

Contents

Spring Framework	3
Spring Beans.....	4
Spring Framework Features:	4
Inversion of Control and Dependency Injection	5
Characteristic of Dependency Injection and the IOC.....	6
More Technique to achieve Inversion of Control.....	7
How inversion of control using dependency injection works in Spring.....	8
Mechanics of Dependency Injection.....	9
The advantage of inversion of control using dependency injection	10
Spring Module	11
Dependency Management	13
Model View Controller	14
MVC Flows :.....	14
Model component in the MVC paradigm:	14
Controller component in the MVC paradigm:.....	15
View component in the MVC paradigm :	15
MVC Paradigm.....	16
Spring MVC Technologies	17
Spring MVC heavily relies on three underlying technology:	17
Servlets:.....	17
Here is a big picture visualization of how the servlet architecture enhances your web server application.....	18
Servlets and Servlets Engine	18
JSP	18
JSTL, JSP Standard Tag Library.....	19
JSTL Tag Classifications	20
Spring MVC Architecture	21
Typical Spring MVC App	21
Role of Dispatcher Servlet.....	22
The Flow of a request using a high level component diagram.....	23
Another way of the structure of a Spring MVC application.	24
Setup Maven Project on Eclipse.....	26

Basic structure of this Maven project.....	32
Adding Spring MVC Dependencies in Maven Project.....	34
Specifying Application Context Programmatically	35
Running a Spring MVC Application Within Eclipse	40
Specifying Application Context Using XML	44
Using Annotation for Bean Specification	48
Configuring the View Resolver	53
Summary.....	57
Quiz.....	58

Spring Framework

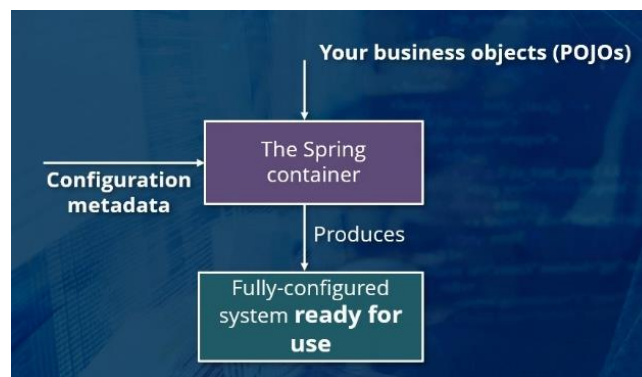
Spring Framework is a Java Framework that provides a convenient way of integrating disparate components into working, enterprise grade application.

The Spring framework :

- A comprehensive programming and configuration model for modern Java-based enterprise applications, which can work on any kind of deployment platform.
- As infrastructural support at the application level. Spring focuses on bringing your disparate elements together. The Spring framework itself can be thought of as an infrastructure platform that brings all of these disparate components together and gets them working together as a whole in an application
- goal is to focus on application level business logic without worrying about how all of them fit together. The Spring framework comprises of many projects, each of which is a separate module or a component, and it makes one aspect of your application development easy.
- Relies heavily on the principle of **inversion of control** to bring together disparate components to work as a single application.

Inversion of control is a principal in software engineering by which the control of objects or portions of a program is transferred to the container or framework within which the program runs.

High level visualization of what the Spring framework offers :



1. The core of Spring is the Spring container. This is the module that follows the principle of inversion of control.
2. Your simple business objects, which are plain old Java objects, form an input into this Spring container. Your objects can be configured using configuration metadata which also forms an input into the Spring container module.
3. The Spring container brings your configuration and your POJOs together to produce a fully configured system that is ready for use.

Spring Beans

Spring beans are just Java objects that form the backbone of your Spring application and are managed by Spring. Any object whose entire lifecycle is managed by Spring is referred to as a Spring bean. A Spring bean is an object that you may have defined, but you've basically asked Spring to take care of its lifecycle at runtime.

- Spring beans are the **basic building block** of the Spring framework, which Spring is responsible for managing these objects at runtime.
- Spring is responsible for **creating** or **constructing** these bean objects and **destroying** these bean objects when they are no longer needed.
- Spring is responsible for providing the **dependencies** for every Spring bean, Spring is also responsible for **configuring** the Spring bean, based on the configuration properties that you have specified.
- Spring takes care of intercepting bean method calls to include other framework features or to enhance bean with additional functionality.

Spring Framework Features:

- An important feature of the Spring framework is how it manages dependencies.
- Help Developer to follow and implementing a formalized best practice of design patterns.
- Spring relies on dependency injection to free developer on figuring out how to manage the dependencies of any component within larger application.

The principle of inversion of control means the burden of managing dependencies is passed on to the Spring container rather than the developer handling it himself or herself. Spring relies on dependency injection to achieve inversion of control.

Inversion of control is a process by which **objects** which hold business logic just **define** their **dependencies** and an external container is responsible for **injecting** those dependencies into the object. The object doesn't need to worry about where its dependencies come from or how they are constructed.

The object only has to declare **what it's dependent on**. But it's not responsible for creating or constructing those dependent objects or figuring out where they come from.

Inversion of Control and Dependency Injection

The Spring framework uses the **inversion of control** software engineering principle in order to manage the dependencies between the components within your enterprise grade application.

Inversion of control in Spring is achieved via a technique called **dependency injection**.

Inversion of control is where an object specifies what it's dependent on, and the container is responsible for constructing and injecting that dependent object.

Basic Component involved in setting up this inversion of control framework in Spring

- **Bean**
Object that depends on behavior of another object
example : an email verification component depends on some object which needs to access your underlying database to checks to see that no other user with the same email address has registered with your website or application.
- **Service**
Object that client bean depends upon
example : The dependent object is the one that can access the database to check for duplicate email IDs.
- **Injector**
Code that tells client bean which service to use.
The injector is the Spring core container which is responsible for providing this inversion of control.
The injector is not something that you need to implement, unlike the bean or the service.
The injector is a part of the Spring framework and it tells the client bean which service meets its requirements, which service it ought to use.
example : A bean declares its dependencies on services. And it's the injector that is responsible for telling the client bean which service it ought to use in this particular case.
- **Actual Process of Injection**
Injection refers to the process of telling the client bean which service to use.
The actual implementation, how exactly it takes place, is referred to as injection.

Characteristic of Dependency Injection and the IOC

- Service becomes part of state of client bean
The process of injection basically allows the service that the client bean needs to become a part of the state of the client bean. The service is injected into the bean, the bean can then use the service as it wishes to.
- The client bean cannot choose, find or build the service that it needs.
It doesn't know how to construct the object that provides the service, it doesn't know where exactly that service comes from.
It just knows that it needs the service, it declares its intent, and the container is responsible for giving it the right implementation.
- Client bean delegates this responsibility to injector
The client bean essentially tells this Spring core container which provides inversion of control which is the injector here, I need the service. Figure out a way to get it to me, make sure that I have it, so that I can perform my operation. So the client bean delegates the responsibility of choosing, finding, and building the service to the injector.
- Client bean DOES need to know interface of service
What the client bean does need to know is the interface of the service that it wants to use, the interface of the service that it depends on. Using this interface, the injector will figure out the right implementation of the service to inject into the client bean.

Inversion of control is a software design principle where beans or clients delegate the responsibility of managing dependencies to an external service. This is the Spring Container.

More Technique to achieve Inversion of Control

1. **Dependency Injection**
Spring achieves IoC using this approach (the best way)
2. **Service Locator Pattern**
where you lay the responsibility on beans to find their own service objects
The beans here are not responsible for constructing the services that they depend on, but they are responsible for finding the right service.
It requires that the developer of the bean knows of the existence and implementations of other components in the system.
3. **Factory pattern.**
This is where the responsibility is on the implementor of the bean to actually construct the service that the implementor depends on and this is typically done using some kind of factory method.

How inversion of control using dependency injection works in Spring.

- Client beans are created via factory (or sometimes via constructor).
The creation of these beans is handled by implementation of Bean Factory Interface and Application Context Sub-Interface.
- Spring is responsible for instantiating, configuring, and assembling beans.
And for common beans, that is common components that every application has to use, Spring provides implementations of these commonly used components out of the box. You don't have to build your own.
- Once beans objects are constructed, dependencies are injected
Spring is responsible for constructing those objects. And once these bean objects have been constructed, dependencies for every bean are injected into that bean. Each of these beans or components are responsible for declaring what they're dependent on. And Spring will make sure that the right implementation is injected.
- IoC Container: Spring framework to instantiate beans and inject dependencies
The IoC container is a part of the core Spring module to instantiate beans and inject dependencies.

Mechanics of Dependency Injection

- **Constructor Arguments (If object created via constructor)**
It specifies the constructor arguments for an object if the object is created using a constructor.
- **Arguments passed to (static or instance) factory method.**
These arguments may also be passed in to a factory method, either a static factory or an instance factory in order to construct objects.
- **Properties set on object after instantiation.**
Once an object has been constructed, additional properties are set on the object, this is after construction or instantiation.
- **Spring supports XML-based declarative instantiation**
Specification of bean properties and dependencies may be done in XML configuration files.

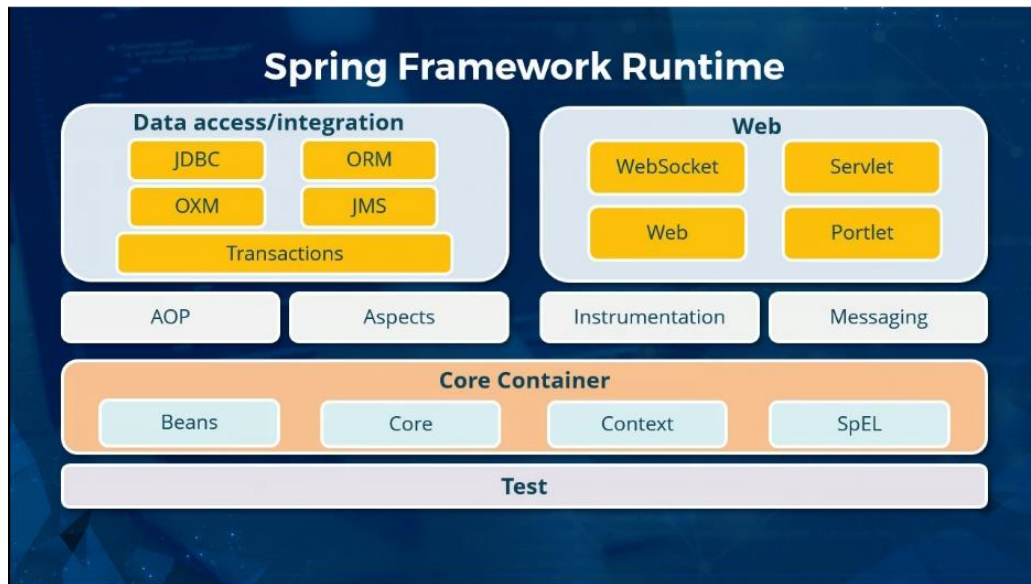
IOC Container : The Core of the Spring framework responsible for the entire **lifecycle** of Spring Beans

What is inversion of control? The injector calls the client to inject dependencies, the client does not call the injector.

The advantage of inversion of control using dependency injection

1. Provides separation of responsibilities.
There is a clean abstraction between the client bean and service objects.
The client bean is only aware of the interface of the service that it needs to use.
It has no knowledge of the underlying implementation.
The implementation can be changed without affecting clients.
2. Promotes code reuse and modularity.
That's because the bean and the service themselves will be a lot simpler when they have no knowledge of how the other component works.
The complexity has been shifted to the Spring framework, which provides the platform for all applications.
3. Simple Client and Simple Service
 - The client bean only needs to know the interface of the service.
 - The client bean needs to have no knowledge about how to build the service it uses.
 - The client bean simply declares what it means, it does not call the injector to get the service implementation.
 - Instead, the injector that is the IoC container calls the client bean and specifies the choice of service it should use.

Spring Module



1. Core Container
 - a. Spring Core and Spring Beans for IoC and dependency injection
 - b. Application Context eliminates singletons and decouple components
 - c. Spring Context for access to object in JNDI (Java Naming and Directory Interface) registry style
 - d. Spring Expression for working with object graph at runtime
2. Data Access / Integration
 - a. Spring JDBC abstract away vendor-specific error codes and handling with relational databases
 - b. Spring ORM for working JPA, Hibernate and other ORM APIs
ORMs helps translate entities and relationships expressed using high level object oriented programming language.
 - c. Spring JMS, Spring messaging for message processing
 - d. Spring TX for working with POJOs declaratively
3. Messaging
 - a. Spring Messaging Module
 - b. Message, Message Channel and Message Handler abstraction
 - c. Annotations for mapping messages to methods
 - d. Similar to Spring MVC annotation-based programming
4. Web
 - a. Spring-Web, Spring WebMVC and Spring-Websocket modules
 - b. Spring web for basic feature, eg. Servlets listeners, HTTP client
 - c. Spring WebMVC for web application programming using MVC paradigm
 - d. Spring-Websocket as thin, lightweight layer above TCP

5. AOP and Instrumentation

Aspect Oriented Programming :

- Programming paradigm that adds new “aspects” to behavior of existing code using “pointcuts” (external specification)
- This Ensure existing code is not modified to add new behavior

AOP in Spring

- a. AOP Alliance-complaint aspect-oriented programming
- b. Define method interceptors, pointcuts and source-level metadata
- c. Implement aspect with @Aspect annotation
- d. Spring AOP module helps combine OOP and AOP

6. Test

- a. Unit-Testing as well as integration Testing
- b. Junit or TestNG
- c. Loading and caching of Application Context objects
- d. Mock Objects to test code in isolation

Dependency Management

Dependency management is fundamentally different from dependency injection.

