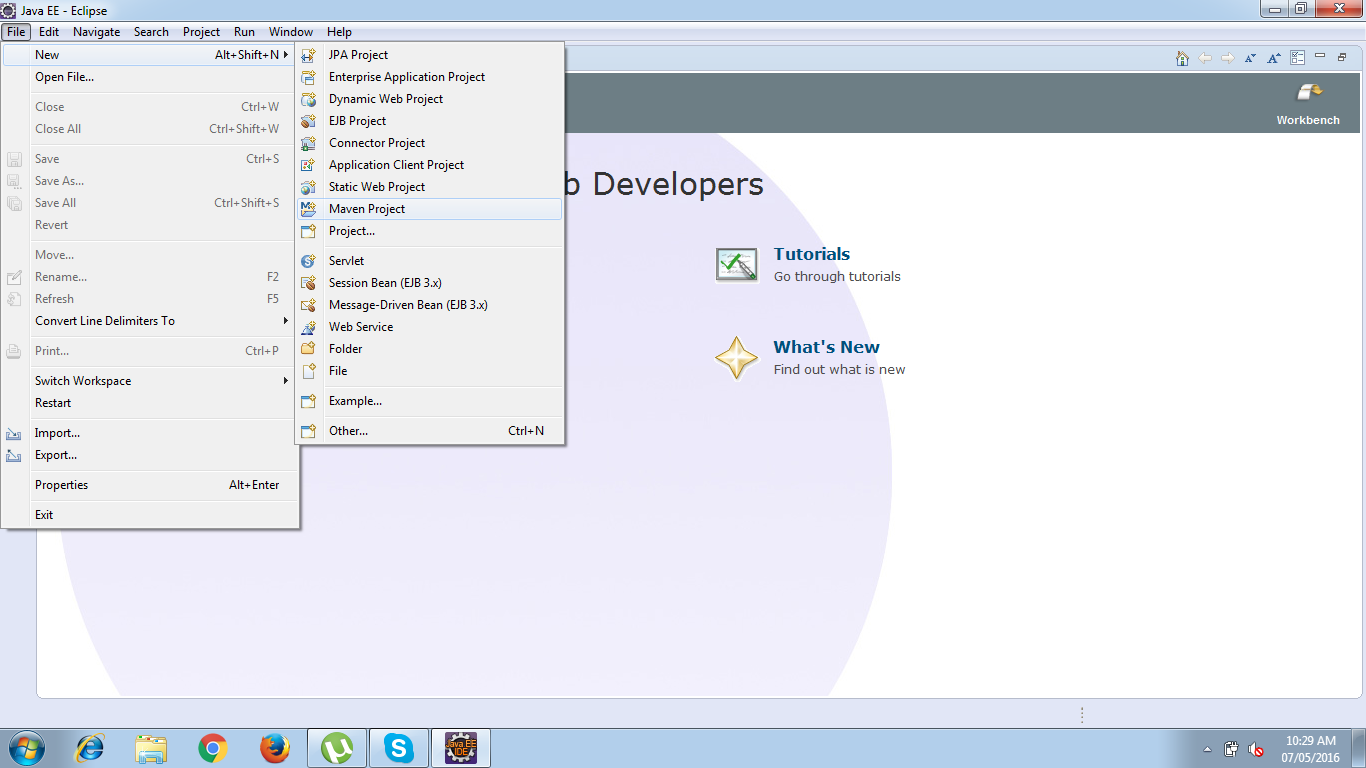
* **Getting Started with Maven and Configuring Tomcat Server**

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

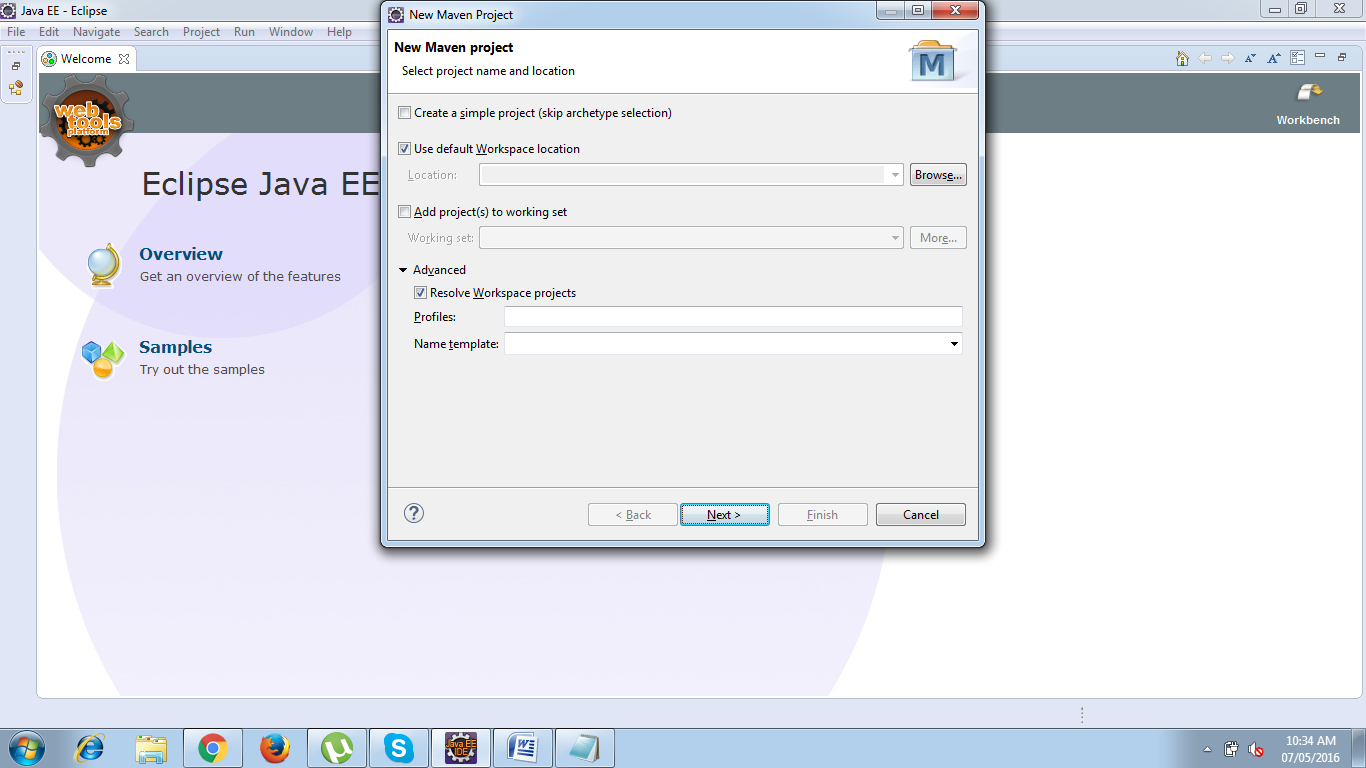
**Step 1.** Create a New Maven Project

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

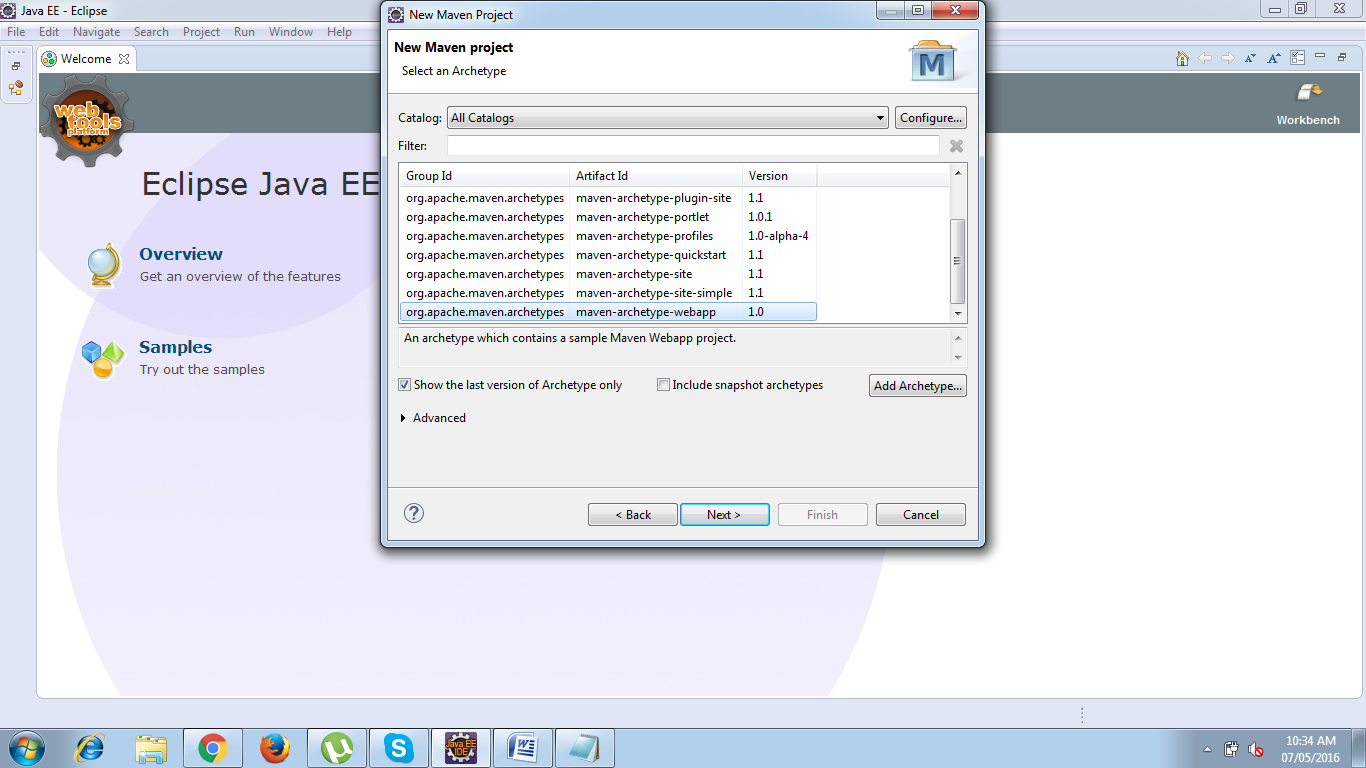


**Step 2.** Select project name and location.

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



##### **Step 3.** Select an archetype



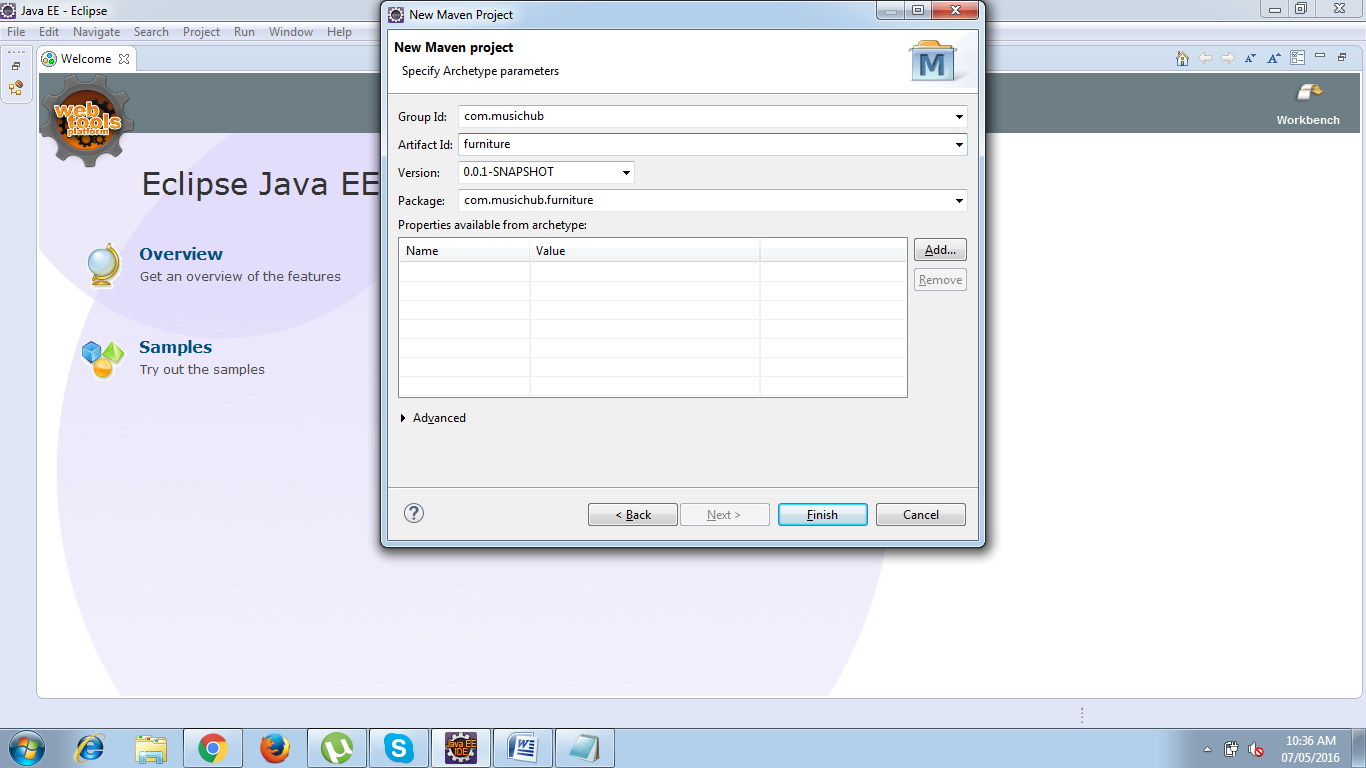
**Step 4.** Enter the Group id and Artifact Id

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.

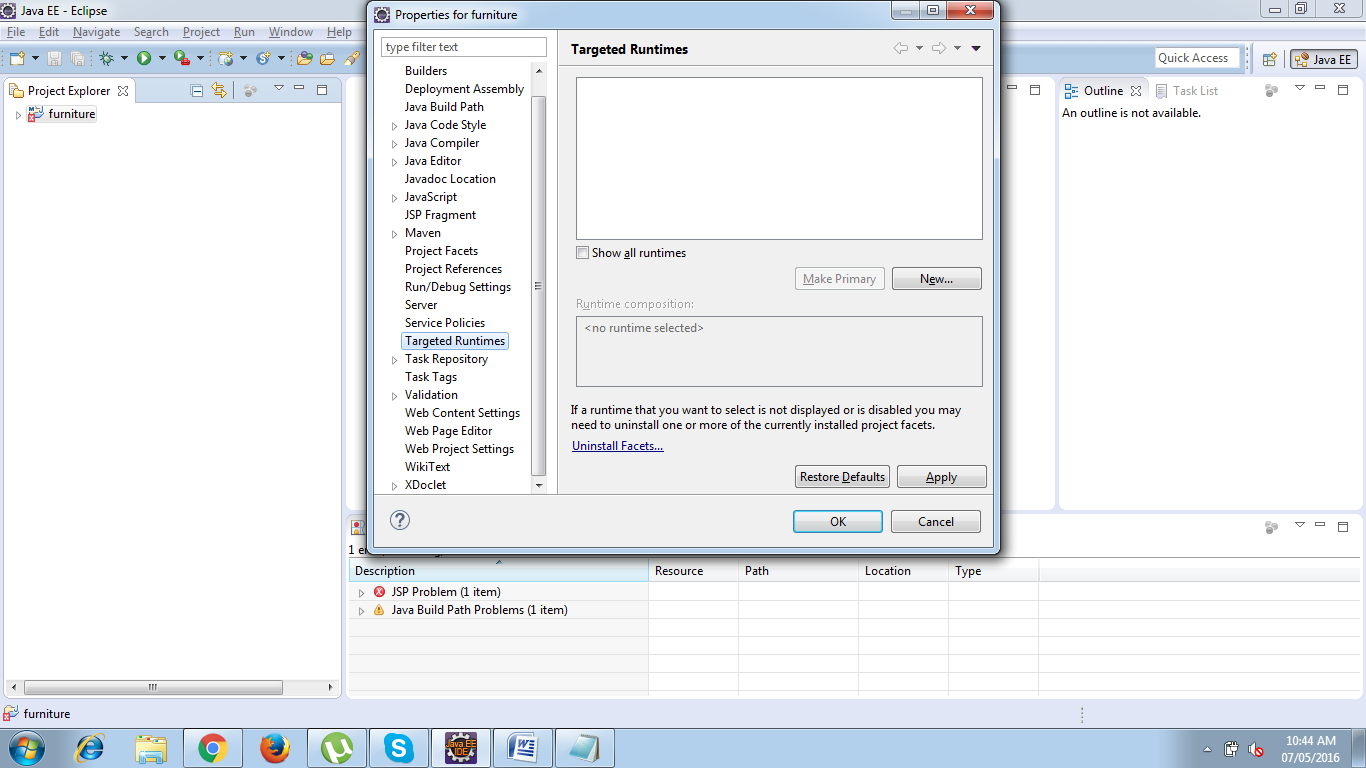
version if you distribute it then you can choose any typical version with numbers and dots (1.0, 1.1, 1.0.1, ...). Don't use dates as they are usually associated with SNAPSHOT (nightly) builds. If it's a third party artifact, you have to use their version number whatever it is, and as strange as it can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



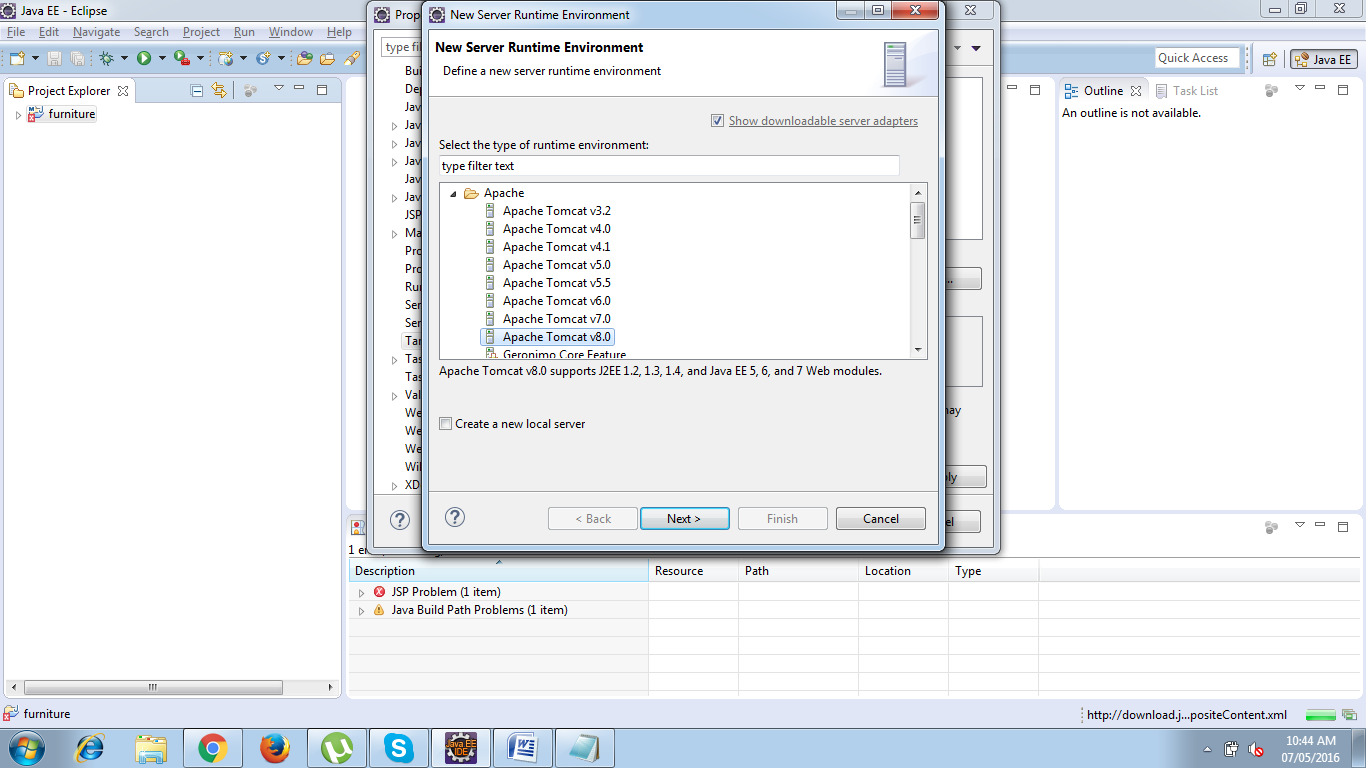
Your project will be ready but with an error due to runtime is not configured for the newly created Workspace.