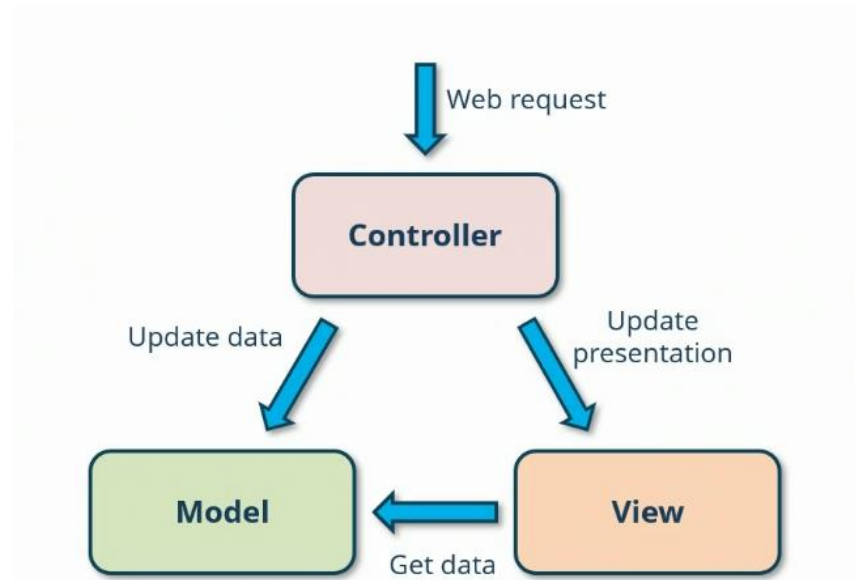
Dependency management refers to how you specify what libraries and jars, Java archives your application depends on.

Dependency management can be summarize as the process of correctly getting all required jar files into the correct locations.

- Process of correctly getting all required jar files into correct location (and into the classpath) so that Spring works correctly
- Extremely important, and somewhat tricky to get right
- Need to get all jar files into right locations
- Compile Time as well as run time
- Distinct from dependency injection
- Deals with physical resources (i.e. files)
- Direct vs. transitive dependencies
- Transitive dependencies are hardest to manage
- Need copy of all jar libraries for Spring
- Separated into modules, use only what is needed
- Spring publishes artifacts to Maven Central
- Also publishes to specific public Maven repo
- Either use Maven, Gradle or Ivy
- Or, Manually download Jars

Model View Controller

Spring MVC / Spring WebMVC : The framework, within Spring, that supports the use of the MVC (Model View Controller) paradigm for building web apps



MVC Flows :

- The controller in a Spring MVC application is responsible for handling the web requests that come in from users.
- The incoming web request might be to access the data available in order to update that data.
- Any update to the data in the underlying database is achieved by updating the model.
- It's also possible that the incoming web request involves updating the view. That is how exactly the existing data is rendered and represented to the user. The controller manages this interaction as well.
- it's quite possible that rendering a new view involves accessing the underlying database to get new data. All data access goes through the model. So the view can request the model to give it the update that it needs.

Model component in the MVC paradigm:

- The model is responsible for **encapsulating application state** and the data that the application represents. (but not application logic and No business logic is present in the model)
- Can be queried to obtain state
The controller and the view both need to be able to access this application state. The model exposes methods, which can be used to **query and obtain the application state**.
- Notified by controller when state needs to change
User interactions with your application requires an update in the state of the application. The model will perform this update when it's notified by the controller.
The controller is responsible for notifying the model when the state needs to change.
- Notifies controller once state has been changed
Once the model has completed updating the application's state, it will then notify the controller.

The state has been changed, and the controller will pick up from there.
Notice that the model and the controller have separate responsibilities.
And they interact with each other in a very well-defined manner.

Controller component in the MVC paradigm:

- The controller **defines the application logic**, but it holds no application state.
The controller defines the business logic of the operations that you want performed on your underlying data.
- The controller responsibility to **map user actions to state changes**.
Think of the controller as the bridge between the view presented to the user, and the underlying data. When the user interacts with your application and performs some kind of update to the underlying data. The mapping of the user's action to state changes is done by the controller.
- The controller **doesn't update the state** directly. It updates the **application state only via the model**.
- The controller is responsible for updating the corresponding views
Update views once application state has changed
The controller manages the interaction between model changes and view rendering.

View component in the MVC paradigm :

- The view is responsible for **presenting the application state** to user via appropriate interface (graphical user interface).
views are implemented depends on the kind of application, whether it's a web application, or a mobile application. The view is responsible for presenting the data present in your application to the user in a nice, user-friendly format.
- The view allows users to **interact with state and modify the state**.
- The view **does not store application state directly** (except in a few cases where you might want the view to cache application state to speed up performance).
The view interacts with the model in order to get the latest application state to render it for users.
- Views in the MVC paradigm are typically **built using reusable and configurable elements**.

MVC Paradigm

- The model, view, and controller interact with one another, are linked using the **observer design pattern**.

The observer design pattern involves setting up of listeners to be notified of changes.

- Synchronous methods vs Asynchronous events.
These notifications often allow individual components to handle these changes in an asynchronous manner. Each component performs its own update. And once the update is complete, it notifies all of its listeners of the changes that it has made.
- Decouple logic and UI
The use of these three components, data access using the model, business logic using the controller, and presentation using the view allows application developers to decouple logic and user interface.
- The Model-View-Controller paradigm is an extremely powerful and popular approach for building all kinds of user interface-based applications.

Spring MVC Technologies

Spring MVC / Spring WebMVC : The framework, within Spring, that supports the use of the MVC (Model View Controller) paradigm for building web apps

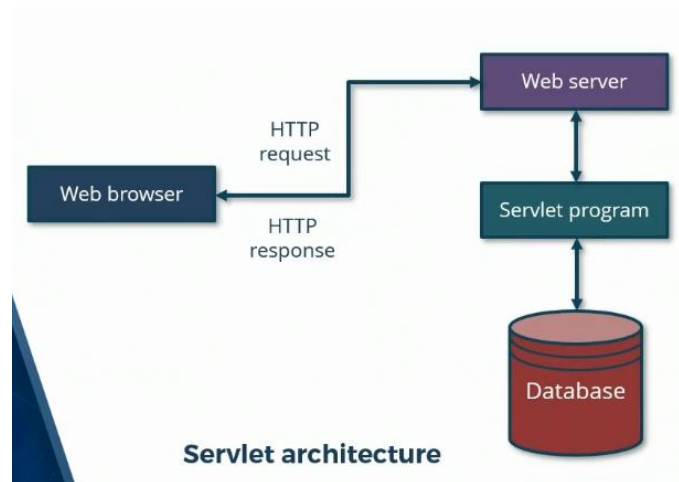
Spring MVC heavily relies on three underlying technology:

- Servlets
Java Code that runs on web server or application server
- JSP
A technology that very similar to PHP, used to create dynamic web pages
JSP pages are responsible for rendering the view but they interact directly with the Java code that you write on the server in order to generate the page dynamically.
- JSTL (Java Server Pages Standard Tag Library)
In order to be able to render dynamic pages, JSP pages use JSTL. JSTL is a collection of useful JSP tags for common tasks such as controlling the flow of how your user interface is rendered.

Servlets:

- Used to build dynamic web pages in the Java programming language.
- Existed since 1996, when most web pages were static
- Run on Java enabled web or app server
- Handle incoming HTTP requests
- Servlets are a server side technology. The actual code for the servlet is on the server. But servlets respond to requests that come from clients.
- Servlets handle incoming HTTP requests, process that HTTP request and render a dynamically generated view in response to that request.
- Servlets thus set up a request response programming model on your application server.

Here is a big picture visualization of how the servlet architecture enhances your web server application.



- Users interact with the interface that you've set up on the web browser, and their actions generate HTTP requests that are sent to your web server.
- Within your web server, you will have a servlet program constantly running and monitoring the HTTP requests that come in.
- This servlet program will know how process that incoming request. And processing that request might involve accessing an underlying database through several layers, of course. Once the response has been generated, the servlet program will then render a dynamic view that is sent back in the form of an HTTP response to the web browser.

Servlets and Servlets Engine

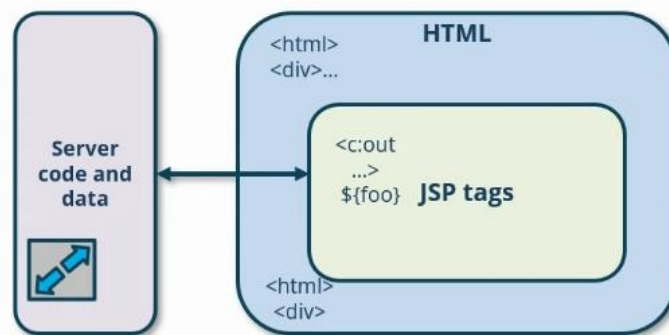
- Servlet (Java Program) runs within environment of Servlet Engine.
The servlet program that you implement in Java runs under a specific environment of the servlet engine.
- Servlet Engine / Servlet Container, provides many important Service.
The servlet engine is also referred to as a servlet container, and it provides many important services over and above hosting servlets.
- Deals with cookies and MIME Type
The servlet engine has capabilities to deal with cookies that you might use within your application, and different MIME types, that is different kinds of data.
- Support for Session Management and security
The servlet engine also provides support for session management and security within your app.

JSP

- JSP or Java Server Pages are an abstraction that run on top of servlets.
- A JSP scriptlet is a basic unit that is enclosed within tags `<% ... %>`.
A combination of the percent sign and the less than and greater than signs. This JSP scriptlet is used to contain any code fragment that is valid for the scripting language that you have used within a page. The scripting language when you're working in Java will be of course Java.

- JSP Scriptlet will be injected into a servlet at runtime.
The code within a JSP scriptlet is transformed into a Java programming language statement fragment, and it is inserted into the service method of the JSP pages servlet.
- Servlet corresponding to JSP scriptlet will be cached and re-used until JSP is modified.
- JSP compiler is needed to compile JSP scriptlet to servlet to compile JSP scriptlet to servlet.
JSP almost looks like a language onto itself, though it isn't really, but you need a JSP compiler to compile the JSP scriptlet and its code fragments to the corresponding servlet.
- This JSP compiler runs on your Java enabled web or application server.
- The servlet code runs inside the Java Virtual Machine on your web server.
- JSP allows Java code and HTML to be interleaved
JSP code if you look at it seems like a strange combination of HTML as well as Java. You will use both HTML tags for presentation as well as programming language constructs within your JSP page.

High level visualization of how JSP works with your Java code on the server.



When you look at a JSP page, you will find a number of HTML tags, which define the presentation or the rendering of the view. Within this HTML, you will find JSP specific tags as well.

These are JSTL tags, which control the flow of dynamic rendering. The JSP compiler will compile this page and inject it so that it's part of the server code and data.

JSTL, JSP Standard Tag Library.

- Scriptlet tags are the basic building blocks of Java Server Pages.
- JSTL is simply a standard library comprising of JSP tags that can be used to dynamically control how your page is rendered.
- Taglibs contain core functionality of JSTL
The core functionality of JSTL is contained within the Taglibs library. And this is a library to which we'll add a dependency, when we build our Spring MVC app.

- Taglibs ship with every servlet and JSP framework.
Taglibs stands for tag libraries that define reusable JSP functionality.

JSTL Tag Classifications

1. JSTL Core tags: loops, control flow, <div> output.
2. JSTL Formatting tags: for formatting how your output is rendered specifically for dates and internationalization.
3. JSTL SQL tags: to access data from a SQL database directly (these tags are usually discouraged because of security implications.)
4. JSTL XML tags: to allow you to work with XML documents.
5. JSTL functions: mainly have to do with string manipulation.

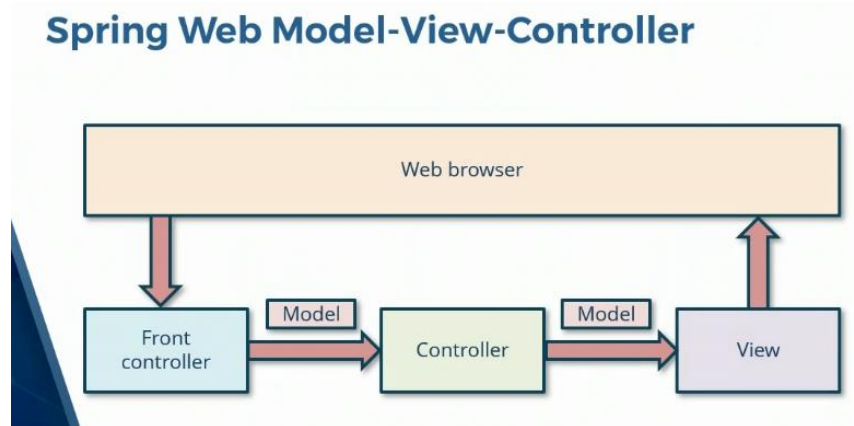
Spring MVC Architecture

- **MVC architecture for web apps**
Spring MVC framework offers the **Model-View-Controller architecture build web applications**. The Model-View-Controller paradigm sets up a reusable, robust pattern that you can use to design user interface-based apps.
- **DispatcherServlet as from controller to receive browser request**
Every Spring MVC application uses a special servlet known as the **DispatcherServlet as the front controller**. Front controller is the controller that **receives all of the incoming browser requests from clients**. The DispatcherServlet is an implementation that is available within your Spring MVC framework.
- **Dispatched to appropriate controller via handler mappings**
After receiving the incoming web request from the browser, the same request is then dispatched to the correct controller which knows how to handle that request. The correct controller is configured using handler mappings. The DispatcherServlet will use the handler mappings that you have specified to figure out which is the right controller to handle this request.
- **Result returned to users via ViewResolver classes**
And once the controller has finished handling the request, the result is returned to the user via ViewResolver classes. The ViewResolver classes map logical view names to physical view implementations. So that your controller code doesn't have to deal with physical views.

Typical Spring MVC App

- **Model – POJO (Plain Old Java Object)**
Within your Spring MVC application, the model that holds application state is typically specified using POJOs, or plain old Java objects. You'll set up Java classes with getters and setters, and that'll be could be your model.
- **Views – JSP templates written with JSTL**
Views in your Spring MVC application are typically implemented using JSP templates that are written using JSTL tag libraries to control the flow of the dynamic rendering of JSP.
- **Controller – Dispatcher Servlet**
The controller within your Spring MVC application is the Dispatcher Servlet. The Dispatcher Servlet is the front controller. Meaning, it's the controller that receives all of the incoming web requests. The Dispatcher Servlet then maps every request to the right controller, which knows how to handle that request.
- **Userfull for classic 3-tier architecture**
This basic setup of the Spring MVC application allows us to set up the classic 3-tier architecture. The data access layer, the business logic layer, and the presentation layer.

The architecture diagram representing the Model-View-Controller paradigm as implemented in Spring MVC.

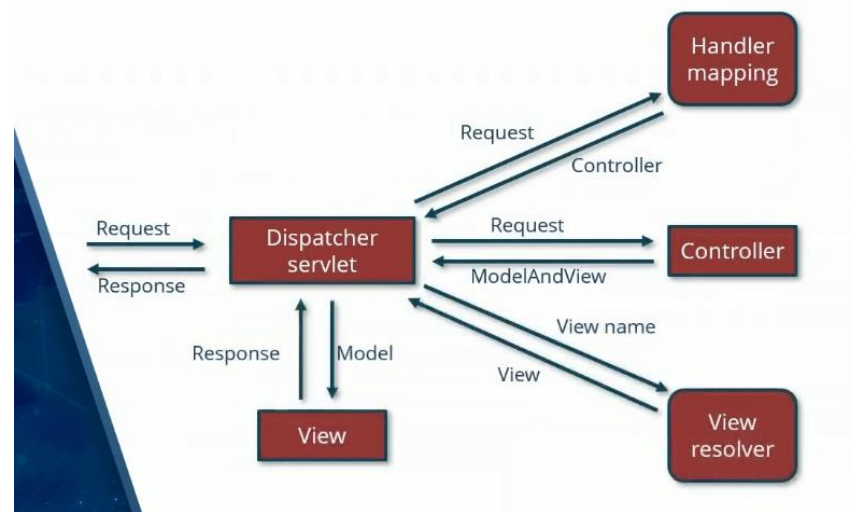


- Views are rendered within the web browser. The user interacts with the web browser view. And then this results in request made to the front controller, which is the Dispatcher Servlet.
- Based on the path for the user request, the front controller will look up the handler mappings that we have configured to figure out which controller is the right one to handle this request.
- The model object associated with the application state will be available to the controller. The controller can use the model to access what information it needs. It can also use the model to specify any updates to the application state. This model is then passed back to the view, which then re-renders itself on the web browser.

Role of Dispatcher Servlet

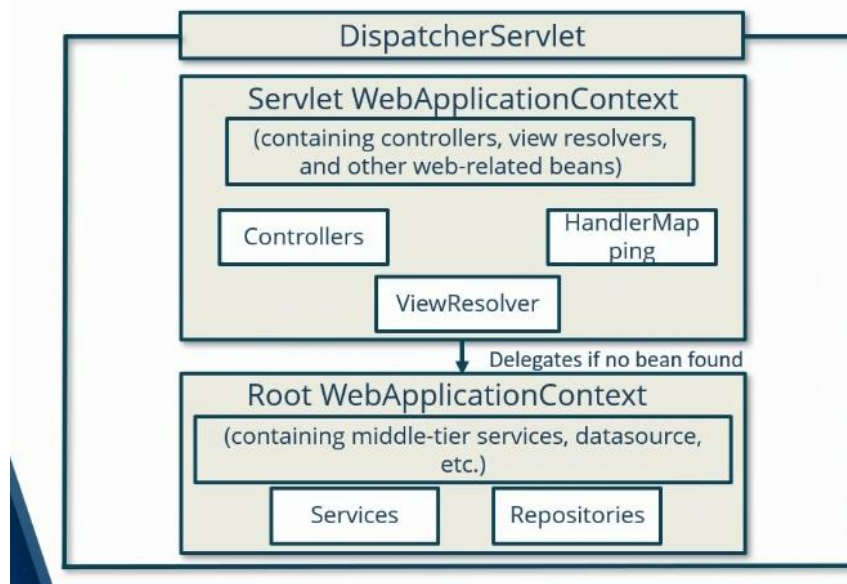
- Dispatcher Servlet a.k.a Spring Controller is a front controller.
The Dispatcher Servlet is part of all Spring MVC applications. And it acts as the front controller.
- Every web request first comes to this controller
It's called the front controller because every web request first comes to the Dispatcher Servlet controller. A default implementation of the Dispatcher Servlet comes as a part of Spring MVC.
- Request handler controller, view resolvers, annotations then play a role.
Once the Dispatcher Servlet receives a request, other components come into play. Such as the request handler controllers, the one which knows how to handle a particular request. View resolvers, which determine which view should be rendered as a part of the response. Annotations for other objects, and so on.
- Request is then passed onto actual handler
The Dispatcher Servlet is responsible for passing the processing on to the right controller. Once it's figured out what the right controller is, the request is then passed on to the actual handler.

The Flow of a request using a high level component diagram.

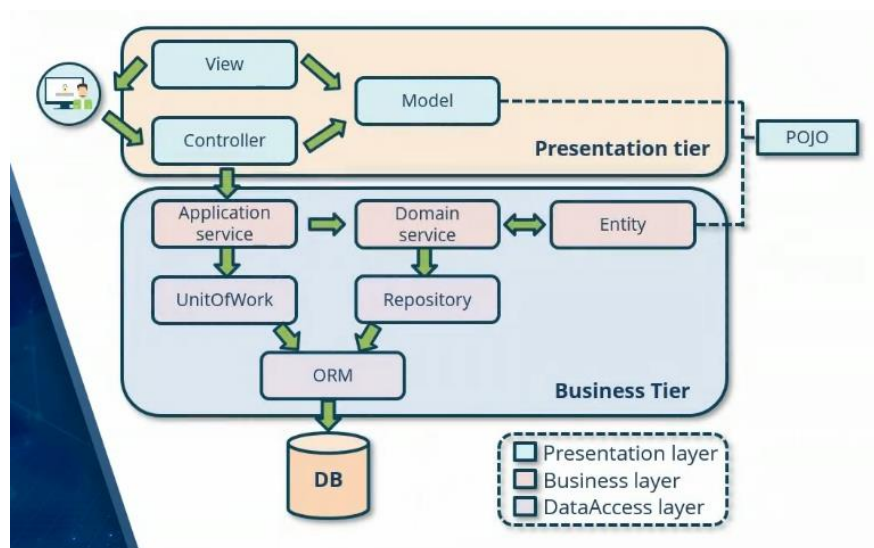


- Starting with the Dispatcher Servlet, notice that the Dispatcher Servlet is responsible for handling every incoming request. And it also sends the response down to the client, that is the web browser.
- The Dispatcher Servlet may not know how to process a particular request. What it does know is which controller knows how to process this request. It knows this via the handler mappings that we configure. Based on the request path, the Dispatcher Servlet will look up handler mappings that have been configured. And find the right controller associated with a particular handler mapping.
- Once the right controller has been found, the Dispatcher Servlet will then pass the request on to that controller. The controller will process the request and this might require updating the underlying model as well. Once the processing is complete, the controller returns a model and view object. The updated application state stored in the model, and the view that needs to be rendered.
- The controller typically works with logical views and not physical view implementations. This additional abstraction allows you to change the physical implementation of a view without changing its logical name. The controller knows the view name. The view resolver then maps that view name to an actual physical view that can be rendered.
- The Dispatcher Servlet then helps render this physical view implementation by passing in the application state via the model. The physical rendering of the view is the response that the Dispatcher Servlet sends back down to the client.

Another way of the structure of a Spring MVC application.



- We have the `DispatcherServlet` that deals with requests and responses. Within your MVC application, the `Servlet WebApplicationContext` contains `Controllers`, `ViewResolvers`, and other web related beans.
- The `Servlet WebApplicationContext` defines the `Controllers` and `HandlerMappings` to controllers.
- We also have a `Root WebApplicationContext` which contains the middle-tier services and data access objects.



- Spring MVC applications thus lend themselves to the classic 3-tier architecture. We have the **Presentation tier**, which is responsible for rendering views to users.
- The **Business tier** contains the business logic of your applications.
- And finally, the portion of your application that deals with the underlying database is the **Data Access tier**.

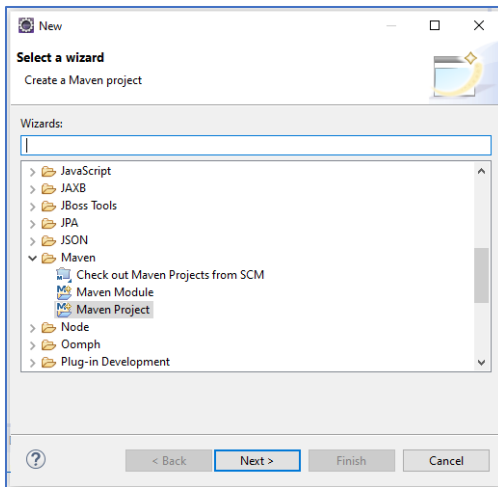
Setup Maven Project on Eclipse

1. Open Eclipse IDE



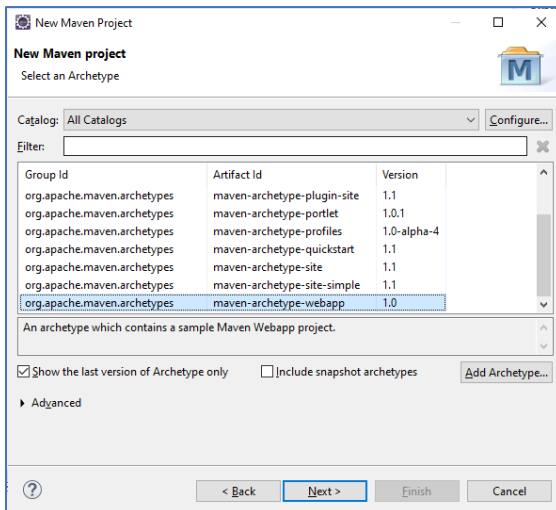
2. Create New Maven Project

File > New > Maven Project



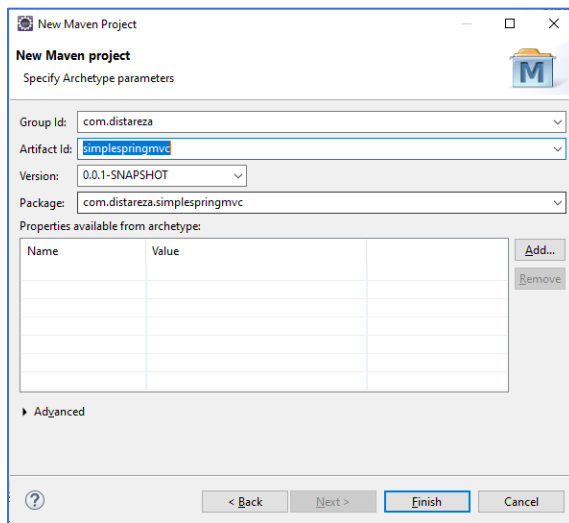
3. select or choose **Workspace** location