Updating Target Runtime Server’s :

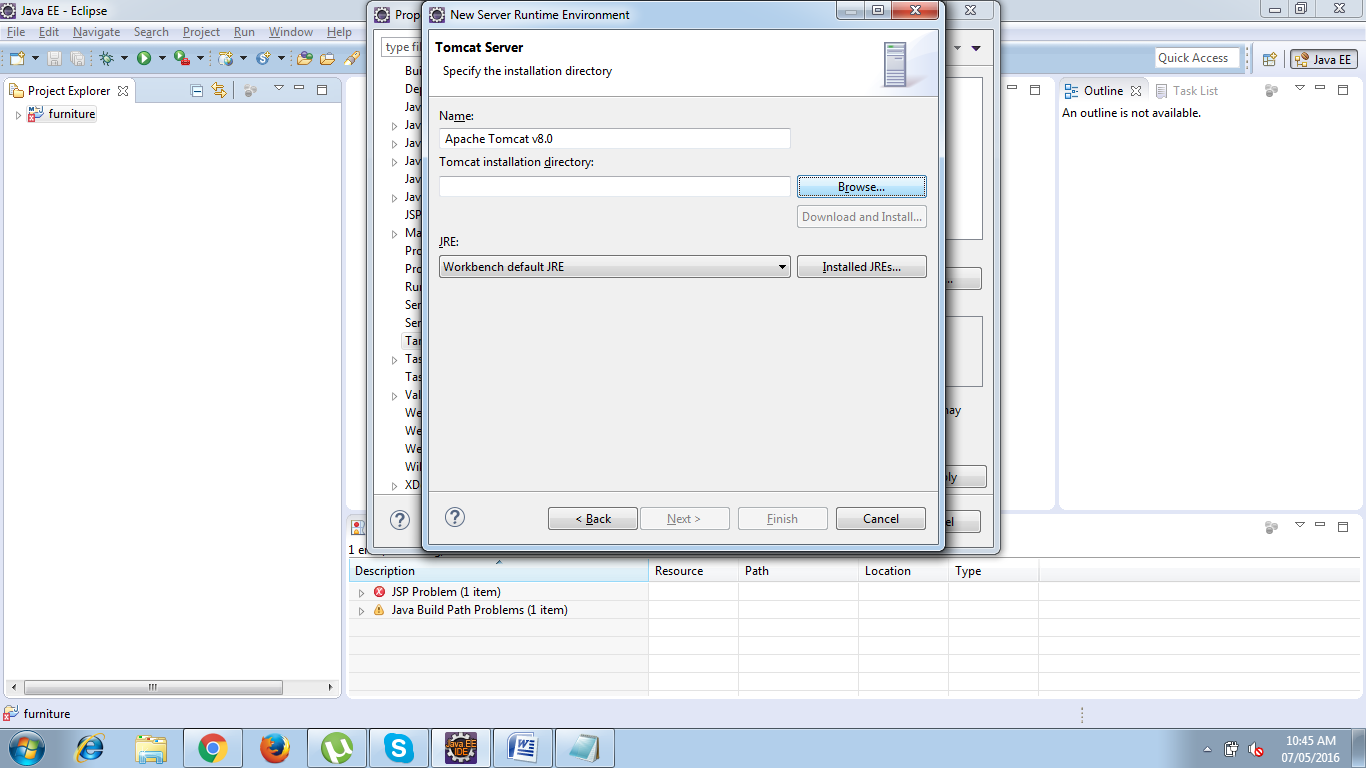
Right Click Project root folder and choose properties :

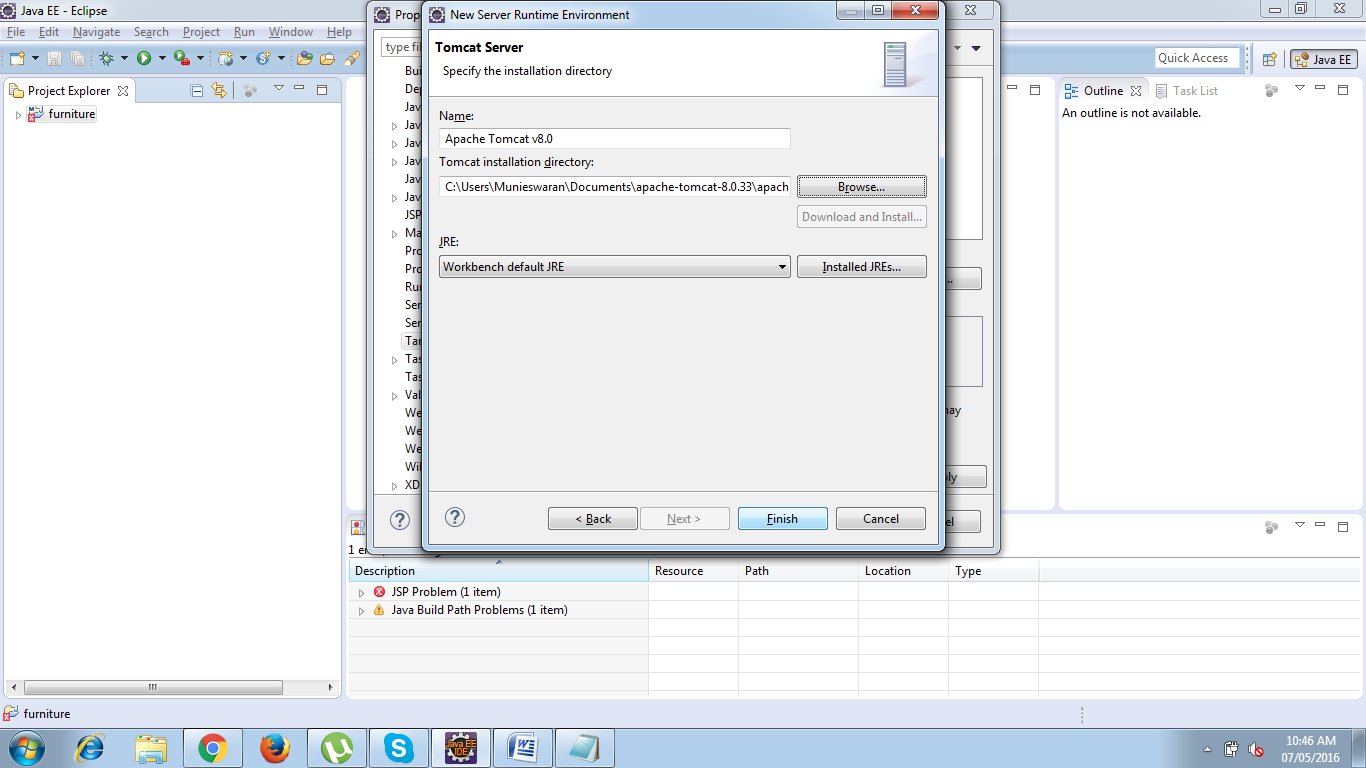


Choose Targeted Runtime which will open New Server Runtime Environment window :



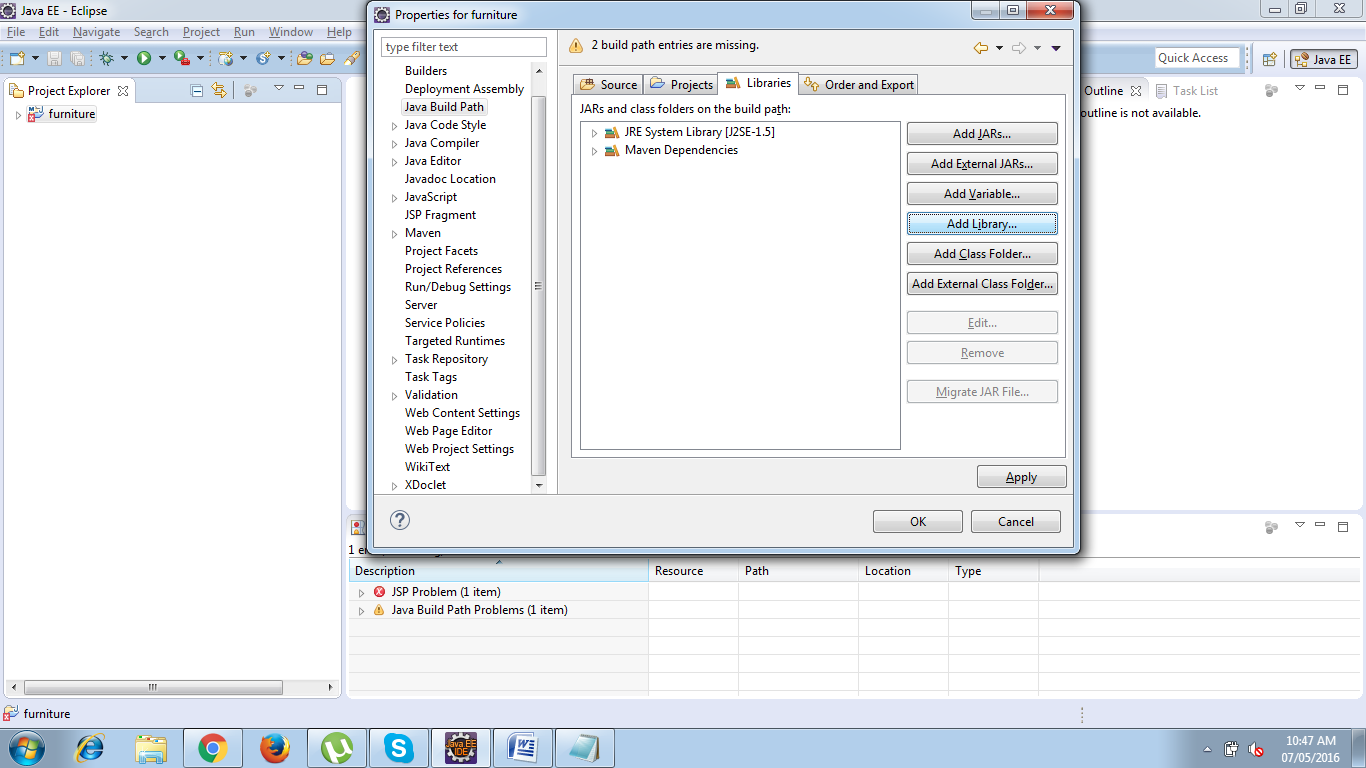
Select Apache Tomcat v8.0 and Click Next -> Choose Tom cat Installation Directory