4. Choose **maven-archetype-webapp (ver 1.0)** as the archetype, click next



Now, in order to generate a project template for a Maven application, we need to choose an archetype. An archetype in Maven is a project templating toolkit. This will set up all of the boilerplate configuration for the kind of project that you want to build using Maven. Now, the kind of project that we want to set up is a web application. So choose the Maven archetype webapp version 1.0

5. Defined the artifact Id, and click finish



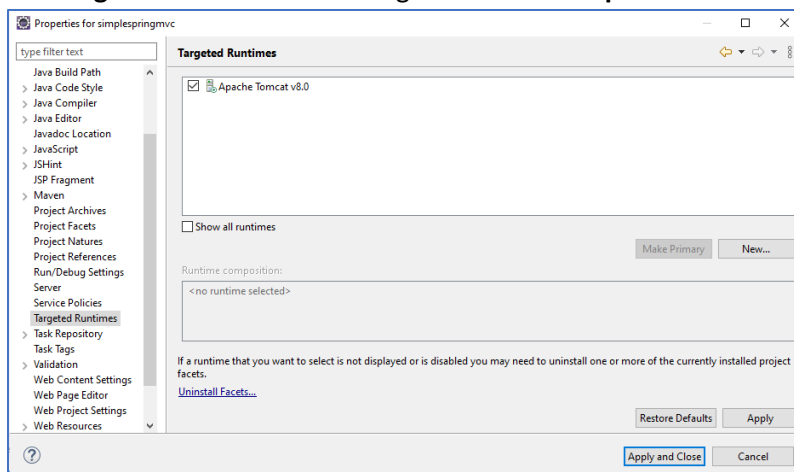
The Group id makes up part of the identifier that is used to uniquely identify this particular project. The Group id is typically the name of the company or the organization that's creating this project.

The Artifact id is the unique name of the project

6. Expand the Project and setup the Target Runtime Server to Tomcat

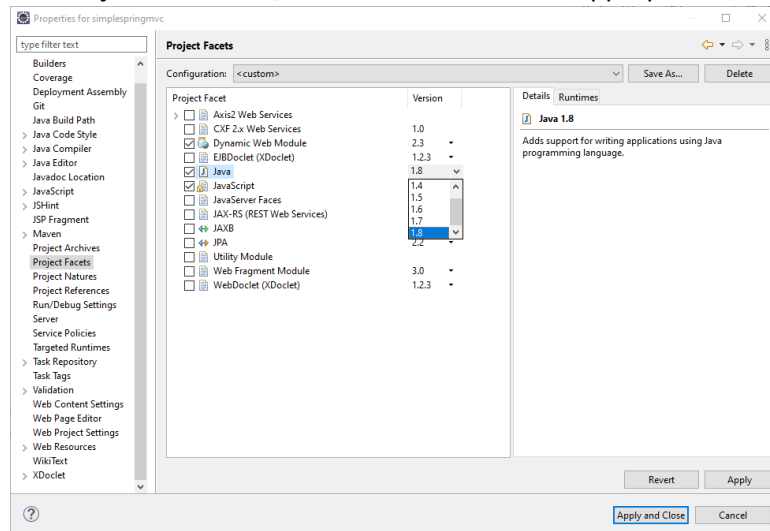
Right click on Project Name and select **Properties**

on **"Target Runtimes"** tab set target runtimes to **Apache Tomcat** and click **Apply**

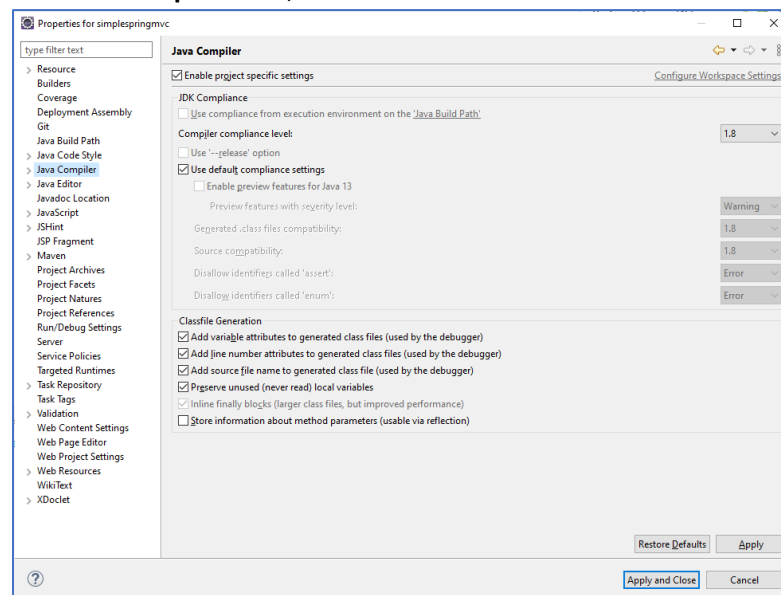


7. Setup Runtime Java Version

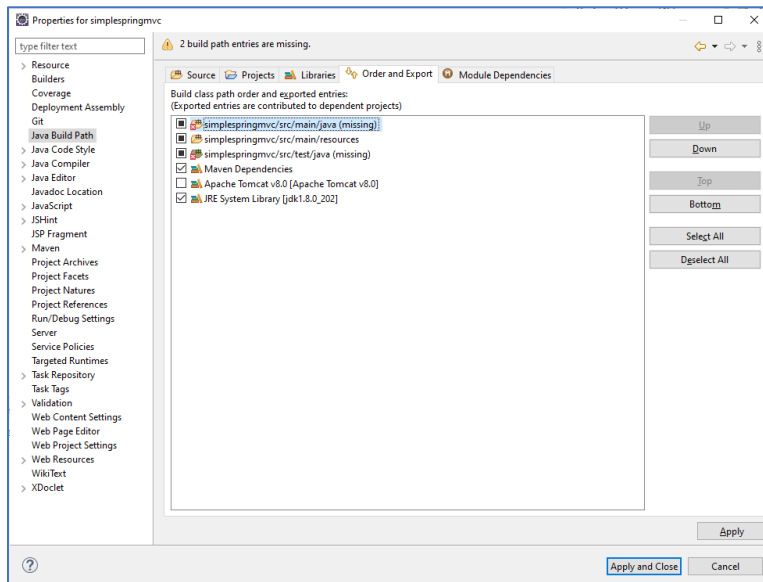
on **“Project Facet”** tab, select **“Java”** and set to appropriate JVM version and click Apply



on **“Java Compiler”** tab, make sure it set to same JVM Version

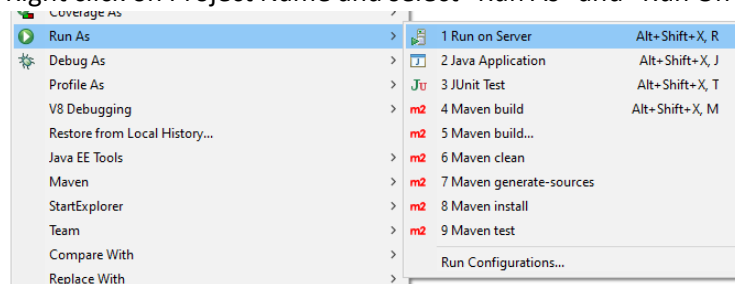


on “**Java Build Path**” tab, make sure that “**Maven Dependencies**” and “**JRE System Library**” is checked, click **Apply and Close**

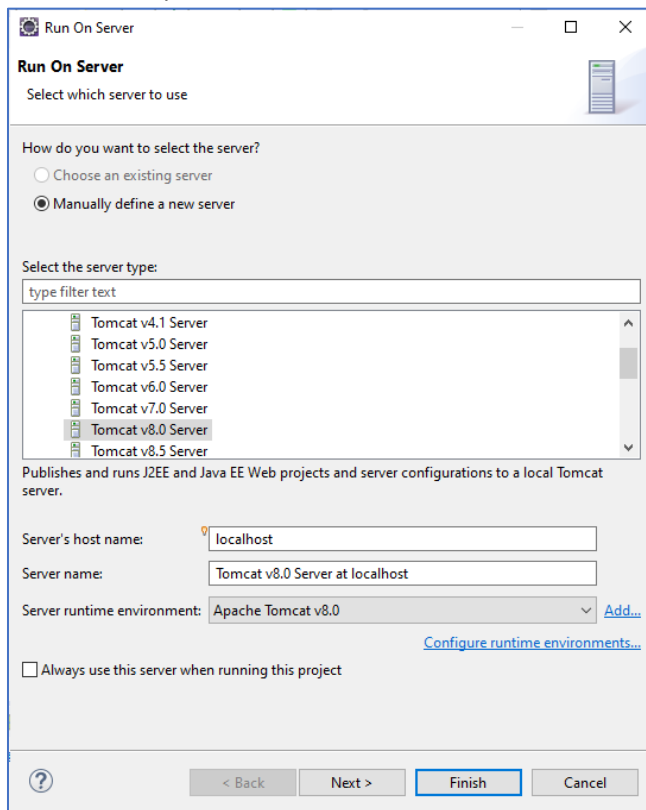


8. Test the default project

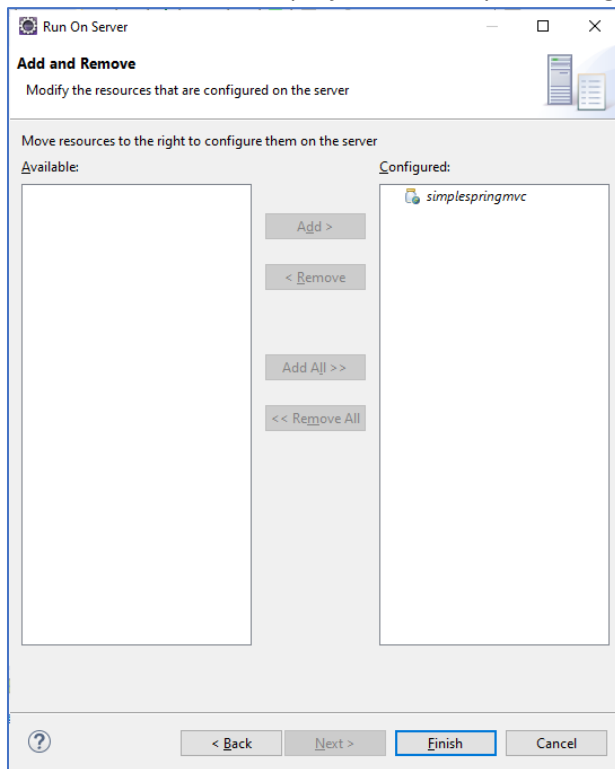
Right click on Project Name and select “Run As” and “Run On Server”



Choose the Apache Tomcat as the server

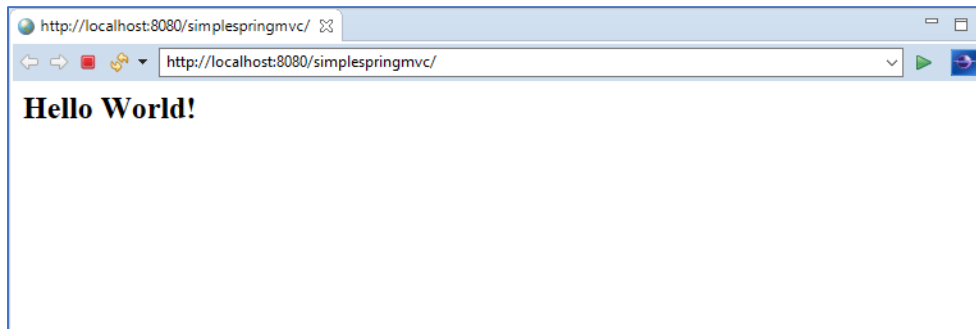


and make sure that our project is already on the right side of “**Configured**” column, click **Finish**



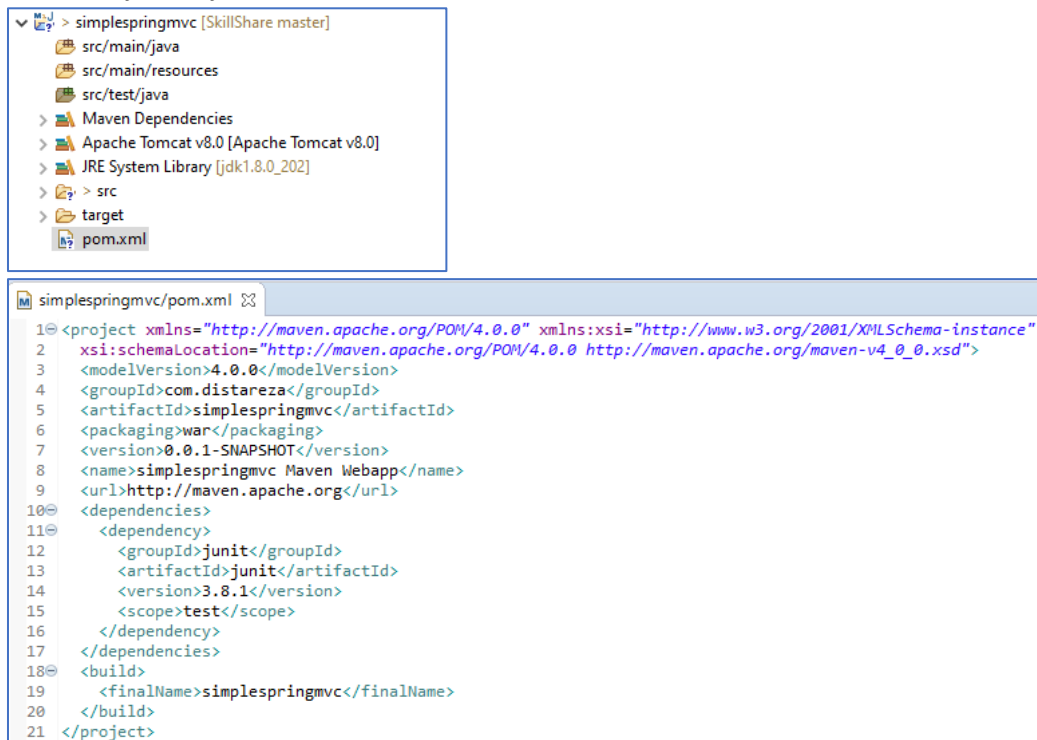
if you wait for a bit, you'll see a bunch of log messages on your console window and you'll see **Hello World!** printed out to screen. Your simple web application has been hosted and is running on your Tomcat Runtime Environment, your Tomcat server. Notice the structure of the URL here, **http://localhost:8080**. That is the port where your application is hosted. This is followed by our project named **simplespringmvc**. This is part of the path. And **Hello World!**, well, Hello World comes from some of the default files that have been set up as a part of your web app.

The Web application should be run inside Eclipse IDE as follows



Basic structure of this Maven project

1. POM : Project Object Model



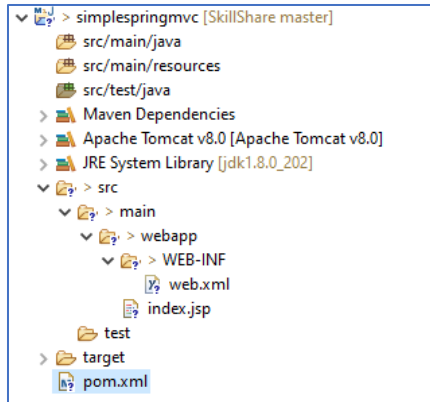
This is a fundamental unit of work in Maven. It's an XML file that contains information about your project and configuration details used by Maven to build the project.

- On lines 4 and 5, you can see the groupId and artifactId of our project. This is what we had configured when we set up the project.
- On line 6, notice the packaging war. This means building this web application will produce war files or web archive files.
- On line 10, you can see the dependencies tag. Within this tag is where we'll specify other jars, or artifacts as they're called in Maven, that our project depends on. The only dependency that has been added by Maven by default is a dependency on the junit library version 3.8.1. This is what we can use to write unit test for our application.

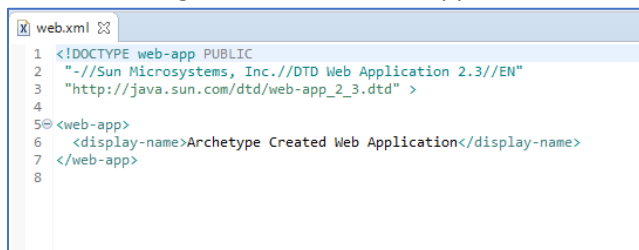
2. The Source directory

Under source main and webapp, you will find an WEB-INF subfolder. This WEB-INF folder is the document root of your web application. This is where you will place your static files such as HTML files, JSP files.

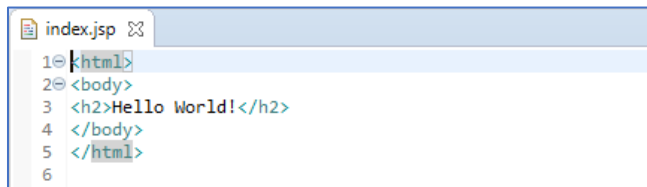
Getting Started with Spring Framework



The only configuration file that has been set up by default is the web.xml file. web.xml here is the basic configuration file for web applications.




Index.jsp, when you hit the URL for a web application, this is the file that will be executed by default. It's a simple HTML file at this point in time. It doesn't contain any jstl or jsp specific annotations. All it contains is the **Hello World!**, which we viewed when we ran our web app.



Adding Spring MVC Dependencies in Maven Project

3. Add following line in POM.xml file



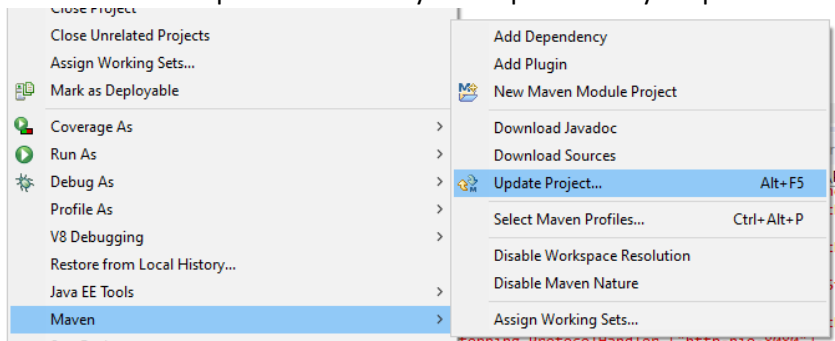
```
1<?xml version="1.0" encoding="UTF-8"?>
2<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
4  <modelVersion>4.0.0</modelVersion>
5  <groupId>com.distareza</groupId>
6  <artifactId>simplespringmvc</artifactId>
7  <packaging>war</packaging>
8  <version>0.0.1-SNAPSHOT</version>
9  <name>simplespringmvc Maven Webapp</name>
10 <url>http://maven.apache.org</url>
11 <dependencies>
12   <dependency>
13     <groupId>junit</groupId>
14     <artifactId>junit</artifactId>
15     <version>3.8.1</version>
16     <scope>test</scope>
17   </dependency>
18   <dependency>
19     <groupId>org.springframework</groupId>
20     <artifactId>spring-webmvc</artifactId>
21     <version>5.1.8.RELEASE</version>
22   </dependency>
23   <dependency>
24     <groupId>javax.servlet</groupId>
25     <artifactId>javax.servlet-api</artifactId>
26     <version>3.0.1</version>
27     <scope>provided</scope>
28   </dependency>
29   <dependency>
30     <groupId>javax.servlet</groupId>
31     <artifactId>jstl</artifactId>
32     <version>1.2</version>
33   </dependency>
34 </dependencies>
35 <build>
36   <finalName>simplespringmvc</finalName>
37 </build>
38 </project>
```

Within the dependencies tag here, add a few more dependency tags.

On line 20. our project will depend on the artifactId **spring-webmvc**. using the version 5.1.8.RELEASE.

And On line 26, 33 Our application depends on the javax.servlet-api and also depend on jstl.

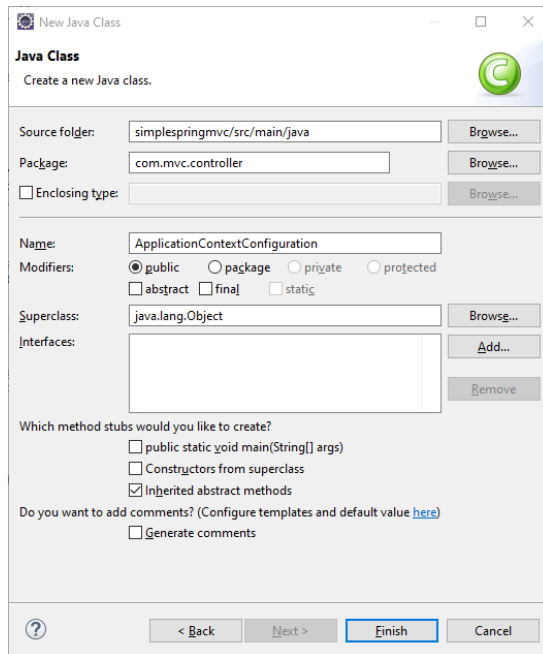
Once you update the POM.xml , please update the project by do **Right Click** on project name and select **Maven > Update Project** to update or download jar dependencies into your local system and your project class path maven lib dependencies. This should allow Eclipse to find an index all of the dependencies that you've specified in your pom.xml file.



Specifying Application Context Programmatically

4. Create Application Context Programmatically

Under src/main/java source directory create Class called “ApplicationContextConfiguration.java”
This is the class that we'll use to programmatically define the application context of our Spring application.



The ApplicationContext is configuration that is associated with all Spring applications including Spring MVC applications. This is the file that holds the Bean definitions for our Spring app. These are configuration properties that apply to Spring as a whole and are not specific to Spring MVC. We are now defining this configuration file programmatically rather than using XML.

```
1 package com.mvc.controller;
2
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5 import org.springframework.web.servlet.view.InternalResourceViewResolver;
6
7 @Configuration
8 public class ApplicationContextConfiguration {
9
10     @Bean(name = "viewResolver")
11     public InternalResourceViewResolver getViewResolver() {
12         InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
13         viewResolver.setPrefix("/");
14         viewResolver.setSuffix(".jsp");
15         return viewResolver;
16     }
17 }
18
19
```

Set up the import statements for all of the libraries that you will use within this Java class. Any programmatic config specification in Spring has to be annotated with the **@Configuration** annotation. This is part of the Spring code framework, and this annotation indicates that this class has Bean definition methods. The Spring framework that we're using will look for classes tagged with this annotation and pick up Bean definitions from this class.

The first Bean that I specify here within this application context is a **viewResolver** bean. A

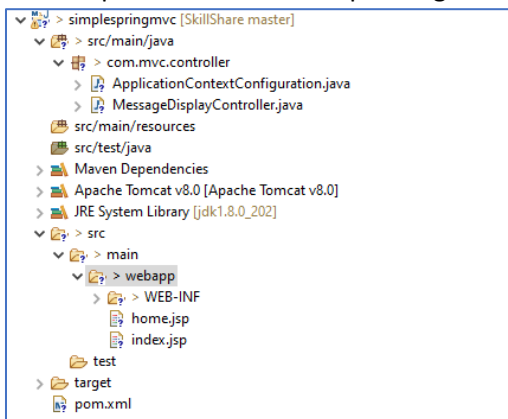
viewResolver in Spring can be thought of as a logical mapping from view names to a specific kind of view. This allows you to render views without being tied to a specific view technology. By always referring to the views within your app using their logical names rather than physical implementation allows you to decouple your app from how views are actually implemented.

Mapping a logical view to an actual one is the role of the viewResolver. The viewResolver implementation that we have specified here is one that is available as a part of Spring MVC. And that is the **internal resource viewResolver**. We instantiate this viewResolver, we set the prefix for all views that's going to be forward / that is our root document folder. The root document folder corresponds to the WEB-INF folder in our web application.

We also set a suffix for all views to be **.jsp** indicating all of our views are jsp pages. When we use the **@Bean** annotation on this method, anytime there is a need for a viewResolver within our application, this internal resource viewResolver is what will be injected into that object.

5. Message Display Controller

There is another class that we need to define first. And this class is the MessageDisplayController. This is going to be the controller object that will handle the incoming web requests from our web application users. Access the model, update state if needed, and then respond with the corresponding view.



Getting Started with Spring Framework

```
MessageDisplayController.java
1 package com.mvc.controller;
2
3 import javax.servlet.http.HttpServletRequest;
4 import javax.servlet.http.HttpServletResponse;
5
6 import org.springframework.web.servlet.ModelAndView;
7 import org.springframework.web.servlet.mvc.Controller;
8
9 public class MessageDisplayController implements Controller {
10
11     public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) throws Exception {
12         return new ModelAndView("home");
13     }
14 }
15
16
home.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1" isELIgnored="false" %>
3 <!DOCTYPE html>
4 <html>
5 <head>
6 <meta charset="ISO-8859-1" content="text/html" http-equiv="Content-Type">
7 <title>Spring MVC Page</title>
8 </head>
9 <body>
10 <h1>Welcome to Spring MVC!</h1>
11 <span>Using Programmatic application context and controller</span>
12 </body>
13 </html>
```

MessageDisplayController is a controller implement the Controller interface. Having the controller implement this interface is an explicit specification. Any class implementing this interface should be able to handle HTTP web requests in a thread safe manner.