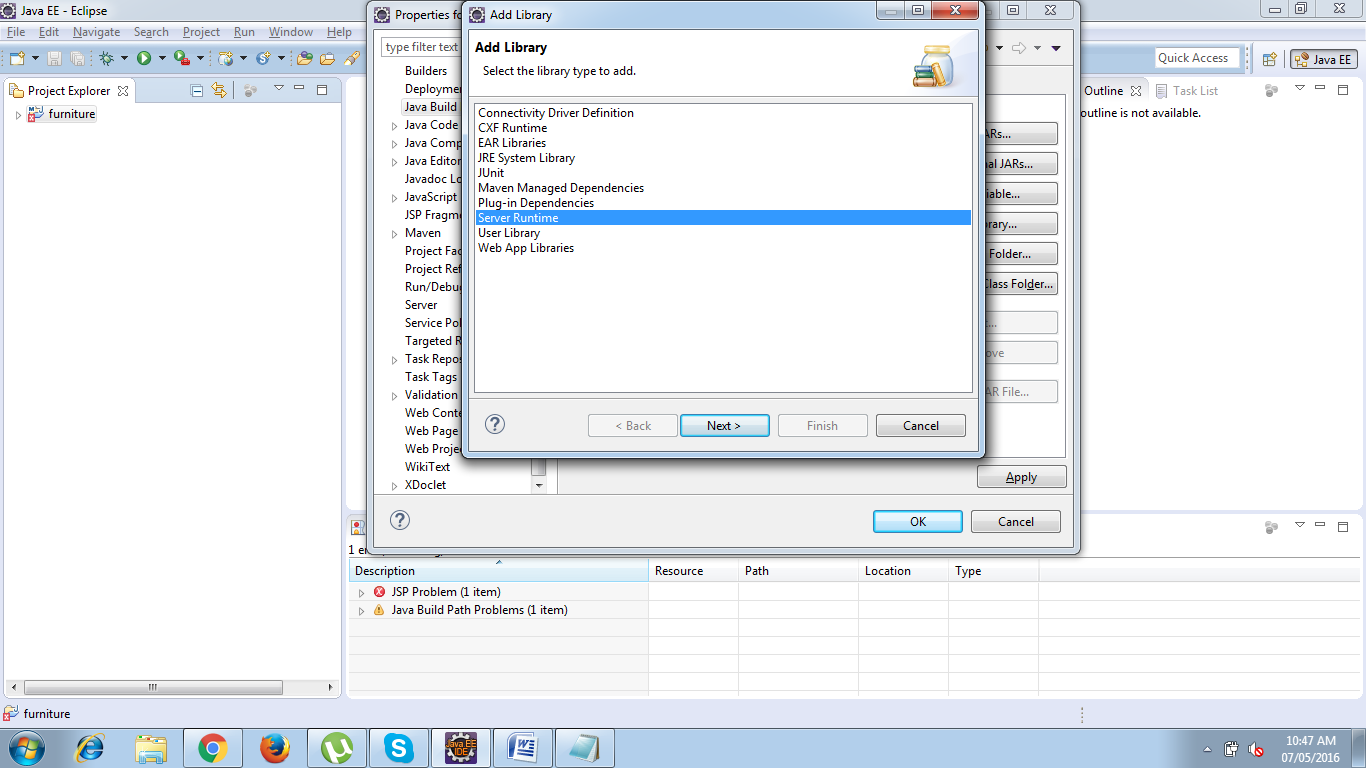


**Click Finish.**

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

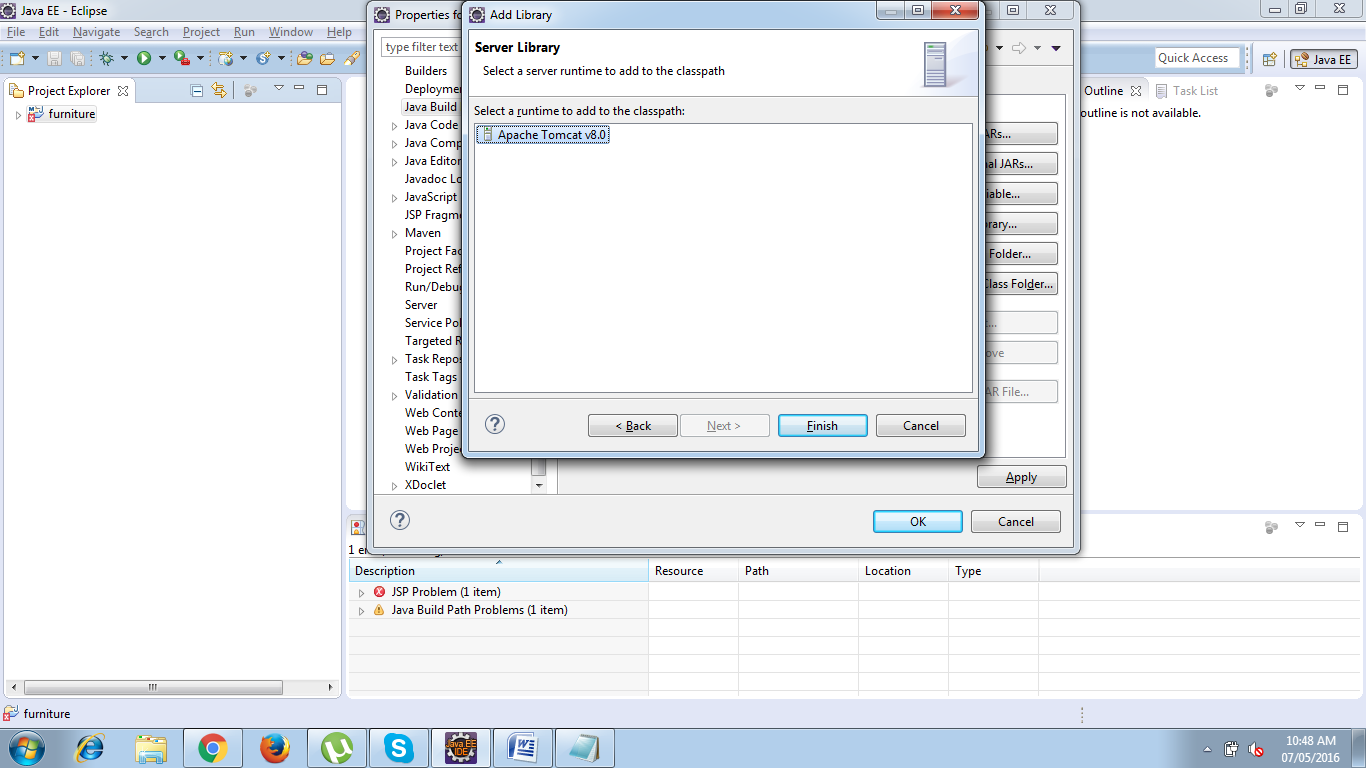


Choose Server Runtime -> Click Next

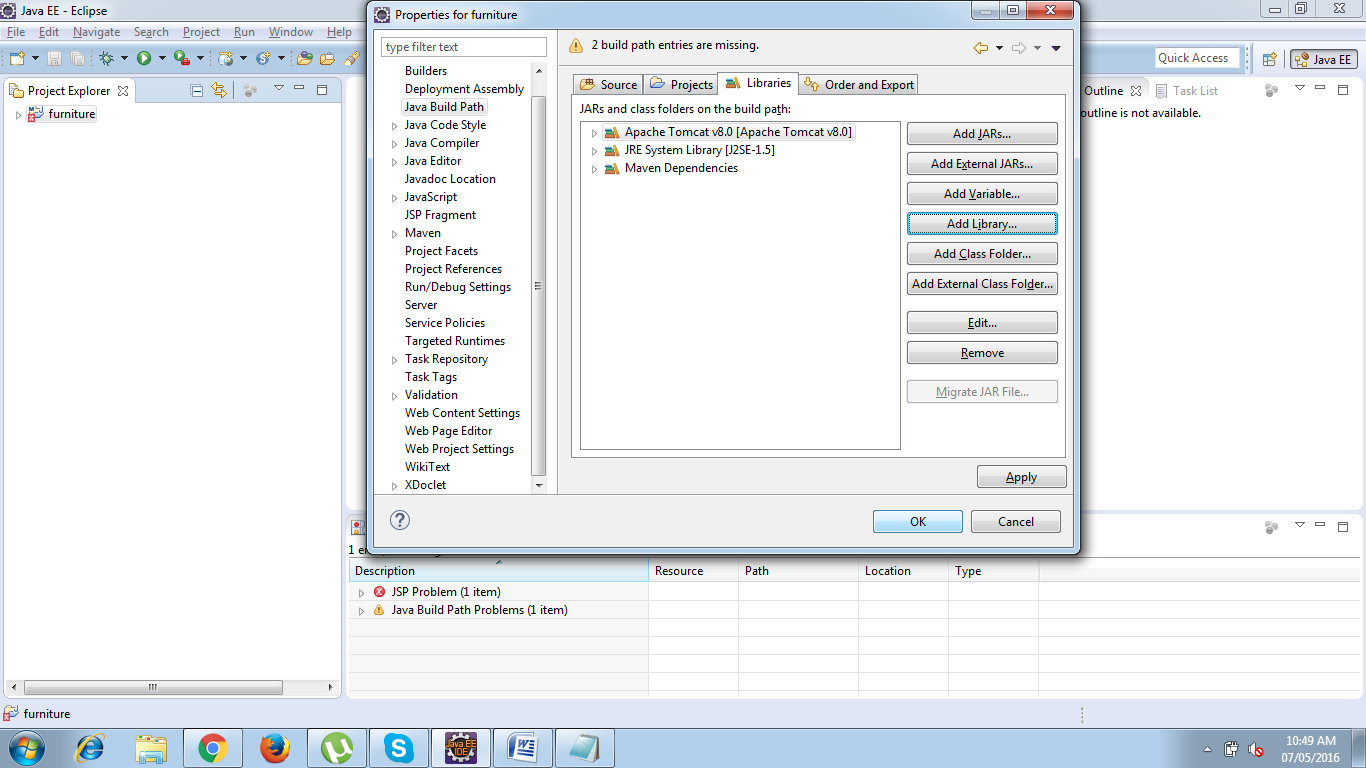


Select a runtime to add to the class path ->Apache Tomcat V8.0

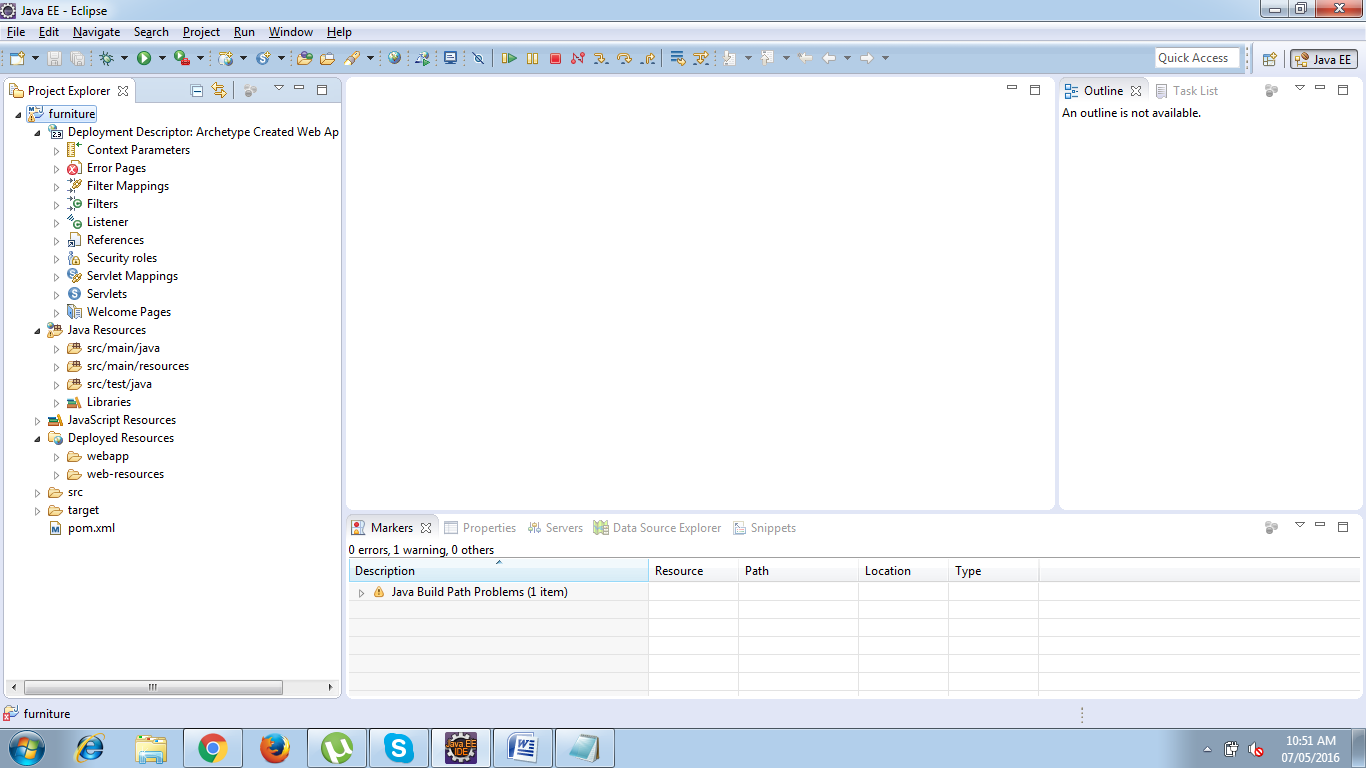
And click finish.



Click OK in the properties window.

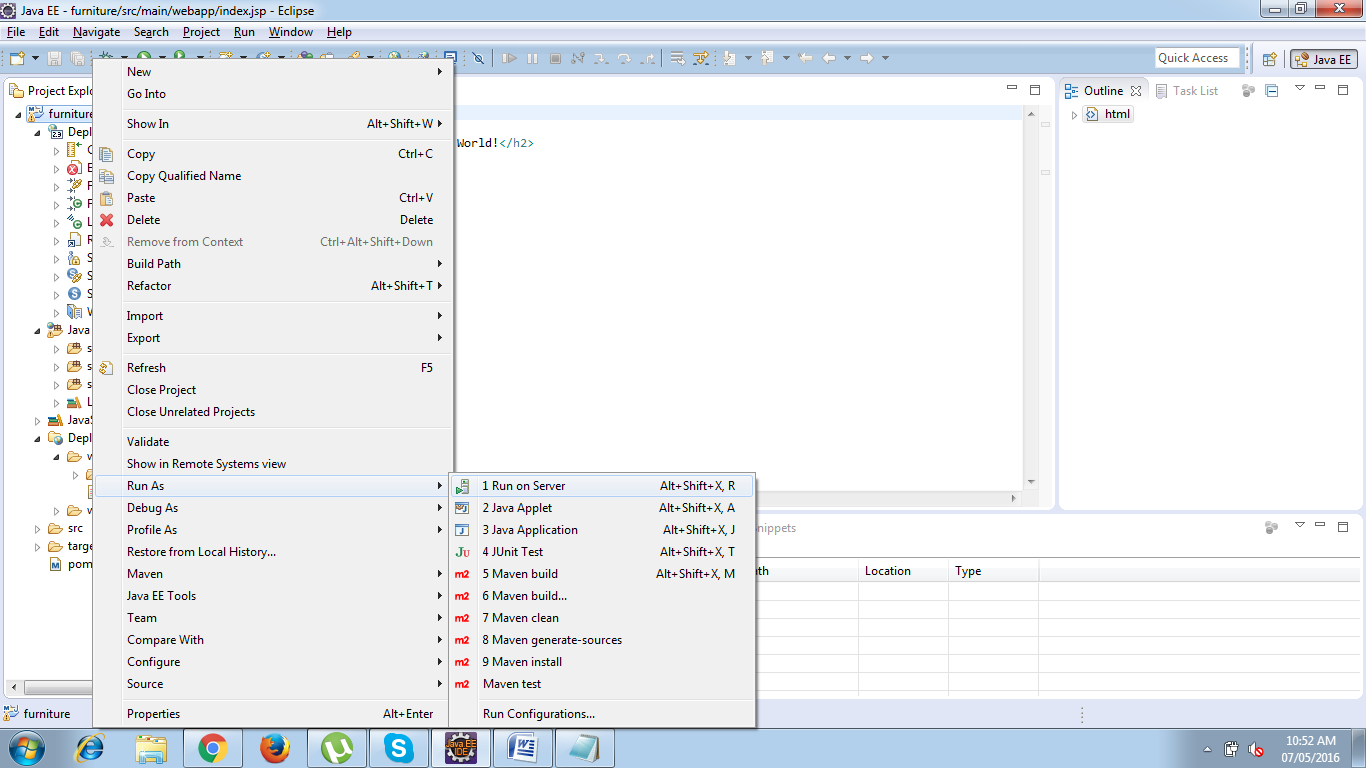


Finally your project is now ready to develop.

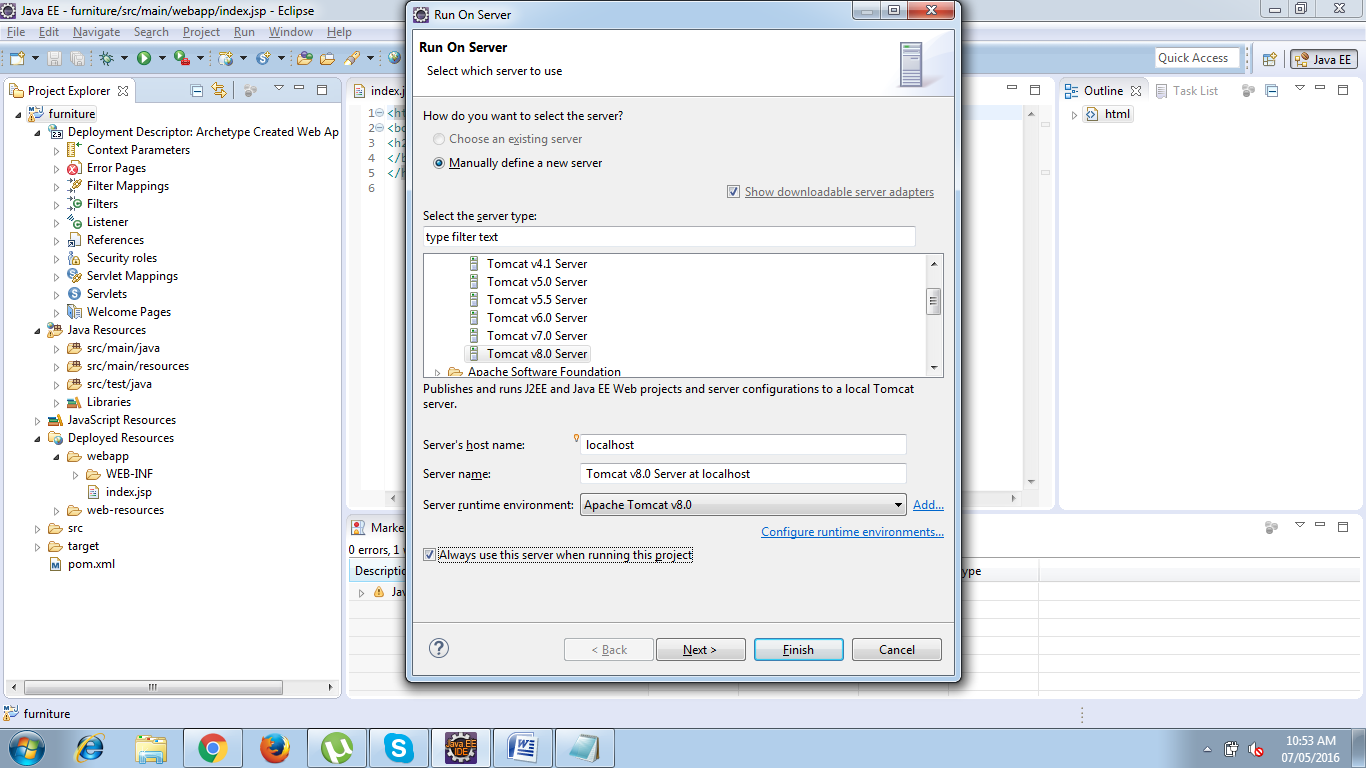


**Testing the Environment**

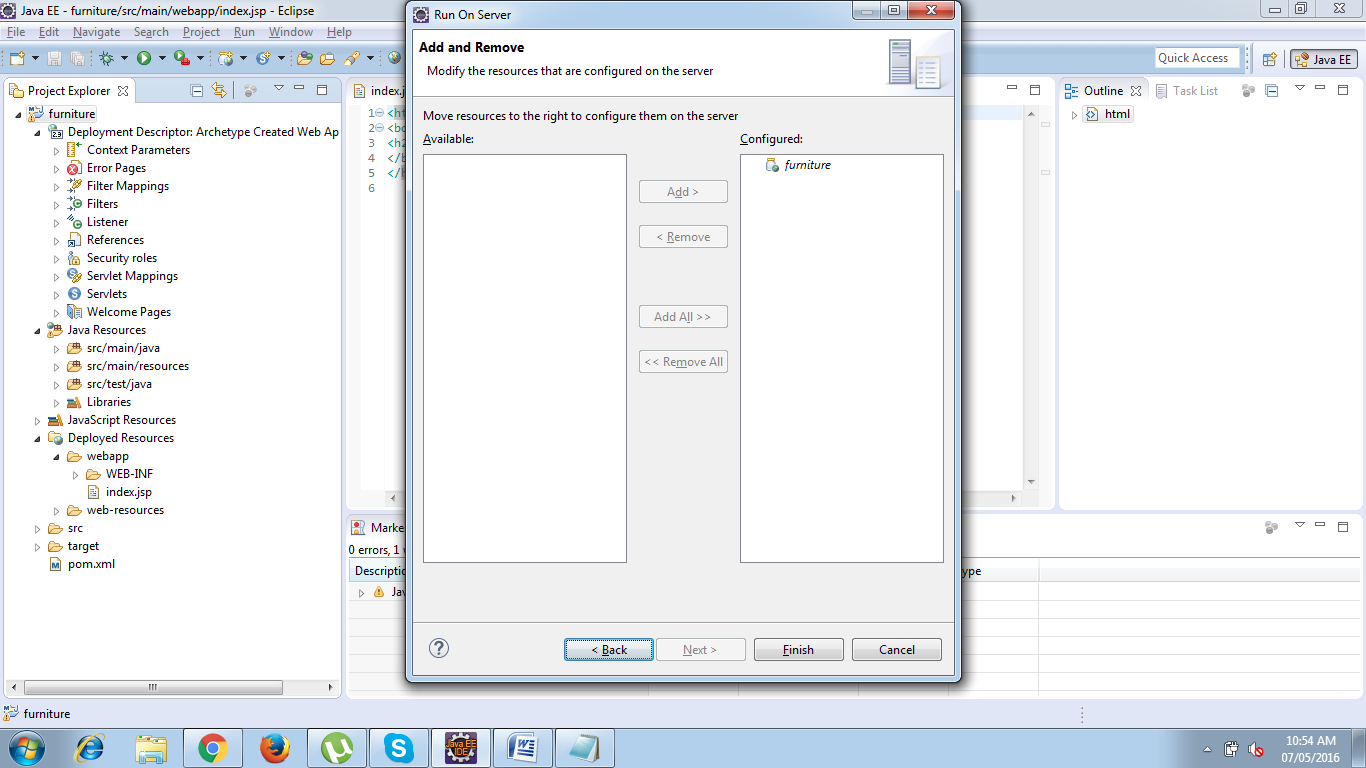
Right Click the project and choose RUN AS -> Run on Server option