The Implementation the methods of this controller interface, **handleRequest** is what is used to handle incoming web requests from users. Notice that the input arguments to this handleRequest method are an HttpServletRequest and an HttpServletResponse.

Our controller is very straightforward here. We don't actually interact with a model, we simply return the view to be rendered using the ModelAndView object.

The ModelAndView object in Spring MVC contains a model map as well as a view and this is what the viewResolver will use to render the view. Notice the name of the view, it's simply **home**.

This ModelAndView object will be received by the dispatcher servlet and the view rendered. Now if you remember the configuration of the viewResolver bean that we had specified in the ApplicationContextConfiguration, we had set that all view prefixes are forward / and suffixes are .jsp. So home here is the logical name for the **home.jsp** file that we had set up earlier.

Now that we have specified an explicit controller for our Spring MVC application, let's head back to our ApplicationContextConfiguration and set up this controller as a Bean (line 20).

Getting Started with Spring Framework

```
ApplicationContextConfiguration.java
1 package com.mvc.controller;
2
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5 import org.springframework.web.servlet.mvc.Controller;
6 import org.springframework.web.servlet.view.InternalResourceViewResolver;
7
8 @Configuration
9 public class ApplicationContextConfiguration {
10
11     @Bean(name = "viewResolver")
12     public InternalResourceViewResolver getViewResolver() {
13         InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
14         viewResolver.setPrefix("/");
15         viewResolver.setSuffix(".jsp");
16         return viewResolver;
17     }
18
19
20     @Bean(name = "/")
21     public Controller getMessageDisplayController() {
22         return new MessageDisplayController();
23     }
24 }
25
26 }
```

I have a method here called `getMessageDisplayController` which returns an object of the controller interface. I instantiate a new `MessageDisplayController` and return that. This is tagged using the **@Bean** annotation and the name of the bean is simply forward `/`.

The forward `/` here corresponds to the root of our web application. This is the handler mapping. So any web requests made to the root of the web application will be given this controller to handle that request.

6. WebServletConfiguration

The next step is to set up the `WebServletConfiguration`, which once again, we'll specify programmatically.

```
WebServletConfiguration.java
1 package com.mvc.controller;
2
3 import javax.servlet.ServletContext;
4 import javax.servlet.ServletException;
5 import javax.servlet.ServletRegistration;
6
7 import org.springframework.web.WebApplicationInitializer;
8 import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
9 import org.springframework.web.servlet.DispatcherServlet;
10
11 public class WebServletConfiguration implements WebApplicationInitializer {
12
13     public void onStartUp(ServletContext servletContext) throws ServletException {
14
15         AnnotationConfigWebApplicationContext appContext = new AnnotationConfigWebApplicationContext();
16         appContext.register(ApplicationContextConfiguration.class);
17
18         ServletRegistration.Dynamic servlet = servletContext.addServlet("dispatcher", new DispatcherServlet(appContext));
19         servlet.setLoadOnStartup(1);
20         servlet.addMapping("/");
21     }
22 }
23
24 }
```

The web servlet configuration are configuration properties that correspond to your Spring MVC application. These are not Spring specific, these are Spring MVC specific. This is the class that we'll use to programmatically configure the Spring MVC servlets.

In order to do so, we need to have this configuration implement the **WebApplicationInitializer** interface. This interface is what *allows us to configure our servlet context programmatically*. Any implementation of this **WebApplicationInitializer** interface is automatically detected by the Spring servlet container. And this will be included as a part of the bootstrap process for that servlet container.

Now let's programmatically specify our web servlet configuration. Let's implement the method here **onStartup** which is a part of the **WebApplicationInitializer** interface. **onStartup** takes in the **ServletContext** that will be injected into it by **Spring**. On the startup of our application, we'll register the **ApplicationContext** corresponding to this application.

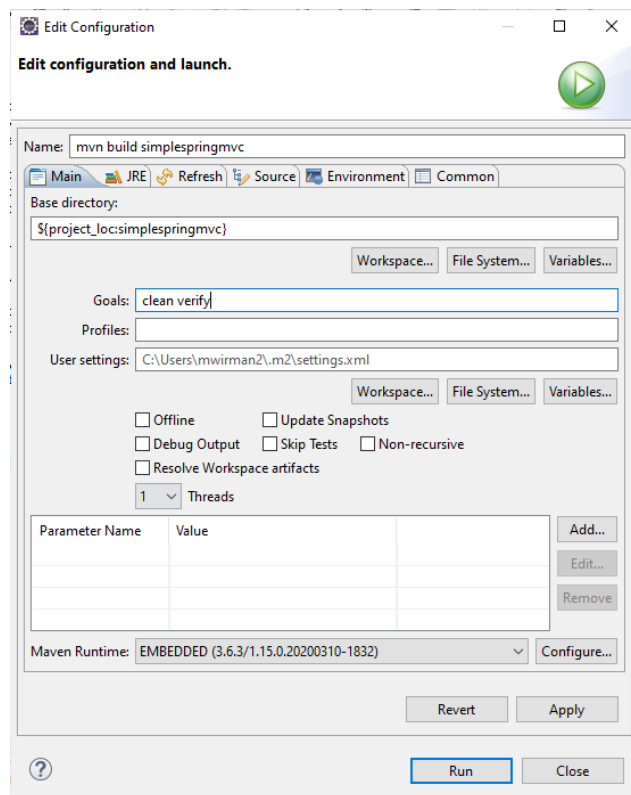
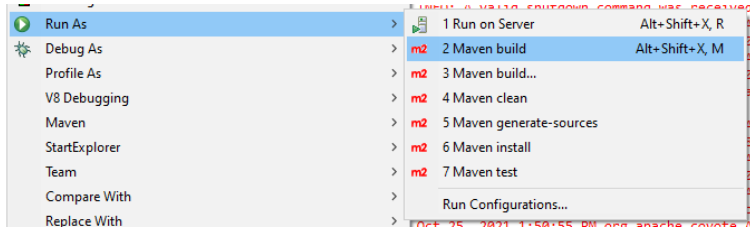
Remember, we have programmatically specified that as well in **ApplicationContextConfiguration**. The registration of the application context is the code that you see on lines 15 and 16. In our Spring MVC application, we need to explicitly register the **DispatcherServlet** as well. So instantiate the **DispatcherServlet**, this you can see on lines 18, I've called servlet **context.addServlet**. I've called this the dispatcher servlet, that's the name.

And I've instantiated a new **DispatcherServlet** that takes in the application context. We set this servlet to load on startup, **setLoadOnStartup** as 1. And we call **servlet.addMapping** and add a mapping for the root document of our web application which is just forward **/**.

Running a Spring MVC Application Within Eclipse

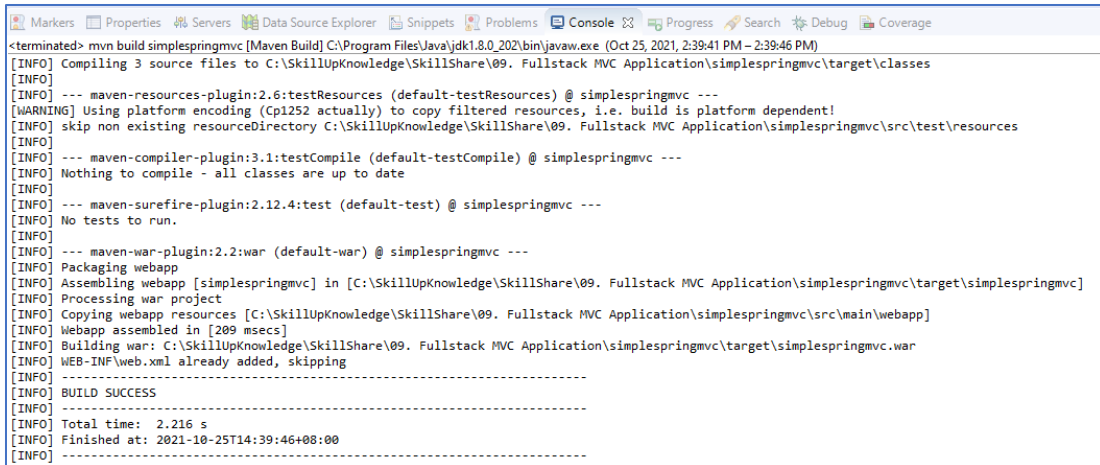
7. Run the code

Now we'll need to build this first. So I'm going to right click and Run As and choose the Maven build option. Within the Goals of this dialog, specify **clean verify**.



So you're always doing a clean Maven build. Go ahead, hit Apply, and then Run. And this will set up a new build configuration.

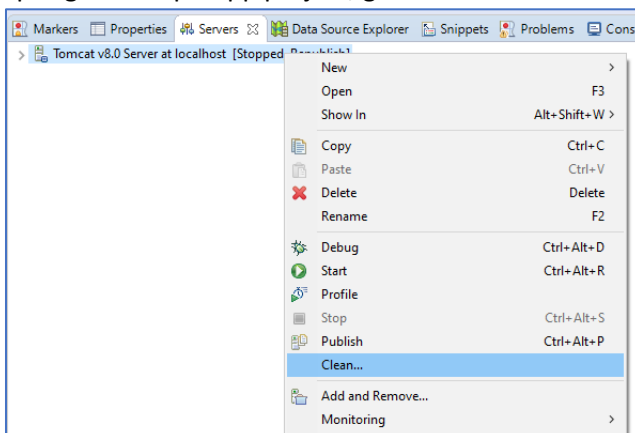
Getting Started with Spring Framework



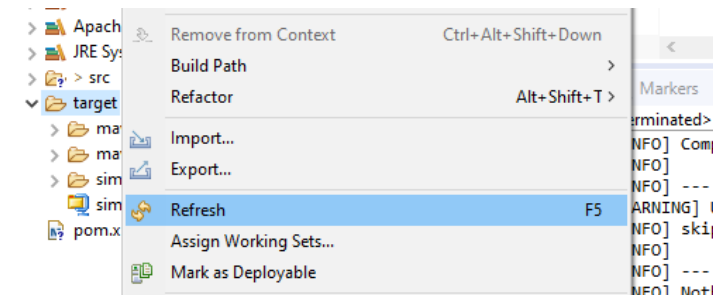
```
<terminated> mvn build simplespringmvc [Maven Build] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (Oct 25, 2021, 2:39:41 PM - 2:39:46 PM)
[INFO] Compiling 3 source files to C:\SkillUpKnowledge\SkillShare\09. Fullstack MVC Application\simplespringmvc\target\classes
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ simplespringmvc ---
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory C:\SkillUpKnowledge\SkillShare\09. Fullstack MVC Application\simplespringmvc\src\test\resources
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ simplespringmvc ---
[INFO] Nothing to compile - all classes are up to date
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ simplespringmvc ---
[INFO] No tests to run.
[INFO] --- maven-war-plugin:2.2:war (default-war) @ simplespringmvc ---
[INFO] Packaging webapp
[INFO] Assembling webapp [simplespringmvc] in [C:\SkillUpKnowledge\SkillShare\09. Fullstack MVC Application\simplespringmvc\target\simplespringmvc]
[INFO] Processing war project
[INFO] Copying webapp resources [C:\SkillUpKnowledge\SkillShare\09. Fullstack MVC Application\simplespringmvc\src\main\webapp]
[INFO] Webapp assembled in [209 msecs]
[INFO] Building war: C:\SkillUpKnowledge\SkillShare\09. Fullstack MVC Application\simplespringmvc\target\simplespringmvc.war
[INFO] WEB-INF\web.xml already added, skipping
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 2.216 s
[INFO] Finished at: 2021-10-25T14:39:46+08:00
[INFO] -----
```

8. Stop and Clean the Tomcat Cache

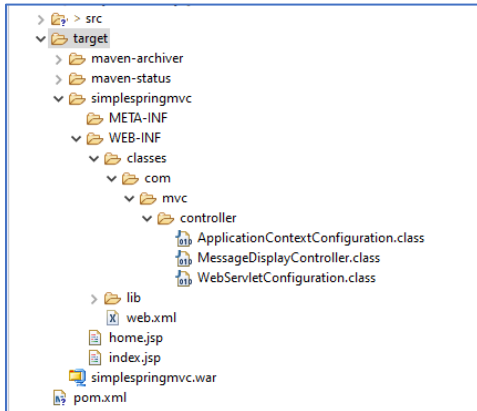
To run this application within Eclipse and ensure that Eclipse doesn't cache a lot of stuff, you need to perform a bunch of steps. Head over to the Servers tab and make sure that your Tomcat server is stopped. Stop it, if it isn't, right click on it and Clean. Make sure you get rid of all of the cached contents within your Tomcat server. We can now build our project, right click on your SpringMvcSimpleApp project, go to Run As and choose Maven build, once again.