****

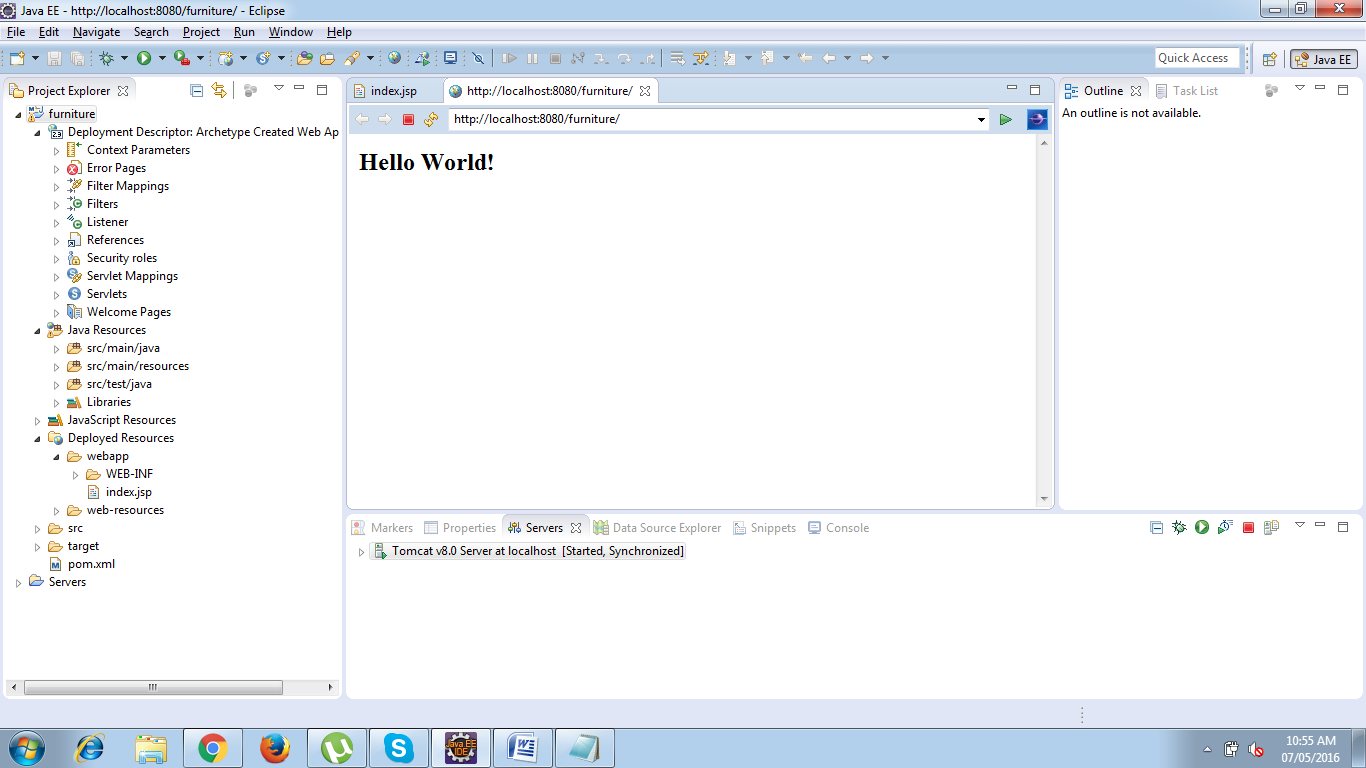
Run on Server window will be now opened , Select the Server to use and select always use this server when running the project check box.



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

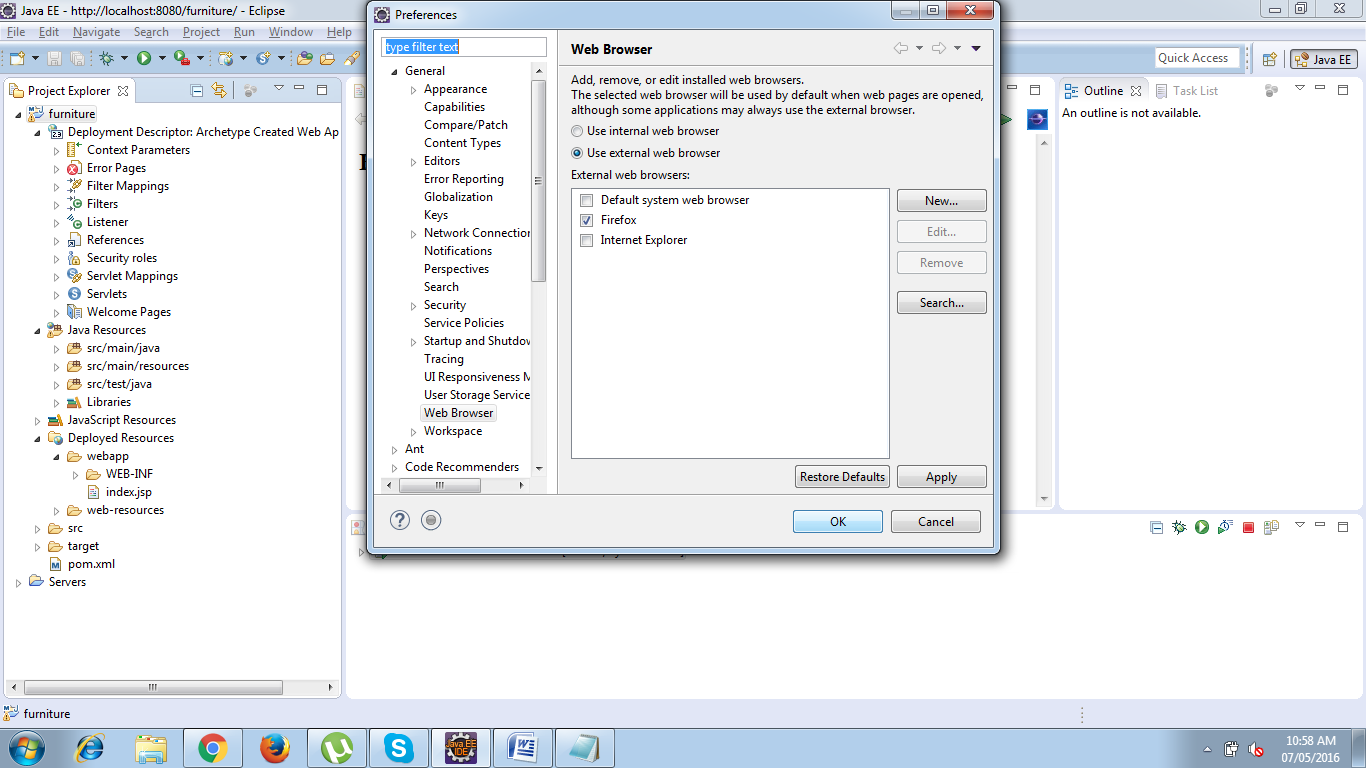


Now your server will start up and your project will be deployed into Tomcat Server and you will see the index page on your default embedded browser.



**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.



* **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.
* **Adding Dependencies in Maven**

**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".

## What is “groupId” in maven ?

“groupId” identifies a particular project uniquely across all projects, so we should follow an naming convention. A very simple and commonly used way of doing this is to use reverse of your domain, i.e. com.beingjavaguys.maven.   
  
A good way of maintaining the integrity of groupId is to use the project structure. In case the project is consists of multiple modules than every module should append an identifier to the parent groupId. i.e. com.beingjavaguys.maven

## What is “artifactId” in maven ?

“artifactId” is the name of war file without version, if you are creating it by yourself you are free to took any name of your choice in lower case and without any strange symbol. But if this is a third party jar than we have to take the name of jar as suggested by it’s distribution.

## What is “archetype” in maven ?

Archetype is a Maven project template toolkit which tells the maven the type of project we are going to create. Archetype enables the maven to create a template project of user’s choice so that the user can get the project up and running instantly.

## What is “archetypeArtifactId” in maven ?

While creating a new project we provide the “archetypeArtifactId” that informs maven about what archetype to use to create the initial structure of the project. Maven looks it up from the “archetypeCatalog” and works accordingly. eg. if we want to create a simple web-app project we specify -DarchetypeArtifactId=maven-archetype-webapp.



# *Adding Boot Strap Front End Framework :*

If you’re doing anything web related, chances are you’ve heard about [Bootstrap](http://getbootstrap.com/). If by now you still don’t know what Bootstrap is, or you just want to find a bootstrap tutorial for beginners to get a better overview of what it is and what it does best, you’ve come to the right place.

Bootstrap is a powerful toolkit - a collection of HTML, CSS, and JavaScript tools for creating and building web pages and web applications. It is a free and open source project, hosted on [GitHub](https://github.com/twbs/bootstrap), and originally created by (and for) [Twitter](https://twitter.com/).

After its open source release in 2011, Bootstrap became popular very quickly, and not without reason. Web designers and web developers like Bootstrap because it is flexible and easy to work with. Its main advantages are that it is responsive by design, it maintains wide browser compatibility, it offers consistent design by using re-usable components, and it is very easy to use and quick to learn. It offers rich extensibility using JavaScript, coming with built-in support for jQuery plugins and a programmatic JavaScript API. Bootstrap can be used with any [IDE](https://en.wikipedia.org/wiki/Integrated_development_environment) or editor, and any server side technology and language, from ASP.NET to PHP to Ruby on Rails.

With Bootstrap, [web developers](https://www.toptal.com/web) can concentrate on the development work, without worrying about design, and get a good looking website up and running quickly. Conversely, it gives web designers a solid foundation for creating interesting Bootstrap themes.

**File Structure**

We’ll focus on the precompiled version, which can be downloaded [here](http://getbootstrap.com/getting-started/#download). When you download the zip archive and uncompress it, the basic file structure looks like this:

bootstrap/

├── css/

│ ├── bootstrap.css

│ ├── bootstrap.css.map

│ ├── bootstrap.min.css

│ ├── bootstrap-theme.css

│ ├── bootstrap-theme.css.map

│ └── bootstrap-theme.min.css

├── js/

│ ├── bootstrap.js

│ └── bootstrap.min.js

└── fonts/

├── glyphicons-halflings-regular.eot

├── glyphicons-halflings-regular.svg

├── glyphicons-halflings-regular.ttf

├── glyphicons-halflings-regular.woff

└── glyphicons-halflings-regular.woff2

**Getting Started with this Bootstrap Tutorial**

Bootstrap is available in two forms; as a precompiled version, and as a source code version. The source code version uses the [Less](http://lesscss.org/) CSS preprocessor, but if you are more into [Sass](http://sass-lang.com/), there is an [official Sass port of Bootstrap](https://github.com/twbs/bootstrap-sass) also available. To make it easier to make use of [CSS vendor prefixes](http://webdesign.about.com/od/css/a/css-vendor-prefixes.htm), Bootstrap uses[Autoprefixer](https://github.com/postcss/autoprefixer).

The source code version comes styles source code written in Less (or Sass), all the JavaScript, and accompanying documentation. This allows more ambitious designers and developers to change and customize, at their will, all the provided styles, and to build their own version of Bootstrap. But if you are not familiar with Less (or Sass), or you are just not interested in changing the source code, don’t worry. You can just use the precompiled vanilla CSS. All the styles can be overridden later by using custom styles.

The Bootstrap structure is pretty simple and self-explanatory. It includes precompiled files that enable quick usage in any web project. Besides compiled and minified CSS and JS files, it also includes fonts from[Glyphicons](http://glyphicons.com/), and the optional starting Bootstrap theme.

This structure can be easily incorporated in your own project’s file structure by just including the Bootstrap files exactly as they come out of the zip archive, or if it suits your project better, you can rearrange these files and place them anywhere you like. Just be sure that the Glyphicons fonts folder is on the same level as the CSS folder.

**Basic HTML Template**

A basic Bootstrap HTML template should look something like this:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1">

<title>Bootstrap Template</title>

<link href="css/bootstrap.min.css" rel="stylesheet">

</head>

<body>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"></script>

<script src="js/bootstrap.min.js"></script>

</body>

</html>

It is important to start any HTML with a HTML 5 Doctype declaration, so that browsers know what kind of a document they can expect. The head contains three important <meta> tags that must be declared first, and any additional head tags must be added after these. If you want to support older browsers like IE8, you can also include [HTML 5 shim](https://github.com/aFarkas/html5shiv/) in the head, which will enable use of HTML5 elements in older browsers, and[Respond.js](https://github.com/scottjehl/Respond), that will polyfill CSS3 Media Queries, in the old versions of Internet Explorer.

<head>

...

<!--[if lt IE 9]>

<script src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js"></script>

<script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>

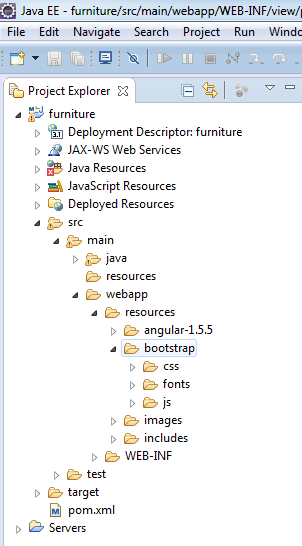
<![endif]-->

</head>

Although this is not very important if you are targeting only modern browsers.

JavaScript files are added to the end of the body to allow the web page to load visibly before any JavaScript is executed. jQuery is needed for Bootstrap plugins, and needs to load before bootstrap.js. If you aren’t using any of Bootstrap’s interactive features, you can also omit these files from the source.

This is the bare minimum that is needed to get a basic Bootstrap layout up and running. If you’re a developer, you’ll probably want to take a look at some more advanced examples at [Bootstrap’s examples page](http://getbootstrap.com/getting-started/#examples). If you’re a designer, or just looking for inspiration, [Bootstrap Expo](http://expo.getbootstrap.com/) showcases sites that are built using Bootstrap. As we’ll see later, every part of Bootstrap can be easily customized in CSS. But if that’s not your thing, and you are looking for a slightly different look and feel from the prepackaged Bootstrap themes, there are a lot of free, open source and premium themes available from sources like [Bootswatch](https://bootswatch.com/) and[WrapBootstrap](https://wrapbootstrap.com/).



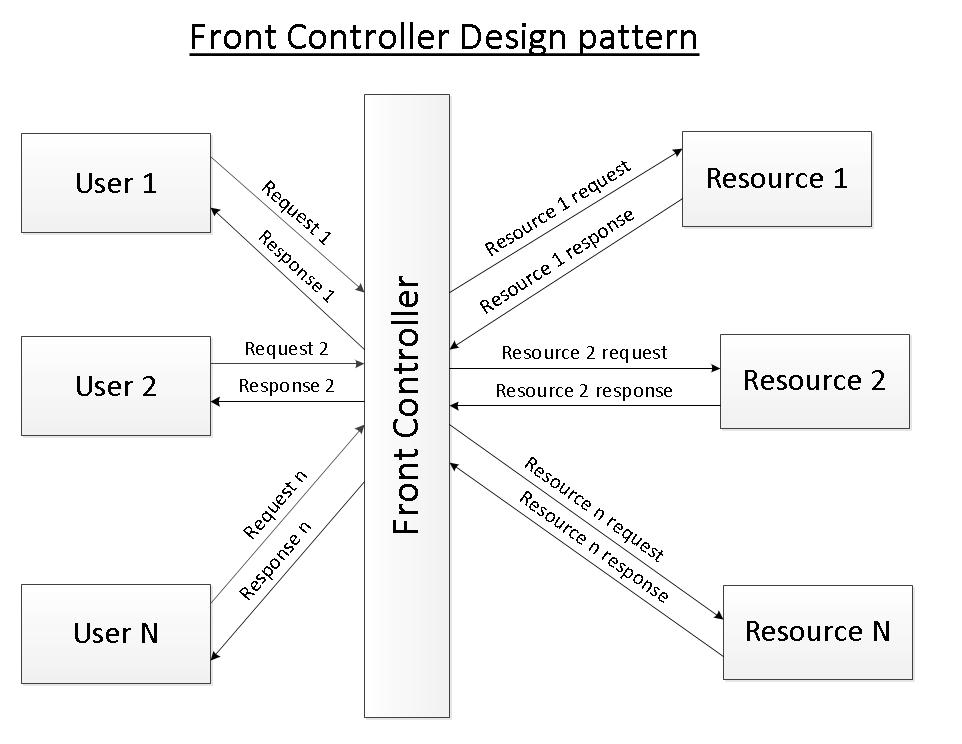
***Spring MVC Pattern :***

The Spring Web model-view-controller (MVC) framework is designed around a Dispatcher Servlet that dispatches requests to handlers, with configurable handler mappings, view resolution, locale, time zone and theme resolution as well as support for uploading files. The default handler is based on the @Controller and @RequestMapping annotations, offering a wide range of flexible handling methods. With the introduction of Spring 3.0, the @Controller mechanism also allows you to create RESTful Web sites and applications, through the @PathVariable annotation and other features.

* **Architecture**

### Front Controller design pattern

This design pattern enforces a single point of entry for all the incoming requests. All the requests are handled by a single piece of code which can then further delegate the responsibility of processing the request to further application objects.



Front Controller Design Pattern

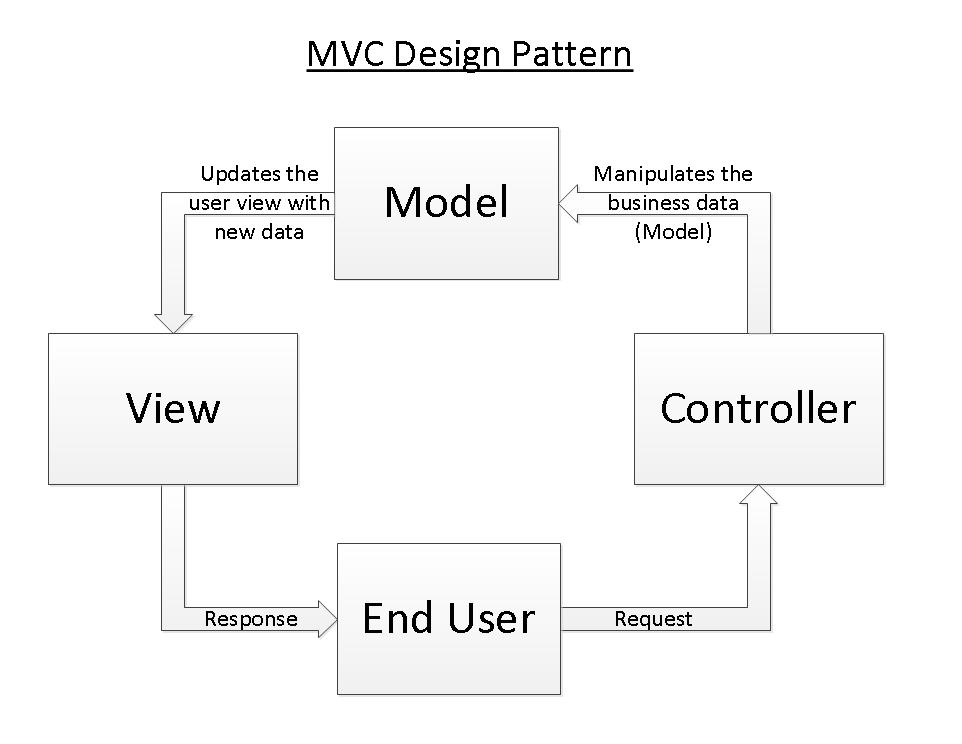
### MVC design pattern

This design pattern helps us develop loosely coupled application by segregating various concerns into different layers. MVC design pattern enforces the application to be divided into three layers, Model, View and Controller.

**Model:** This represents the application data.

**View:** This represents the application’s user interface. View takes model as the input and renders it appropriately to the end user.

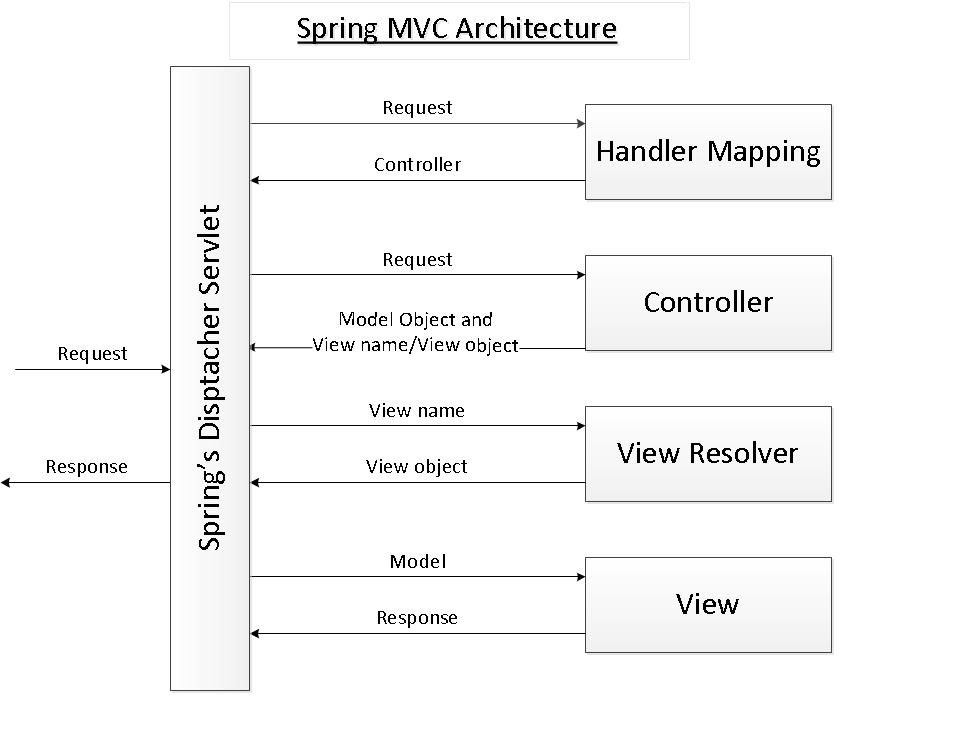
**Controller:** The controller is responsible for handling the request and generating the model and selecting the appropriate view for the request.



MVC Design Pattern

### Spring’s MVC module

Spring’s MVC module is based on front controller design pattern followed by MVC design pattern. All the incoming requests are handled by the single servlet named DispatcherServlet which acts as the front controller in Spring’s MVC module. The DispatcherServlet then refers to the  HandlerMapping to find a controller object which can handle the request.DispatcherServlet then dispatches the request to the controller object so that it can actually perform the business logic to fulfil the user request. (Controller may delegate the responsibility to further application objects known as service objects). The controller returns an encapsulated object containing the model object and the view object (or a logical name of the view). In Spring’s MVC, this encapsulated object is represented by class ModelAndView. In case ModelAndView contains the logical name of the view, the  DispatcherServlet refers the ViewResolver to find the actual View object based on the logical name. DispatcherServlet then passes the model object to the view object which is then rendered to the end user.



* **Dispatcher Servlet**

DispatcherServlet acts as the front controller in the Spring’s MVC module. All the user requests are handled by this servlet. Since this is like any other servlet, it must be configured in the application’s web deployment descriptor file i.e. web.xml.

We have named the servlet as “myLibraryAppFrontController”. The URI pattern in the servlet mapping section is “\*.htm”. Thus all the requests matching the URI pattern will be handled by myLibraryAppFrontController.

<web-app xsi:schemaLocation="http://java.sun.com/xml/ns/javaee

http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"

id="WebApp\_ID"

version="2.5"

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

xmlns="http://java.sun.com/xml/ns/javaee"

xmlns:web="http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">

   <display-name>Library</display-name>

   <servlet>

      <servlet-name>myLibraryAppFrontController</servlet-name>

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

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

   </servlet>

   <servlet-mapping>

      <servlet-name>myLibraryAppFrontController</servlet-name>

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

   </servlet-mapping>

   <welcome-file-list>

      <welcome-file>welcome.htm</welcome-file>

   </welcome-file-list>

</web-app>

* **Spring Application Context**

**Default Application context file**

By default the dispatcher servlet loads the Spring application context from XML file with name *[servlet name]-servlet.xml*. Thus when our servletmyLibraryAppFrontController is loaded by the container, it will load the Spring application context from XML file “*/WEB-INF/myLibraryAppFrontController-servlet.xml*”.

**User defined application context file**

We can override the name and location of the default XML file by providing the initialization parameters to the dispatcher servlet. The name of the initialization parameter is contextConfigLocation. The parameter value specifies the name and location of the application context which needs to be loaded by the container.

<servlet>

   <servlet-name>myLibraryAppFrontController</servlet-name>

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

   <init-param>

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

      <param-value>classpath:libraryAppContext.xml</param-value>

   </init-param>

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

</servlet>

In the above configuration of myLibraryAppFrontController, when the container initializes the dispatcher servlet, it will load the Spring application context from XML file “classpath:libraryAppContext.xml” instead of “*/WEB-INF/myLibraryAppFrontController-servlet.xml*”.

**Multiple application context files**

It is a good practice to split the application into multiple logical units and have multiple application context file. Thus on servlet initialization we need to load all these application context files. It is possible to load the Spring application context from multiple XML file as shown below:

In the above servlet configuration, we have provided multiple XML files as initialization parameter value. All these XML files will be loaded by the container on initialization of the servlet myLibraryAppFrontController.

<servlet>

   <servlet-name>myLibraryAppFrontController</servlet-name>

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

   <init-param>

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

      <param-value>classpath:libraryAppContext.xml

                   classpath:books.xml

                   classpath:chapters.xml

                   classpath:titles.xml</param-value>

   </init-param>

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

</servlet>

* **Handler Mapping**

As the name specifies, the handler mapping maps the request with the corresponding request handler (in fact handler execution chain).  When a request comes to Spring’s dispatcher servlet, it hands over the request to the handler mapping. Handler mapping then inspects the request and identifies the appropriate handler execution chain and delivers it to dispatcher servlet. The handler execution chain contains handler that matches the incoming request and optionally contains the list of interceptors that are applied for the request. Dispatcher servlet then executes the handlers and any associated handler interceptor.

There are number of implementation of hander mapping provided by Spring’s MVC module. Some of these are described below. All the handler mappings classes implement the interface org.springframework.web.servlet.HandlerMapping.

### BeanNameUrlHandlerMapping

This implementation of handler mapping matches the URL of the incoming request with the name of the controller beans. The matching bean is then used as the controller for the request. This is the default handler mapping used by the Spring’s MVC module i.e. in case the dispatcher servlet does not find any handler mapping bean defined in Spring’s application context then the dispatcher servlet uses BeanNameUrlHandlerMapping.

Let us assume that we have three web pages in our application. The URL of the pages are:

1. [http://servername:portnumber/ApplicationContext/welcome.htm](http://servernameportnumber/)
2. [http://servername:portnumber/ApplicationContext/listBooks.htm](http://servernameportnumber/)
3. [http://servername:portnumber/ApplicationContext/displayBookContent.htm](http://servernameportnumber/)

The controllers which will perform the business logic to fulfil the request made to the above pages are:

1. net.codejava.frameorks.spring.mvc.controller.WelcomeController
2. net.codejava.frameorks.spring.mvc.controller.ListBooksController
3. net.codejava.frameorks.spring.mvc.controller.DisplayBookTOCController

Thus we need to define the controllers in Spring’s application context file such that the name of the controller matches the URL of the request. The controller beans in XML configuration file will look as below.

bean

    name="/welcome.htm"

    class="net.codejava.frameorks.spring.mvc.controller.WelcomeController" />

<bean

    name="/listBooks.htm"

    class="net.codejava.frameorks.spring.mvc.controller.ListBooksController"/>

<bean

    name="/displayBookTOC.htm"

    class="net.codejava.frameorks.spring.mvc.controller.DisplayBookTOCController"/>

Note that we need not define the BeanNameUrlHandlerMapping in Spring’s application context file because this is the default one being used.

**SimpleUrlHandlerMapping**

The BeanNameUrlHandlerMapping puts a restriction on the name of the controller beans that they should match the URL of the incoming request. SimpleUrlHandlerMapping removes this restriction and maps the controller beans to request URL using a property “*mappings*”.

The key of the <prop> element is the URL pattern of the incoming request. The value of the <prop> element is the name of the controller bean which will perform the business logic to fulfil the request. SimpleUrlHandlerMapping is one of the most commonly used handler mapping.

<bean

 id="myHandlerMapping"

 class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

   <property name="mappings">

      <props>

         <prop key="/welcome.htm">welcomeController</prop>

         <prop key="/listBooks.htm">listBooksController</prop>

         <prop key="/displayBookTOC.htm">displayBookTOCController</prop>

      </props>

   </property>

</bean>

<bean name="welcomeController"

 class="net.codejava.frameorks.spring.mvc.controller.WelcomeController"/>

<bean name="listBooksController"

 class="net.codejava.frameorks.spring.mvc.controller.ListBooksController"/>

<bean name="displayBookTOCController"

 class="net.codejava.frameorks.spring.mvc.controller.DisplayBookTOCController"/>

* **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 MainController{

@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.



* **ModelAndView and ViewResolver**

## ModelAndView

Spring’s MVC module encapsulates the model object and the view object in a single entity which is represented by the object of class ModelAndView. This object contains the model object and view object or the logical name of the view. The model object is the application data and the view is the object that renders the output to the user. The controller returns an object ofModelAndView to the dispatcher servlet for further processing.

## ViewResolver

In case ModelAndView object contains the logical name of the view then the DispatcherServlet needs resolving the view object based on its logical name. To resolve the view object, DispatcherServlet take the help of ViewResolver. There are number of implementation of view resolver provided by Spring. All the view resolvers implement the interfaceorg.springframework.web.servlet.ViewResolver.

### InternalResourceViewResolver

It resolves the logical name of the view to an internal resource by prefixing the logical view name with the resource path and suffixing it with the extension.

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

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

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

</bean>

If the logical name of the view returned by the controller in ModelAndView object is Welcome then the view which is shown to the user is /WEB-INF/jsp/Welcome.jsp

**BeanNameViewResolver**

It resolves the logical name of the view to the bean name which will render the output to the user. The bean should be defined in the Spring app context file. So if the logical name returned by the controller in ModelAndView object is Welcomethen the bean with name Welcome defined in the application context will be responsible to render the model to the user.

**XMLFileViewResolver**

This view resolver is the same as BeanNameViewResolver with only difference is that instead of looking for the beans in Spring’s application context file it looks for beans defined in a separate XML file (/WEB-INF/views.xml by default). The location and file name can be overridden by providing a location property while defining the XMLFileViewResolver.

<bean name="propertiesMethodNameResolver"

      class="org.springframework.web.servlet.view.XMLFileViewResolver">

      <propertyname="location">

            <value>classpath:myViews.xml</value>

      </property>

</bean>

**ResourceBundleViewResolver**

It resolves the logical name of the view to the actual view defined in the resource bundle. This view resolver takes basenameas the input which is the name of the property file where views can be located.

<bean name="propertiesMethodNameResolver"

      class="org.springframework.web.servlet.view.ResourceBundleViewResolver">

   <property name="basename">

      <value>myViews</value>

   </property>

</bean>

So if the logical name returned by the controller in ModelAndView object is Welcome then the view resolver will look for the property Welcome.class in properties file myViews.properties (or myViews\_en\_US.properties depending upon the user language and locale).

* **Spring MVC Form Model Implementation**

Create a Java class Product and under the com.musichub.furniture.modelpackage 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

packagecom.musichub.furniture.model;

importjavax.persistence.Entity;

@Entity

public class Product {

Integer pid;

String pname;

String brand;

String ptype;

String price;

public Integer getPid() {

returnpid;

}

public void setPid(Integer pid) {

this.pid = pid;

}

public String getPname() {

returnpname;

}

public void setPname(String pname) {

this.pname = pname;

}

public String getBrand() {

return brand;

}

public void setBrand(String brand) {

this.brand = brand;

}

public String getPtype() {

returnptype;

}

public void setPtype(String ptype) {

this.ptype = ptype;

}

public String getPrice() {

return price;

}

public void setPrice(String price) {

this.price = price;

}

}

In the Main Controller class we will use the model object to persist data.

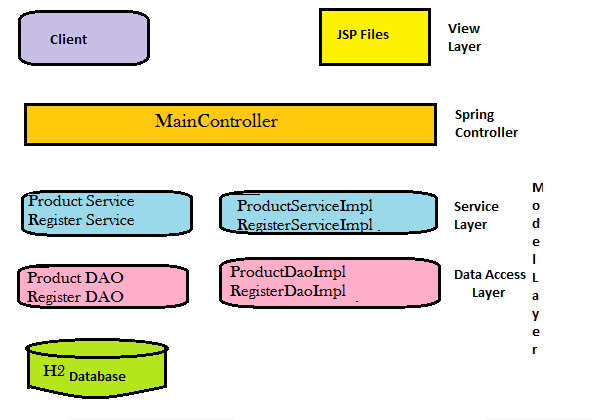
@RequestMapping(value = "/product", method = RequestMethod.GET)

publicModelAndViewproduct() {

return new ModelAndView("employeeForm", "command", new Product());

}

***CRUD Operations Using Spring-Hibernate Integration***

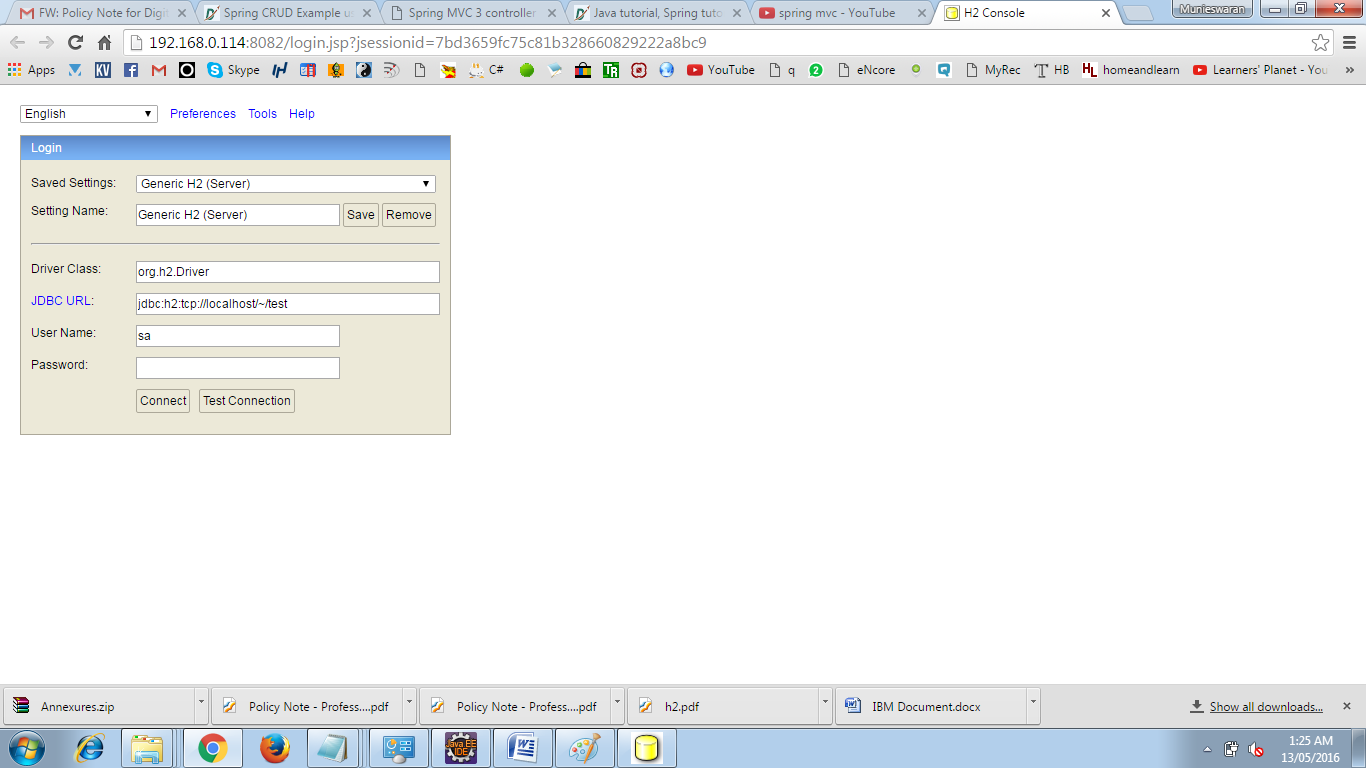


**Files used**

* 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

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-orm</artifactId>

<version>4.2.5.RELEASE</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>5.1.0.Final</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-entitymanager</artifactId>

<version>5.1.0.Final</version>

</dependency>

<dependency>

<groupId>javax.persistence</groupId>

<artifactId>persistence-api</artifactId>

<version>1.0.2</version>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<version>1.4.191</version>

</dependency>

<dependency>

<groupId>org.hsqldb</groupId>

<artifactId>hsqldb</artifactId>

<version>2.3.1</version>

</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.