Your Maven dependencies have been updated, your build should be a success. A quick way to check whether an up to date binary has been created is to select the target option here. Right click and choose Refresh.



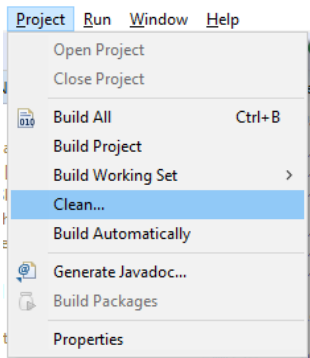
Now, within this target folder you'll be able to expand this folder and see that a new SpringMVCSimpleApp.WAR file has been generated. You should be able to dive into the folders for the Spring MVC app. And within the controller package, you should be able to find the dot class definitions corresponding to every java file that you have within your Java app.



Also to ensure that Eclipse hasn't cached your previously generated WAR file, you can head over to the Project menu here and choose the Clean option so that all Eclipse related caches will be cleaned. Choose Clean for your SpringMvcSimpleApp project.

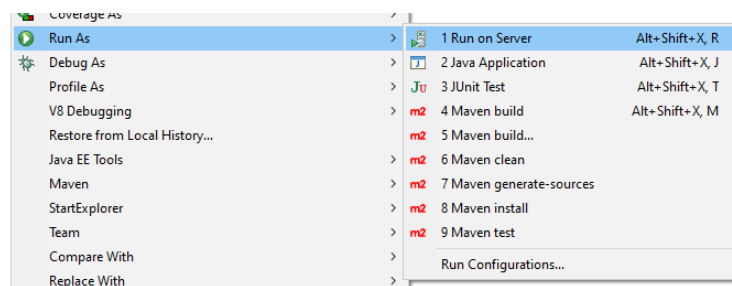
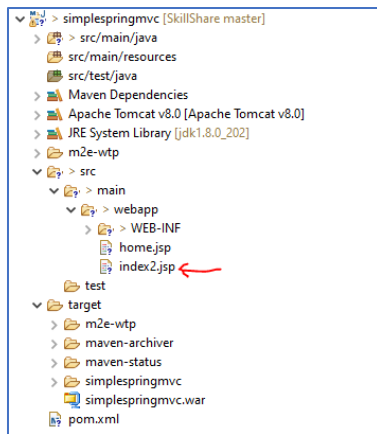
9. Run the app

You're now ready to run your code. Right click on your project, choose Run As and then choose Run on Server. Wait for a little bit and notice that your web browser has opened up and here is your first Spring MVC application running on your Apache Tomcat server.

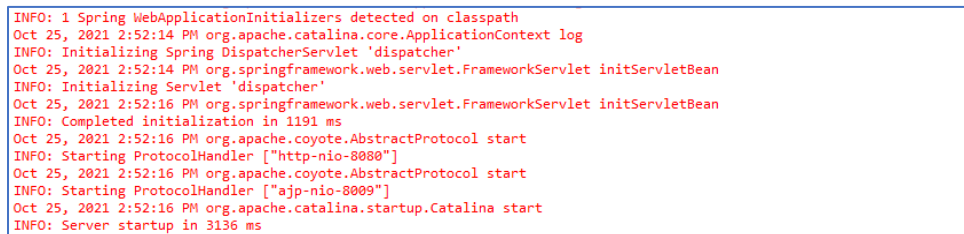


remove index.jsp or rename it to index2.js

Getting Started with Spring Framework



Notice we are still running on localhost 8080. Notice the path, simplespringmvc/. We'd configured the root of our application to be mapped to home.jsp. And the contents of home.jsp is what is rendered here on screen in front of you. Back to our Eclipse IDE and let's take a quick look at some logs on our Console window.



Catalina is the Servlet container within the Tomcat server which has been started up as you can see from the logs here. And if you scroll down, you will find that our dispatcher Servlet has been registered. There is an info log there at the top of your screen initializing Servlet dispatcher.

Now the elaborate set of steps that I followed here to run the Spring MVC application from within Eclipse on Apache Tomcat is not strictly required. However, Eclipse does a lot of caching. So often your WAR files might not be updated within Eclipse. It's best to follow all of the steps to ensure you're getting the latest version of your app.

Specifying Application Context Using XML

1. Explicitly Specified the source and target JVM Version in POM.xml

Make sure that Eclipse are running same JVM Version in your local environment by explicitly specifying the source and target versions of your **Java SDK. 1.8** here within the specification

```
38< build>
39  <finalName>simplespringmvc</finalName>
40  <sourceDirectory>src/main/java</sourceDirectory>
41< /plugins>
42< /plugin>
43  <artifactId>maven-compiler-plugin</artifactId>
44  <version>3.5.1</version>
45< /configuration>
46  <source>1.8</source>
47  <target>1.8</target>
48< /configuration>
49< /plugin>
50< /plugins>
51< /build>
```

2. Create ApplicationContext Xml File under **WEB-INF** directory

```
application-context.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2< beans xmlns="http://www.springframework.org/schema/beans"
3  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4  xmlns:context="http://www.springframework.org/schema/context"
5  xsi:schemaLocation="http://www.springframework.org/schema/beans
6    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7    http://www.springframework.org/schema/context
8    http://www.springframework.org/schema/context/spring-context-3.0.xsd">
9
10  <bean name="/" class="com.mvc.controller.MessageDisplayController" />
11< /bean>
12  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
13    <property name="prefix" value="/" />
14    <property name="suffix" value=".jsp"/>
15  </bean>
16< /beans>
```

This application-context.xml file is where we'll specify the beans that Spring will manage.

The first bean specification here is for our **MessageDisplayController** object with the name of the bean is simply forward **/**.

This is the only controller that handles the root mapping of our web application, this will simply display the home page.

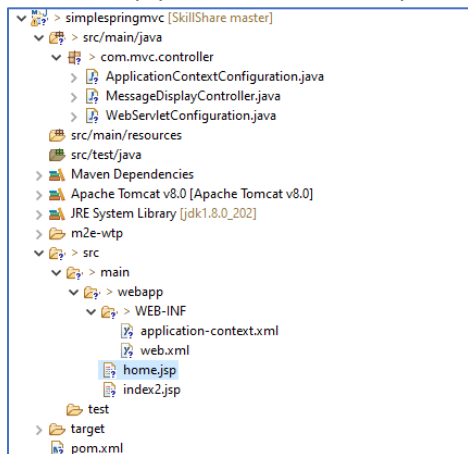
The second bean specification here within this application context corresponds to our view resolver. This maps to the **InternalResourceViewResolver** class in Spring MVC.

All our views are mapped starting at the root document of our web app. The prefixes are all forward / and all our views are JSP files, we use the view suffix .jsp.

3. Contains of MessageDisplayController.java

```
MessageDisplayController.java
1 package com.mvc.controller;
2
3 import javax.servlet.http.HttpServletRequest;
4
5 public class MessageDisplayController implements Controller {
6
7     public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) throws Exception {
8         return new ModelAndView("home");
9     }
10 }
11
12
13
14
15
16
```

The MessageDisplayController implements the controller interface, and/or has overridden the handle request method. This controller returns a model and view object, the logical name of our view is simply home. This will map to the **home.jsp** file that we have in our webapp folder.



4. Contains of WebServletConfiguration file

```

1 package com.mvc.controller;
2
3 import javax.servlet.ServletContext;
4 import javax.servlet.ServletException;
5 import javax.servlet.ServletRegistration;
6
7 import org.springframework.web.WebApplicationInitializer;
8 import org.springframework.web.context.support.XmlWebApplicationContext;
9 import org.springframework.web.servlet.DispatcherServlet;
10
11
12 public class WebServletConfiguration implements WebApplicationInitializer {
13
14     public void onStartUp(ServletContext servletContext) throws ServletException {
15
16         XmlWebApplicationContext appContext = new XmlWebApplicationContext();
17         appContext.setConfigLocation("/WEB-INF/application-context.xml");
18
19         ServletRegistration.Dynamic servlet = servletContext.addServlet("dispatcher",
20             new DispatcherServlet(appContext));
21
22         servlet.setLoadOnStartup(1);
23         servlet.addMapping("/");
24     }
25 }
26
27
28

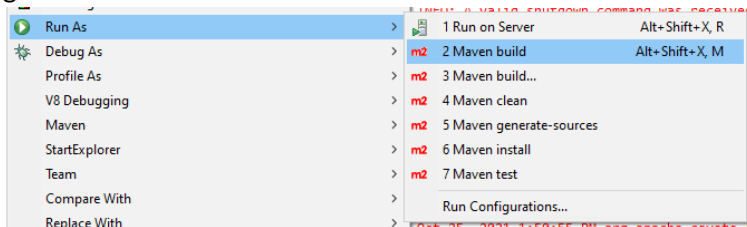
```

The **WebServletConfiguration** still implements the **WebApplicationInitializer** and overrides the **onStartup** method. Now within this startup method, we need to specify the right application context. Since that is an XML based specification, we use the **XmlWebApplicationContext** class. Instantiate this class on line 16 and then call **appContext.setConfigLocation**. And point to where **application-context.xml** file lives which is under the **WEB-INF** folder. The **XmlWebApplicationContext** class will read in this XML file and set up the configuration for your app accordingly.

We set up the **DispatcherServlet** exactly as we did before, pass in the app context. Call **setLoadOnStartup** for this servlet and add the forward **/** mapping, that is our document root.

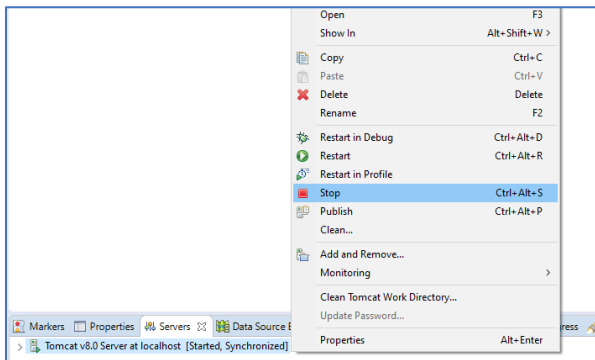
5. Run the app

We are now ready to build and run this Spring MVC application. The first thing is to generate the WAR file, select the project, go to Run As and choose Maven build. This will run a clean build and generate a brand new WAR file.

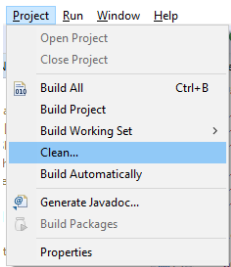


Getting Started with Spring Framework

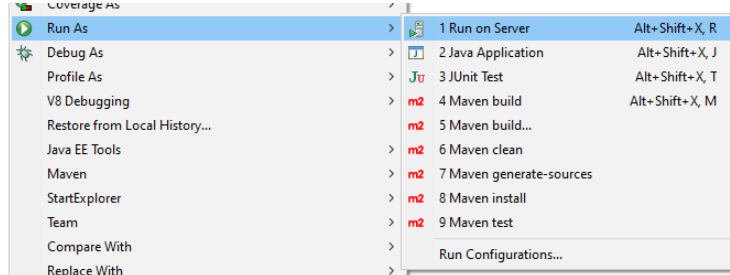
In order to run your app within Eclipse, remember make sure you stop your Tomcat server, clean its cache.



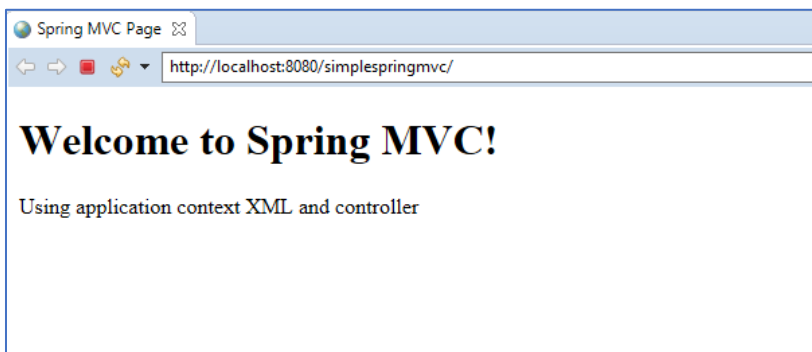
Clean Eclipse's cache as well using project clean and only then should you go ahead and run your server.



And the way you run your server is select the project, right click, go to Run As and say Run on Server.



This will bring up Tomcat and deploy your WAR to Tomcat. And you should be able to see on **localhost:8080/simplespringmvc**, *Welcome to Spring MVC!* We've used an XML-based application context to run this Spring application.