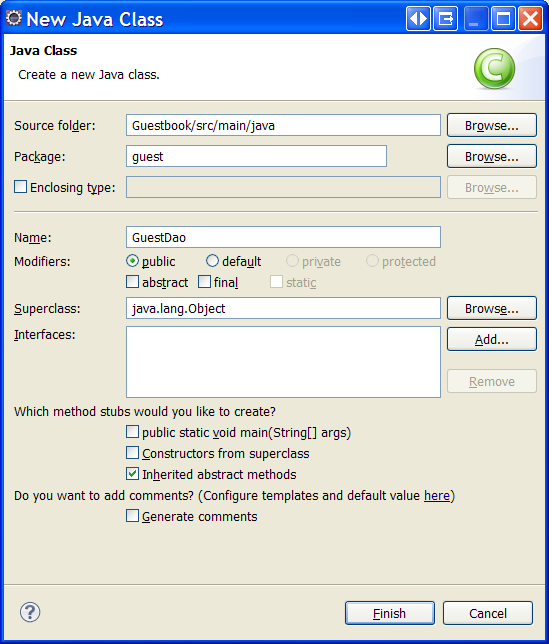
**@Column Annotation:**

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 Classes**

Operations on the database will be performed by an instance of a Data Access Object (DAO) that we will define in this step as a Spring MVC component:

* Open the [New Java Class] dialog box by right clicking the guest package node (in the [Package Explorer] window) and selecting **New > Class**.
* The package name should be **guest**.
* Enter **GuestDao** as the class name - use **exactly** that case sensitive class name.
* Click **Finish** to create the new DAO Spring component class.



Now replace the content of the new source file with the following code:

package guest;

import java.util.List;

import javax.persistence.EntityManager;

import javax.persistence.PersistenceContext;

import javax.persistence.TypedQuery;

import org.springframework.stereotype.Component;

import org.springframework.transaction.annotation.Transactional;

@Component

public class GuestDao {

// Injected database connection:

@PersistenceContext private EntityManager em;

// Stores a new guest:

@Transactional

public void persist(Guest guest) {

em.persist(guest);

}

// Retrieves all the guests:

public List<Guest> getAllGuests() {

TypedQuery<Guest> query = em.createQuery(

"SELECT g FROM Guest g ORDER BY g.id", Guest.class);

return query.getResultList();

}

}

The GuestDao Spring component class defines two methods:

* persist - for storing a new Guest entity object in the database.
* getAllGuests - for retrieving all the existing Guest objects from the database.

By using Spring components we can move some work from the application to the Spring Framework. For instance, in this example the Spring container:

* Manages the instantiation of the DAO component class and injects an instance of the DAO component class into the controller, as shown in the next step.
* Prepares an EntityManager automatically and injects it into the em field (because it is annotated with the @PersistenceContext annotation).
* Handles transactions automatically for methods that are annotated with the @Transactionalannotation, saving the need to wrap every operation that modifies the database (such as persist) by transaction begin and commit.

@Repository – Indicates DAO component in the persistence layer

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

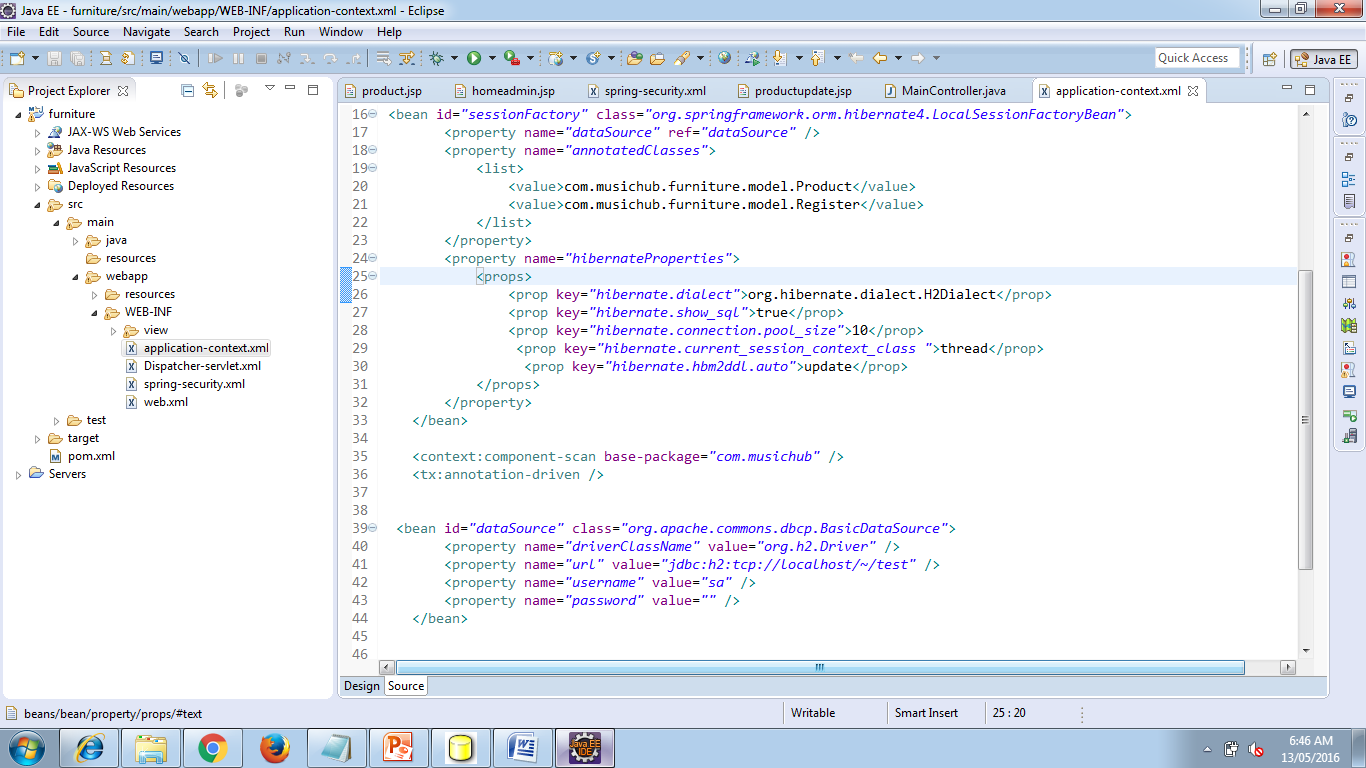
@Autowired annotation provides more fine-grained control over where and how autowiring should be accomplished. The @Autowired annotation can be used to autowire bean on the setter method just like @Required annotation, constructor, a property or methods with arbitrary names and/or multiple arguments.

* **Implementing SessionFactory**

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 Hibernate Configuration File**

In hibernate framework, we provide all the database information hibernate.cfg.xml file. But if we are going to integrate the hibernate application with spring, we don't need to create the hibernate.cfg.xml file. We can provide all the information in the applicationContext.xml file. The Spring framework provides **HibernateTemplate** class, so you don't need to follow so many steps like create Configuration, BuildSessionFactory, Session, beginning and committing transaction etc.

**employee.hbm.xml**

This mapping file contains all the information of the persistent class.

<?xml version='1.0' encoding='UTF-8'?>

<!DOCTYPE hibernate-mapping PUBLIC

"-//Hibernate/Hibernate Mapping DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

<**class** name="com.javatpoint.Employee" table="emp558">

          <id name="id">

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

         </id>

          <property name="name"></property>

          <property name="salary"></property>

</**class**>

</hibernate-mapping>

**applicationContext.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: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">

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

     <property name="driverClassName"  value="oracle.jdbc.driver.OracleDriver"></property>

<property name="url" value="jdbc:oracle:thin:@localhost:1521:xe"></property>

 <property name="username" value="system"></property>

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

</bean>

     <bean id="mysessionFactory"  **class**="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

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

     <property name="mappingResources">

 <list>

 <value>employee.hbm.xml</value>

 </list>

 </property>

<property name="hibernateProperties">

   <props>

 <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>

 <prop key="hibernate.hbm2ddl.auto">update</prop>

<prop key="hibernate.show\_sql">**true**</prop>

</props>

 </property>

</bean>

    <bean id="template" **class**="org.springframework.orm.hibernate3.HibernateTemplate">

<property name="sessionFactory" ref="mysessionFactory"></property>

</bean>

<bean id="d" **class**="com.javatpoint.EmployeeDao">

<property name="template" ref="template"></property>

 </bean>

 </beans>

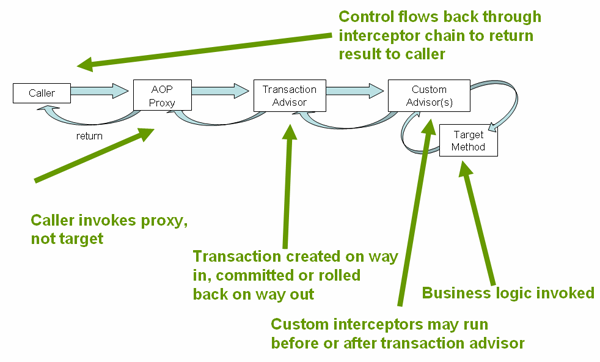
* **Implementing Transaction Manager**

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.

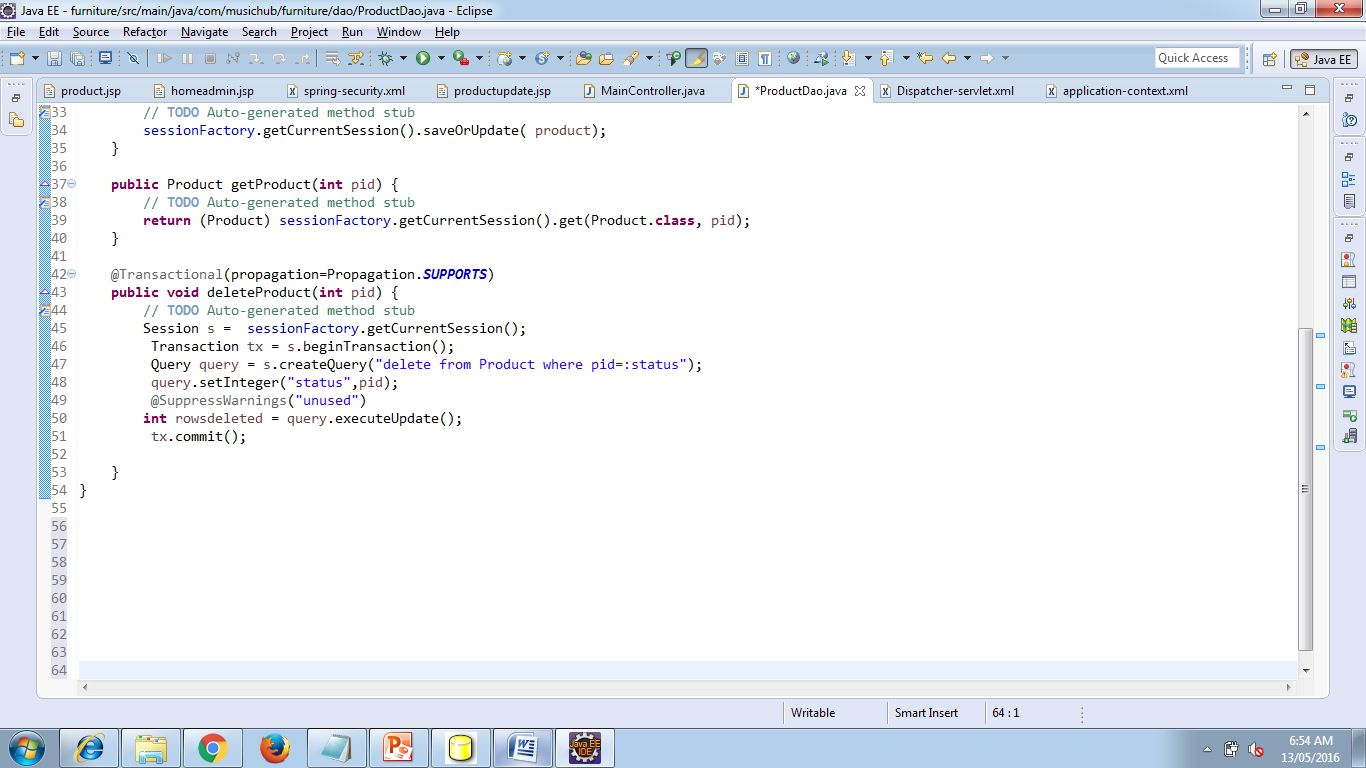
<beanid=*"transactionManager"*

class=*"org.springframework.orm.hibernate4.HibernateTransactionManager"*> <propertyname=*"sessionFactory"*ref=*"sessionFactory"*/>

</bean>



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.



***Implementing Spring Security :***

**Application Security Areas:**

There are two main areas for application securities.

1. **Authentication:** Process of checking the user, who they claim to be.
2. **Authorization:** Process of deciding whether an user is allowed to perform an activity within the application.

**Authentication Models supported by Spring Security:**

Spring security supports more then 20 Models for authentication. Some of them are

1. X.509 client certificate exchange
2. LDAP Authentication
3. OpenID authentication
4. Java Open Source Single Sign On

* **Spring Security Modules**

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:

<properties>

<org.springframework.version>3.0.5.RELEASE</org.springframework.version>

</properties>

<!-- Spring Security -->

<dependency>

<groupid>org.springframework.security</groupid>

<artifactid>spring-security-core</artifactid>

<version>${org.springframework.version}</version>

<type>jar</type>

<scope>compile</scope>

</dependency>

<dependency>

<groupid>org.springframework.security</groupid>

<artifactid>spring-security-web</artifactid>

<version>${org.springframework.version}</version>

<type>jar</type>

<scope>compile</scope>

</dependency>

<dependency>

<groupid>org.springframework.security</groupid>

<artifactid>spring-security-config</artifactid>

<version>${org.springframework.version}</version>

<type>jar</type>

<scope>compile</scope>

</dependency>

<dependency>

<groupid>org.springframework.security</groupid>

<artifactid>spring-security-taglibs</artifactid>

<version>${org.springframework.version}</version>

<type>jar</type>

<scope>compile</scope>

</dependency>

**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.
* **Configure DelegatingFilterProxy in web.xml**

Spring Security's web infrastructure is based entirely on standard servlet filters. It doesn't use servlets or any other servlet-based frameworks (such as Spring MVC) internally, so it has no strong links to any particular web technology. It deals in HttpServletRequests andHttpServletResponses and doesn't care whether the requests come from a browser, a web service client, an HttpInvoker or an AJAX application.

Spring Security maintains a filter chain internally where each of the filters has a particular responsibility and filters are added or removed from the configuration depending on which services are required. The ordering of the filters is important as there are dependencies between them. If you have been using [namespace configuration](http://docs.spring.io/spring-security/site/docs/3.0.x/reference/ns-config.html), then the filters are automatically configured for you and you don't have to define any Spring beans explicitly but here may be times when you want full control over the security filter chain, either because you are using features which aren't supported in the namespace, or you are using your own customized versions of classes.

When using servlet filters, you obviously need to declare them in your web.xml, or they will be ignored by the servlet container. In Spring Security, the filter classes are also Spring beans defined in the application context and thus able to take advantage of Spring's rich dependency-injection facilities and lifecycle interfaces. Spring's DelegatingFilterProxy provides the link between web.xml and the application context.

When using DelegatingFilterProxy, you will see something like this in the web.xml file:

<filter>

<filter-name>myFilter</filter-name>

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

</filter>

<filter-mapping>

<filter-name>myFilter</filter-name>

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

</filter-mapping>

Notice that the filter is actually a DelegatingFilterProxy, and not the class that will actually implement the logic of the filter. WhatDelegatingFilterProxy does is delegate the Filter's methods through to a bean which is obtained from the Spring application context. This enables the bean to benefit from the Spring web application context lifecycle support and configuration flexibility. The bean must implementjavax.servlet.Filter and it must have the same name as that in the filter-name element. Read the Javadoc for DelegatingFilterProxy for more information

* **Configuring Spring-Security.xml**

We have created a filter name as “springSecurityFilterChain” can help you configure the basic infrastructure using element.





**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.

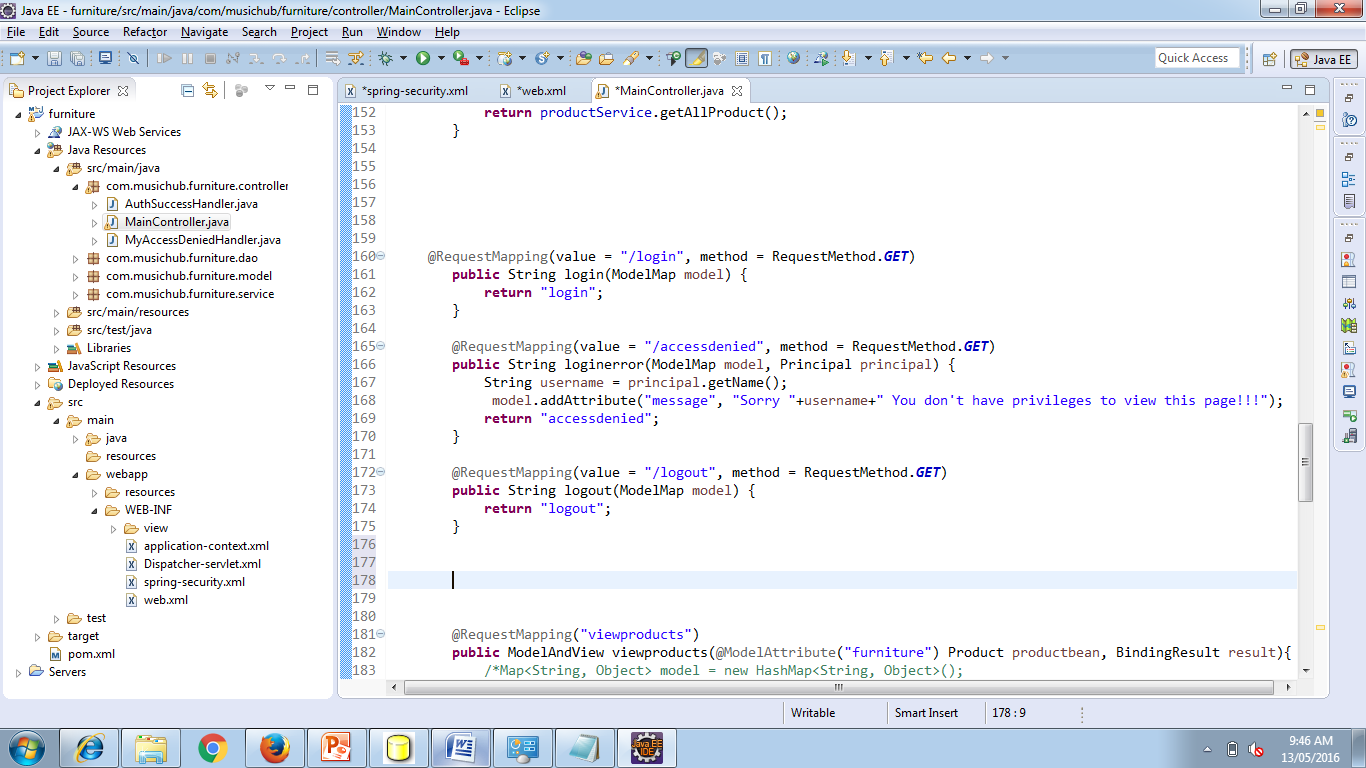
**form-login:** This will come into picture when user will try to access any secured URL. A login page mapped to “login-page” attribute will be served for authentication check. If not provided, spring will provide an inbuilt login page to user. It also contains attribute for default target if login success,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



**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.

**DelegatingFilterProxy**

<form name="f" action="<c:url value='j\_spring\_security\_check'/>"

method="POST">

<table>

<tr>

<td>Username:</td>

<td><input type='text' name='j\_username' /></td>

</tr>

<tr>

<td>Password:</td>

<td><input type='password' name='j\_password'></td>

</tr>

<tr>

<td colspan="2">&nbsp;</td>

</tr>

<tr>

<td colspan='2'><input name="submit" type="submit">&nbsp;<input name="reset" type="reset"></td>

</tr>

</table>

</form>

Next additional view is, **logout.jsp** file.

< % session.invalidate(); %> You are now logged out!!

<a href="${pageContext.request.contextPath}/login">go back</a>

This view simply invalidate the session and provide a link to go back to login page.

**denied.jsp**

This jsp file will come in user screen when user will try to authenticate with invalid user name and password combinations. It will show the corresponding message as configured in message.properties in your classpath.

<%@ tagliburi="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<html>

<body>

<h1 id="banner">Unauthorized Access !!</h1>

<hr />

<c:if test="${not empty error}">

<div style="color:red">

Your fake login attempt was bursted, dare again !!<br />

Caused : ${sessionScope["SPRING\_SECURITY\_LAST\_EXCEPTION"].message}

</div>

</c:if>

<p class="message">Access denied!</p>

<a href="/Spring3HibernateIntegration/login">Go back to login page</a>

</body>

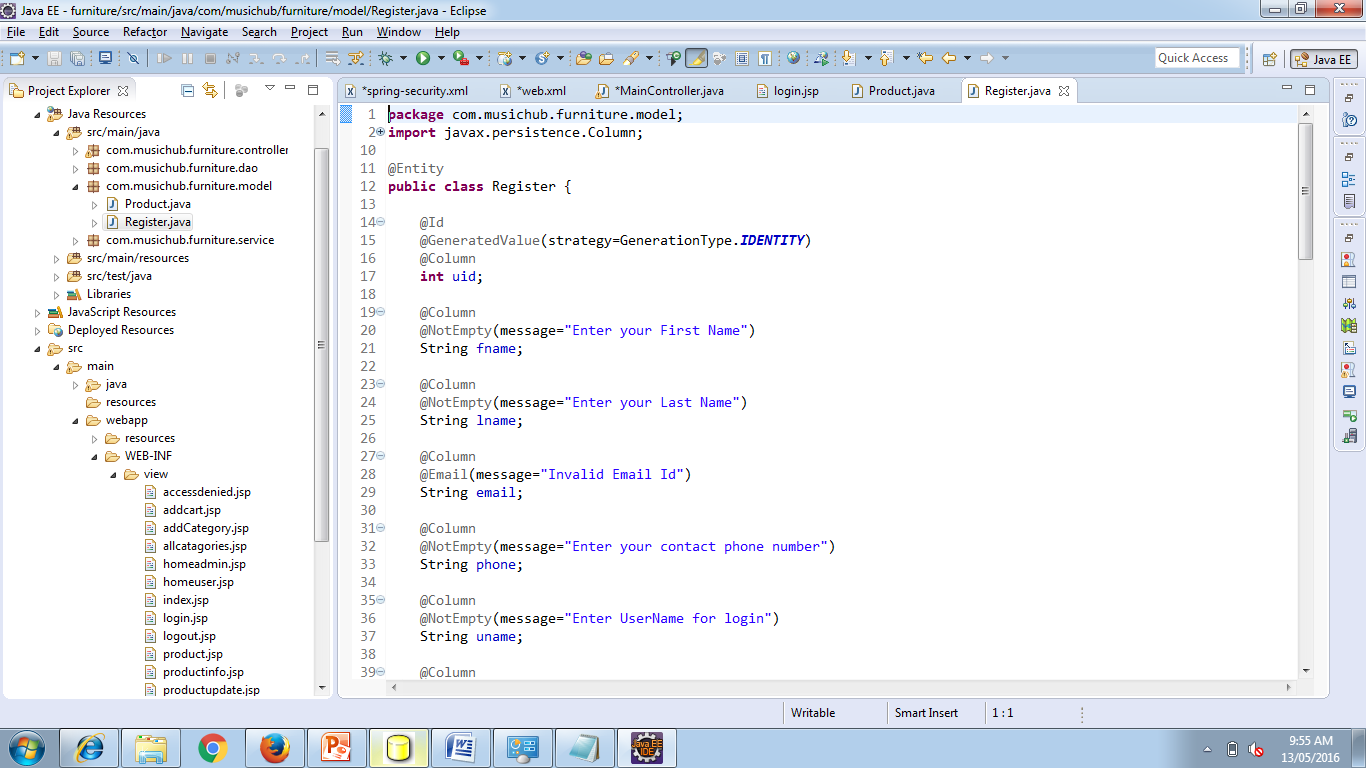
</html>

**Implementing Validation**

We have created Spring MVC application that take user input and checks the input using standard validation annotations. Nowwe 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>

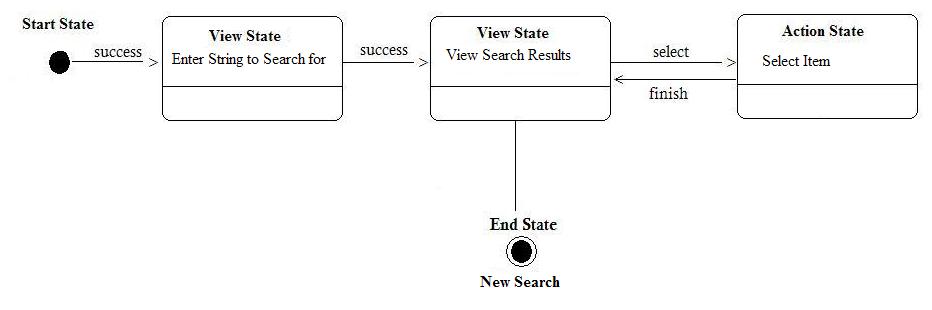
<artifactId>hibernate-validator</artifactId>

<version>5.1.0.Final</version>

</dependency>

***Implementing Spring Webflow :***

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.

***Implementing Multi Part Concept***

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;

importorg.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 orwithjavax.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.

src/main/java/hello/FileUploadController.java

packagecom.musichub.furniture.controller;

importorg.springframework.stereotype.Controller;

importorg.springframework.ui.Model;

importorg.springframework.web.bind.annotation.ModelAttribute;

importorg.springframework.web.bind.annotation.RequestMapping;

importorg.springframework.web.bind.annotation.RequestMethod;

importorg.springframework.web.multipart.MultipartFile;

importcom.javabeat.form.FileUploadForm;

@Controller

public class FileUploadController {

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

public String displayForm() {

return "file\_upload\_form";

}

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

public String save(@ModelAttribute("uploadForm") FileUploadFormuploadForm,

Model map) {

MultipartFilemultipartFile = uploadForm.getFile();

String fileName = "";

if (multipartFile != null) {

fileName = multipartFile.getOriginalFilename();

}

map.addAttribute("files", fileName);

return "file\_upload\_success";

}

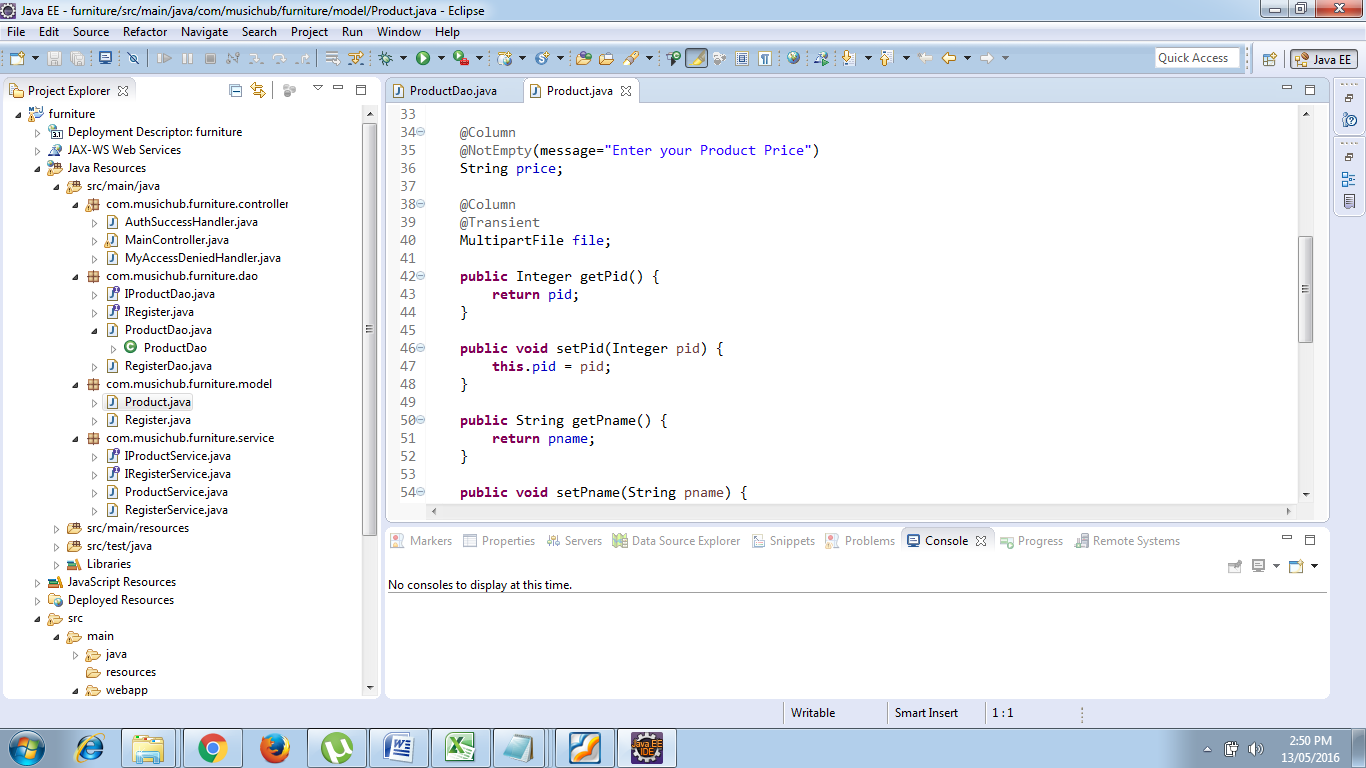
}

The entire class is marked up with @Controller so Spring MVC can pick it up and look for routes.

Each method is tagged with @RequestMapping to flag the path and the HTTP action. In this case, GET looks up the current list of uploaded files (stored in Application.ROOT folder) and load it into a Thymeleaf template.

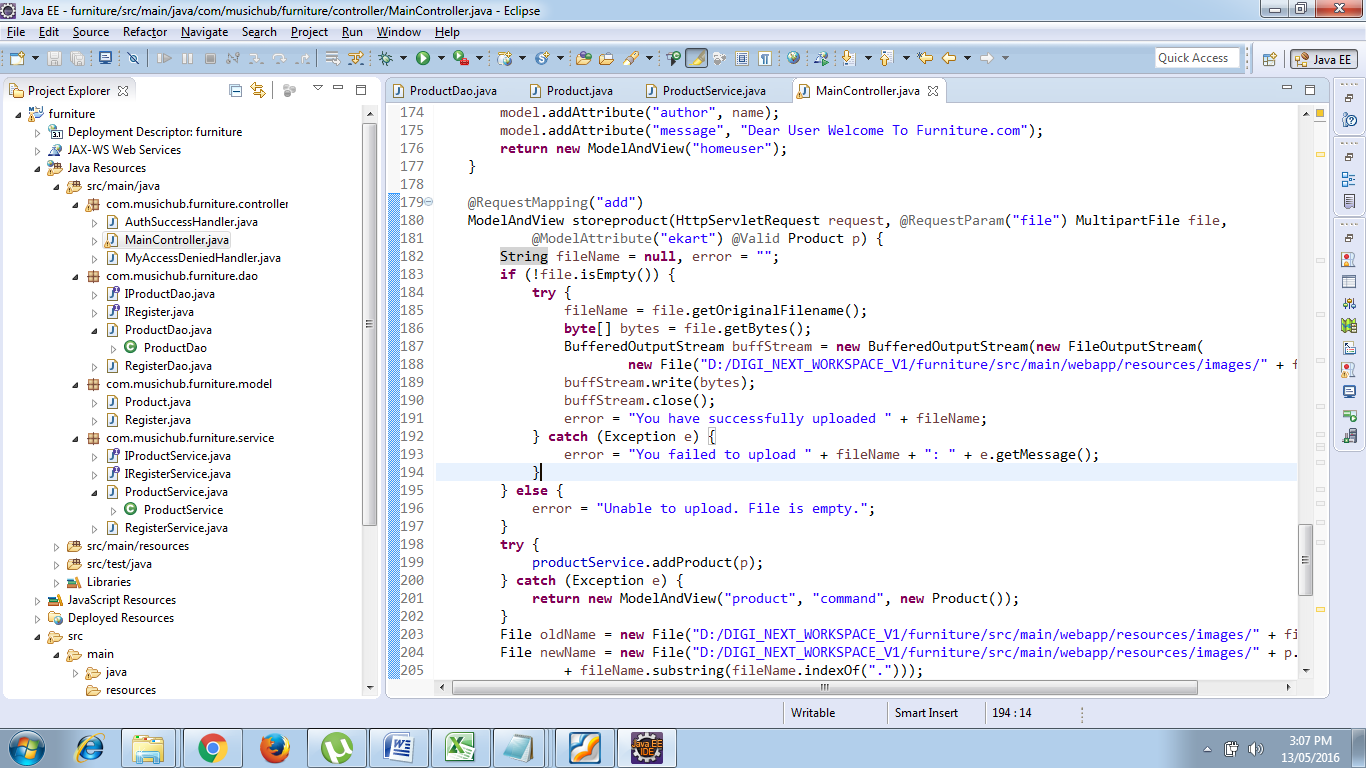
The handleFileUpload method is geared to handle a two-part message: name and file. It checks to make sure the file name contains no illegal characers, and then copies the bytes into a target file through a BufferedOutputStream.

## Updating the Model

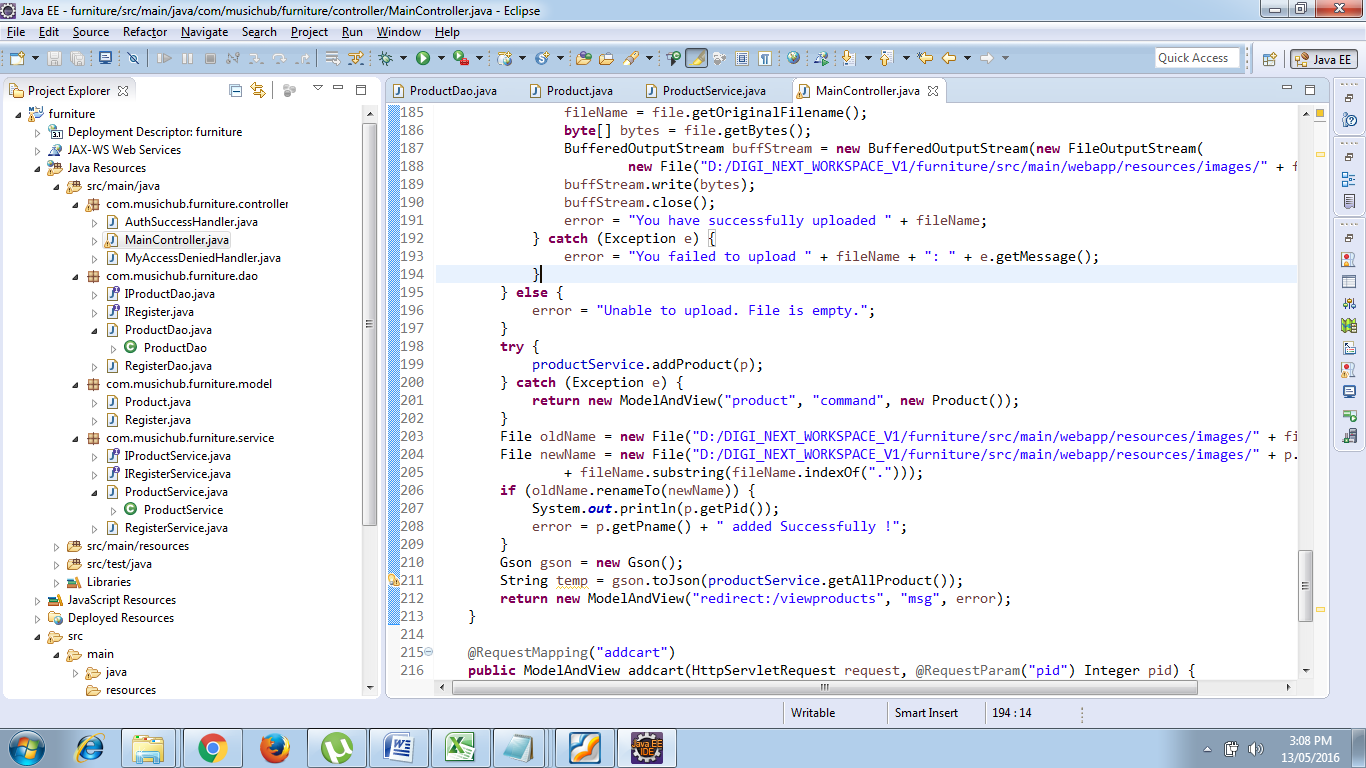


@Transient

MultipartFilefile; (import org.springframework.web.multipart.MultipartFile;)



**Updating the Main Controller**



**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.

***Use of Hibernate Validator :***

**Prerequisites**

* [Java Runtime](http://www.oracle.com/technetwork/java/index.html) >= 6
* [Apache Maven](http://maven.apache.org/)

**Project set up**

In order to use Hibernate Validator within a Maven project, simply add the following dependency to your pom.xml:

**<dependency>**

**<groupId>**org.hibernate**</groupId>**

**<artifactId>**hibernate-validator**</artifactId>**

**<version>**5.2.4.Final**</version>**

**</dependency>**

This transitively pulls in the dependency to the Bean Validation API (javax.validation:validation-api:{bv\_api}).

### Unified Expression Language (EL)

Hibernate Validator also requires an implementation of the Unified Expression Language ([JSR 341](http://jcp.org/en/jsr/detail?id=341)) for evaluating dynamic expressions in constraint violation messages. When your application runs in a Java EE container such as [WildFly](http://wildfly.org/), an EL implementation is already provided by the container. In a Java SE environment, however, you have to add an implementation as dependency to your POM file. For instance you can add the following two dependencies to use the JSR 341 [reference implementation](http://uel.java.net/):

**<dependency>**

**<groupId>**javax.el**</groupId>**

**<artifactId>**javax.el-api**</artifactId>**

**<version>**2.2.4**</version>**

**</dependency>**

**<dependency>**

**<groupId>**org.glassfish.web**</groupId>**

**<artifactId>**javax.el**</artifactId>**

**<version>**2.2.4**</version>**

**</dependency>**

### CDI

Bean Validation defines integration points with CDI (Contexts and Dependency Injection, [JSR 346](http://jcp.org/en/jsr/detail?id=346)). If your application runs in an environment which does not provide this integration out of the box, you may use the Hibernate Validator CDI portable extension by adding the following Maven dependency to your POM:

**<dependency>**

**<groupId>**org.hibernate**</groupId>**

**<artifactId>**hibernate-validator-cdi**</artifactId>**

**<version>**5.2.4.Final**</version>**

**</dependency>**

## Applying constraints

Lets dive into an example to see how to apply constraints:

**package** **org.hibernate.validator.referenceguide.chapter01**;

**import** **javax.validation.constraints.Min**;

**import** **javax.validation.constraints.NotNull**;

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

**public** **class** **Car** {

@NotNull

**private** **String** manufacturer;

@NotNull

@Size(min = 2, max = 14)

**private** **String** licensePlate;

@Min(2)

**private** **int** seatCount;

**public** Car(**String** manufacturer, **String** licencePlate, **int** seatCount) {

this.manufacturer = manufacturer;

this.licensePlate = licencePlate;

this.seatCount = seatCount;

}

// getters and setters ...

}

The @NotNull, @Size and @Min annotations are used to declare the constraints which should be applied to the fields of a Car instance:

* manufacturer must never be null
* licensePlate must never be null and must be between 2 and 14 characters long
* seatCount must be at least 2

|  |  |
| --- | --- |
|  | You can find the complete source code of all examples used in this getting started guide in the Hibernate Validator[source repository](https://github.com/hibernate/hibernate-validator/tree/master/documentation/src/test)on GitHub. |

## Validating constraints

To perform a validation of these constraints, you use a Validator instance. To demonstrate this, let’s have a look at a simple unit test:

**package** **org.hibernate.validator.referenceguide.chapter01**;

**import** **java.util.Set**;

**import** **javax.validation.ConstraintViolation**;

**import** **javax.validation.Validation**;

**import** **javax.validation.Validator**;

**import** **javax.validation.ValidatorFactory**;

**import** **org.junit.BeforeClass**;

**import** **org.junit.Test**;

**import** **static** **org.junit.Assert.assertEquals**;

**public** **class** **CarTest** {

**private** **static** **Validator** validator;

@BeforeClass

**public** **static** **void** setUp() {

ValidatorFactory factory = Validation.buildDefaultValidatorFactory();

validator = factory.getValidator();

}

@Test

**public** **void** manufacturerIsNull() {

Car car = **new** Car( null, "DD-AB-123", 4 );

**Set**<ConstraintViolation<Car>> constraintViolations =

validator.validate( car );

assertEquals( 1, constraintViolations.size() );

assertEquals(

"may not be null",

constraintViolations.iterator().next().getMessage()

);

}

@Test

**public** **void** licensePlateTooShort() {

Car car = **new** Car( "Morris", "D", 4 );

**Set**<ConstraintViolation<Car>> constraintViolations =

validator.validate( car );

assertEquals( 1, constraintViolations.size() );

assertEquals(

"size must be between 2 and 14",

constraintViolations.iterator().next().getMessage()

);

}

@Test

**public** **void** seatCountTooLow() {

Car car = **new** Car( "Morris", "DD-AB-123", 1 );

**Set**<ConstraintViolation<Car>> constraintViolations =

validator.validate( car );

assertEquals( 1, constraintViolations.size() );

assertEquals(

"must be greater than or equal to 2",

constraintViolations.iterator().next().getMessage()

);

}

@Test

**public** **void** carIsValid() {

Car car = **new** Car( "Morris", "DD-AB-123", 2 );

**Set**<ConstraintViolation<Car>> constraintViolations =

validator.validate( car );

assertEquals( 0, constraintViolations.size() );

}

}

In the setUp() method a Validator instance is retrieved from the ValidatorFactory. Validator instances are thread-safe and may be reused multiple times.

The validate() method returns a set of ConstraintViolation instances, which you can iterate in order to see which validation errors occurred. The first three test methods show some expected constraint violations:

* The @NotNull constraint on manufacturer is violated in manufacturerIsNull()
* The @Size constraint on licensePlate is violated in licensePlateTooShort()
* The @Min constraint on seatCount is violated in seatCountTooLow()

If the object validates successfully, validate() returns an empty set as you can see in carIsValid().

Note that only classes from the package javax.validation are used. These are provided from the Bean Validation API. No classes from Hibernate Validator are directly referenced, resulting in portable code.

## What is a constraint?

A constraint is a rule that a given element (field, property or bean) has to comply to. The rule semantic is expressed by an annotation. A constraint usually has some attributes used to parameterize the constraints limits. The constraint applies to the annotated element.

## Built in constraints

Hibernate Validator comes with some built-in constraints, which covers most basic data checks. As we'll see later, you're not limited to them, you can literally in a minute write your own constraints.

**Table 2.1. Built-in constraints**

| **Annotation** | **Apply on** | **Runtime checking** | **Hibernate Metadata impact** |
| --- | --- | --- | --- |
| @Length(min=, max=) | property (String) | check if the string length match the range | Column length will be set to max |
| @Max(value=) | property (numeric or string representation of a numeric) | check if the value is less than or equals to max | Add a check constraint on the column |
| @Min(value=) | property (numeric or string representation of a numeric) | check if the value is more than or equals to min | Add a check constraint on the column |
| @NotNull | property | check if the value is not null | Column(s) are not null |
| @NotEmpty | property | check if the string is not null nor empty. Check if the connection is not null nor empty | Column(s) are not null (for String) |
| @Past | property (date or calendar) | check if the date is in the past | Add a check constraint on the column |
| @Future | property (date or calendar) | check if the date is in the future | none |
| @Pattern(regex="regexp", flag=) or @Patterns( {@Pattern(...)} ) | property (string) | check if the property match the regular expression given a match flag (see **java.util.regex.Pattern**  ) | none |
| @Range(min=, max=) | property (numeric or string representation of a numeric) | check if the value is between min and max (included) | Add a check constraint on the column |
| @Size(min=, max=) | property (array, collection, map) | check if the element size is between min and max (included) | none |
| @AssertFalse | property | check that the method evaluates to false (useful for constraints expressed in code rather than annotations) | none |
| @AssertTrue | property | check that the method evaluates to true (useful for constraints expressed in code rather than annotations) | none |
| @Valid | property (object) | perform validation recursively on the associated object. If the object is a Collection or an array, the elements are validated recursively. If the object is a Map, the value elements are validated recursively. | none |
| @Email | property (String) | check whether the string is conform to the email address specification | none |
| @CreditCardNumber | property (String) | check whether the string is a well formated credit card number (derivative of the Luhn algorithm) | none |
| @Digits(integerDigits=1) | property (numeric or string representation of a numeric) | check whether the property is a number having up to **integerDigits** integer digits and **fractionalDigits**fractonal digits | define column precision and scale |
| @EAN | property (string) | check whether the string is a properly formated EAN or UPC-A code | none |

## Error messages

Hibernate Validator comes with a default set of error messages translated in about ten languages (if yours is not part of it, please sent us a patch). You can override those messages by creating a **ValidatorMessages.properties** or ( **ValidatorMessages\_loc.properties** ) and override the needed keys. You can even add your own additional set of messages while writing your validator annotations. If Hibernate Validator cannot resolve a key from your resourceBundle nor from ValidatorMessage, it falls back to the default built-in values.

Alternatively you can provide a **ResourceBundle** while checking programmatically the validation rules on a bean or if you want a completly different interpolation mechanism, you can provide an implementation of **org.hibernate.validator.MessageInterpolator** (check the JavaDoc for more informations).

## Annotating your domain model

Since you are already familiar with annotations now, the syntax should be very familiar

public class Address {

private String line1;

private String line2;

private String zip;

private String state;

private String country;

private long id;

// a not null string of 20 characters maximum

@Length(max=20)

@NotNull

public String getCountry() {

return country;

}

// a non null string

@NotNull

public String getLine1() {

return line1;

}

//no constraint

public String getLine2() {

return line2;

}

// a not null string of 3 characters maximum

@Length(max=3) @NotNull

public String getState() {

return state;

}

// a not null numeric string of 5 characters maximum

// if the string is longer, the message will

//be searched in the resource bundle at key 'long'

@Length(max=5, message="{long}")

@Pattern(regex="[0-9]+")

@NotNull

public String getZip() {

return zip;

}

// should always be true

@AssertTrue

public boolean isValid() {

return true;

}

// a numeric between 1 and 2000

@Id @Min(1)

@Range(max=2000)

public long getId() {

return id;

}

}

While the example only shows public property validation, you can also annotate fields of any kind of visibility

@MyBeanConstraint(max=45)

public class Dog {

@AssertTrue private boolean isMale;

@NotNull protected String getName() { ... };

...

}}

You can also annotate interfaces. Hibernate Validator will check all superclasses and interfaces extended or implemented by a given bean to read the appropriate validator annotations.

public interface Named {

@NotNull String getName();

}

public class Dog implements Named {

@AssertTrue private boolean isMale;

public String getName() { ... };}

The name property will be checked for nullity when the Dog bean is validated.

## Database schema-level validation

Out of the box, Hibernate Annotations will translate the constraints you have defined for your entities into mapping metadata. For example, if a property of your entity is annotated **@NotNull**, its columns will be declared as **not null** in the DDL schema generated by Hibernate.

Using hbm2ddl, domain model constraints will be expressed into the database schema.

If, for some reason, the feature needs to be disabled, set **hibernate.validator.apply\_to\_ddl** to **false**.

### Hibernate event-based validation

Hibernate Validator has two built-in Hibernate event listeners. Whenever a **PreInsertEvent** or **PreUpdateEvent**occurs, the listeners will verify all constraints of the entity instance and throw an exception if any constraint is violated. Basically, objects will be checked before any inserts and before any updates made by Hibernate. This includes changes applied by cascade! This is the most convenient and the easiest way to activate the validation process. On constraint violation, the event will raise a runtime **InvalidStateException** which contains an array of **InvalidValue**s describing each failure.

If Hibernate Validator is present in the classpath, Hibernate Annotations (or Hibernate EntityManager) will use it transparently. If, for some reason, you want to disable this integration, set **hibernate.validator.autoregister\_listeners** to false

**Note**

If the beans are not annotated with validation annotations, there is no runtime performance cost.

In case you need to manually set the event listeners for Hibernate Core, use the following configuration in **hibernate.cfg.xml**:

<hibernate-configuration>

...

<event type="pre-update">

<listener

class="org.hibernate.validator.event.ValidateEventListener"/>

</event>

<event type="pre-insert">

<listener

class="org.hibernate.validator.event.ValidateEventListener"/>

</event>

</hibernate-configuration>

### Java Persistence event-based validation

Hibernate Validator is not tied to Hibernate for event based validation: a Java Persistence entity listener is available. Whenever an listened entity is persisted or updated, Hibernate Validator will verify all constraints of the entity instance and throw an exception if any constraint is violated. Basically, objects will be checked before any inserts and before any updates made by the Java Persistence provider. This includes changes applied by cascade! On constraint violation, the event will raise a runtime **InvalidStateException** which contains an array of **InvalidValue**s describing each failure.

Here is how to make a class validatable:

@Entity

@EntityListeners( JPAValidateListener.class )

public class Submarine {

...}

**Note**

Compared to the Hibernate event, the Java Persistence listener has two drawbacks. You need to define the entity listener on every validatable entity. The DDL generated by your provider will not reflect the constraints.

## Application-level validation

Hibernate Validator can be applied anywhere in your application code.

ClassValidator personValidator = new ClassValidator( Person.class );

ClassValidator addressValidator = new ClassValidator( Address.class, ResourceBundle.getBundle("messages", Locale.ENGLISH) );

InvalidValue[] validationMessages = addressValidator.getInvalidValues(address);

The first two lines prepare the Hibernate Validator for class checking. The first one relies upon the error messages embedded in Hibernate Validator (see [Section 2.3, “Error messages”](https://access.redhat.com/documentation/en-US/JBoss_Enterprise_Web_Platform/5/html-single/Hibernate_Validator_Reference_Guide/index.html#validator-defineconstraints-error)), the second one uses a resource bundle for these messages. It is considered a good practice to execute these lines once and cache the validator instances.

The third line actually validates the **Address** instance and returns an array of **InvalidValue**s. Your application logic will then be able to react to the failure.

You can also check a particular property instead of the whole bean. This might be useful for property per property user interaction

ClassValidator addressValidator = new ClassValidator( Address.class, ResourceBundle.getBundle("messages", Locale.ENGLISH) );

//only get city property invalid values

InvalidValue[] validationMessages = addressValidator.getInvalidValues(address, "city");

//only get potential city property invalid values

InvalidValue[] validationMessages = addressValidator.getPotentialInvalidValues("city", "Paris");

## Presentation layer validation

When working with JSF and JBoss Seam , one can triggers the validation process at the presentation layer using Seam's JSF tags **<s:validate>** and **<s:validateAll/>**, letting the constraints be expressed on the model, and the violations presented in the view

<h:form>

<div>

<h:messages/>

</div>

**<s:validateAll>**

<div>

Country:

<h:inputText value="#{location.country}" required="true"/>

</div>

<div>

Zip code:

<h:inputText value="#{location.zip}" required="true"/>

</div>

<div>

<h:commandButton/>

</div>

**</s:validateAll>**

</h:form>

Going even further, and adding Ajax4JSF to the loop will bring client side validation with just a couple of additional JSF tags, again without validation definition duplication.

Check the [JBoss Seam](http://www.jboss.com/products/seam) documentation for more information.

## Validation informations

As a validation information carrier, hibernate provide an array of **InvalidValue**. Each **InvalidValue** has a buch of methods describing the individual issues.

* **getBeanClass()** retrieves the failing bean type
* **getBean()**retrieves the failing instance (if any ie not when using **getPotentianInvalidValues()**)
* **getValue()** retrieves the failing value
* **getMessage()** retrieves the proper internationalized error message
* **getRootBean()** retrieves the root bean instance generating the issue (useful in conjunction with **@Valid**), is null if getPotentianInvalidValues() is used.
* **getPropertyPath()** retrieves the dotted path of the failing property starting from the root bean