Using Annotation for Bean Specification

Spring MVC applications can be configured and run by programmatically specify the application context using additional Spring MVC specific annotations to discover other components in Spring MVC app as follows

1. Prepare the view entry point "index.jsp"

```
index.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1" isELIgnored="false" %>
3 <html>
4 <head>
5   <meta charset="ISO-8859-1" content="text/html" http-equiv="Content-Type">
6 </head>
7 <title>Spring MVC Page</title>
8 </head>
9 <body>
10   <h1>Welcome to Spring MVC!</h1>
11   <span>We are going to explore a brand new world</span> ${message}
12 </body>
13 </html>
14
```

Take a look at the `${message}`. Within JSP, this is a reference to a server side variable that has been specified as a part of the model that this view is suppose to render. While resolving and rendering this view on the server, the controller needs to specify some value for this message property. And the value of the message property is what will be displayed here within our JSP file. If no value is specified for the message property, Spring will automatically consider this to be an empty string.

2. Prepare the Web MVC Configuration `SpringMvcConfiguration.java` file
This is the file that will contain the bean configuration for our application.

```
SpringMvcConfiguration.java
1 package com.mvc.controller;
2
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.ComponentScan;
5 import org.springframework.context.annotation.Configuration;
6
7 import org.springframework.web.servlet.ViewResolver;
8 import org.springframework.web.servlet.config.annotation.EnableWebMvc;
9 import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
10 import org.springframework.web.servlet.view.InternalResourceViewResolver;
11 import org.springframework.web.servlet.view.JstlView;
12
13
14 @EnableWebMvc
15 @ComponentScan(basePackages = "com.mvc")
16 @Configuration
17 public class SpringMvcConfiguration implements WebMvcConfigurer {
18
19     @Bean
20     public ViewResolver viewResolver() {
21         InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
22
23         viewResolver.setViewClass(JstlView.class);
24         viewResolver.setPrefix("/");
25         viewResolver.setSuffix(".jsp");
26         return viewResolver;
27     }
28
29 }
30
```

The first thing to notice here is that the **SpringMvcConfiguration** class implements the **WebMvcConfigurer** interface. This is *the interface that you have to implement if you want to use*

Java-base configuration for your Spring MVC project rather than XML-based configuration for your servlets.

The first attributes on line 14 is the **@EnableWebMvc**. This basically tells our Spring MVC application to **enable programmatic annotations for components**, which will then be injected into my application. **@EnableWebMvc** will allow Spring to use Java annotations to specify the different components of my Spring MVC application such as repositories, controllers, and so on.

The **@ComponentScan** annotation on line 15 tells Spring MVC the base package which should start scanning for classes which have these annotations. The base package specified here is *com.mvc*. All the Java classes that we defined within sub-packages to this base package will be scanned for annotations. And those objects will be automatically instantiated and injected into our Spring application.

Third annotation, the **@Configuration** annotation. This tells Spring MVC that this Java class should be treated as the programmatic configuration of our Spring MVC servlets.

The only bean specification within our application context here is our **viewResolver**. We continue to use the *InternalResourceViewResolver*. We instantiate an object, we set the view class to be **JstlView.class**. This indicates to Spring that we'll use JSP files with JSTL tags. We'll set the prefix to be / as before, and suffix to be .jsp and we'll return the viewResolver.

This will allow us to reference views using their logical names rather than their actual implementations. JSP here is a specific implementation of the view. Notice that we do not have an explicit method here that returns an implementation of the controller to handle mappings within our Spring MVC application.

That's because of the **@ComponentScan** annotation that we have on line 15. This **ComponentScan** annotation will cause Spring MVC through look through all of the Java code that we have within *com.mvc* and pick up and instantiate any classes that has been tagged with the **@Controller** annotation.

3. Prepare web app to setup configuration in WebServletConfiguration.java file.

```

WebServletConfiguration.java
1 package com.mvc.controller;
2
3 import javax.servlet.ServletContext;
4 import javax.servlet.ServletException;
5 import javax.servlet.ServletRegistration;
6
7 import org.springframework.web.WebApplicationInitializer;
8 import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
9 import org.springframework.web.servlet.DispatcherServlet;
10
11
12 public class WebServletConfiguration implements WebApplicationInitializer {
13
14     public void onStartUp(ServletContext servletContext) throws ServletException {
15
16         AnnotationConfigWebApplicationContext appContext =
17             new AnnotationConfigWebApplicationContext();
18         appContext.register(SpringMvcConfiguration.class);
19         appContext.setServletContext(servletContext);
20
21         ServletRegistration.Dynamic servlet = servletContext.addServlet("dispatcher",
22             new DispatcherServlet(appContext));
23
24         servlet.setLoadOnStartup(1);
25         servlet.addMapping("/");
26     }
27 }
28
29 }
30

```

This implements the **WebApplicationInitializer** interface and override the **onStartup** method, which is injected with the current **ServletContext**. The code within this method have a programmatic specification of the application context, we initialize the **AnnotationConfigWebApplicationContext** object. Register the **SpringMvcConfiguration.class**, and set the current **ServletContext**.

Instantiate the **DispatcherServlet**, make sure you setLoadOnStartup to 1 and add the / mapping.

4. Prepare MessageDisplayController.java.

```

MessageDisplayController.java
1 package com.mvc.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.ui.Model;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.servlet.ModelAndView;
7
8 @Controller
9 public class MessageDisplayController {
10
11     @RequestMapping(value = "/hi")
12     public ModelAndView sayHi(Model model) {
13         model.addAttribute("message", "Hi There");
14         return new ModelAndView("index");
15     }
16
17     @RequestMapping(value = "/begin")
18     public ModelAndView beginLearning(Model model) {
19         model.addAttribute("message", "Let's begin learning");
20         return new ModelAndView("index");
21     }
22 }
23
24

```

Observe that this class Controller **MessageDisplayController.java** has the **@Controller** annotation on the class. From Spring version 3.0 onwards, all you need to do to define a controller class in Spring MVC is to tag it with the **@Controller** annotation. When an **@Controller** annotated controller receives a request, it looks for the right method handler to handle that request.

You can have as many controllers as you want to within a Spring MVC app. It's common practice to logically break up your requests across controllers. In addition, a single controller can specify multiple handler mappings.

There are two mappings here in this controller mapping a request to a handler method. These correspond to the two methods in this class annotated using the **@RequestMapping** annotation. These are on lines 11 and 17.

Now is a good time to formally define a handler mapping. This maps a request, an incoming web request to a particular method, which processes the request. And after performing the processing, it renders a view with the results.

Let's take a look at the first of these handler methods here which has the *RequestMapping /hi*. The */hi* is a relative path relative to the root of our web application and represents the path of the web requests made by the client. The **RequestMapping** annotation, as you can see, is used to decorate a handler method. The name of our method here is **sayHi()**. The name of the method is actually irrelevant, it doesn't really matter. But you might want to choose a meaningful name, just for your sanity. **sayHi** takes as an input argument, in our case, a **Model** parameter. And it returns an object of type **ModelAndView** which contains a model map, as well as the view that needs to be rendered.

The only processing that this handler method does is to add a message property to the model using **model.addAttribute**. The message says, Hi there, this is the same message property that will be accessed and rendered out by our **index.jsp** file.

Another thing to note here is that our **ModelAndView** object returns the virtual name of the view, simply **index** and not **index.jsp**.

This is the logical name of the view with no reference to the actual implementation or the view's technology. That mapping is taken care of by the view resolver.

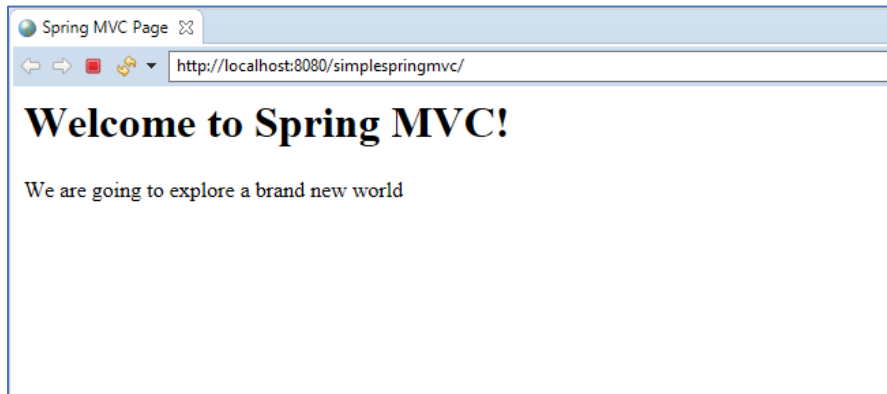
The second handler method here is used to process a different request, request made to the path **/begin**. This path is a relative path to the root of our web application. The actual processing within this handler is straightforward, we add the message property.

The message now is different. It says, Let's begin learning and we return a **ModelAndView** object referencing the **index** logical view. I'm going to right click and build this Maven file. This will generate a new WAR file with my web application and once the build is complete successfully,

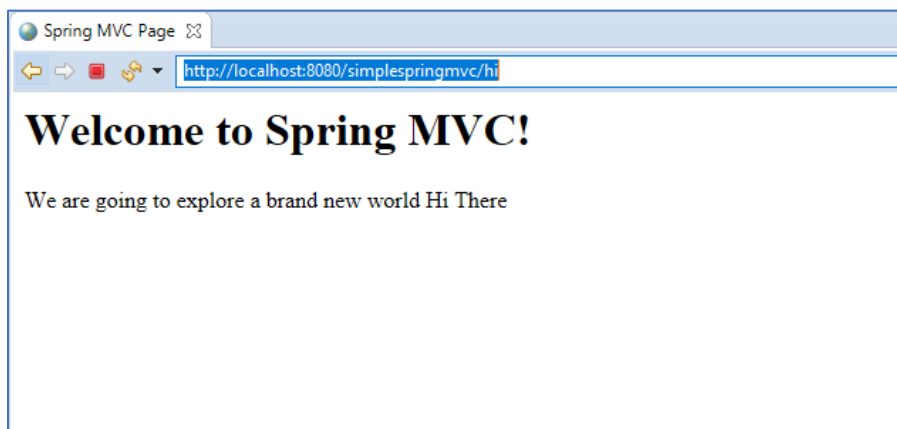
5. Run the Application

To run the application do right click, Run As, Run on Server. Make sure you clean all of the caches for your Tomcat server as well as Eclipse.

This is the root path to my application.

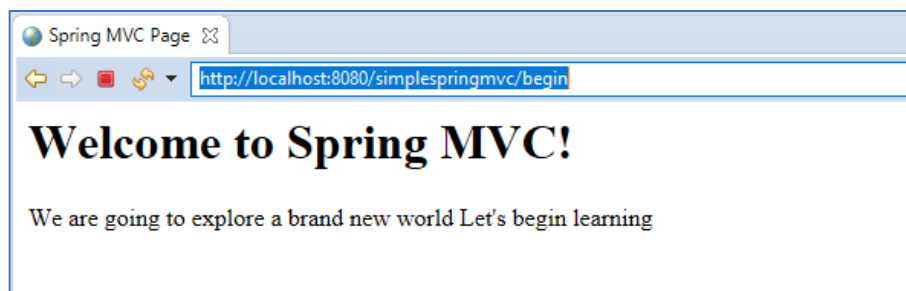


Now Change URL to make a request to the **/hi** path, this will be handled by my controller object. The handler method that is invoked behind the scenes here is the sayHi method. Within the sayHi method, we add a message property, Hi there!, and this message property is displayed here on screen.



We are going to explore a brand new world. Hi there!

Make another request with **/begin** path. Notice, this path is relative to the root of my web app. This request will now be handled by the method that says, beginLearning. The message property is now, Let's begin learning and that is the message that is displayed here on screen.



Configuring the View Resolver

Understand how the viewResolver works. This is what the code currently looks like.

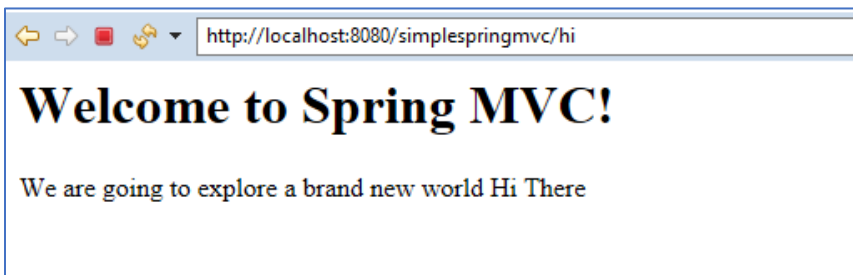
Here is our controller class, the MessageDisplayController, with two handler mappings.

```
8 @Controller
9 public class MessageDisplayController {
10
11     @RequestMapping(value = "/hi")
12     public ModelAndView sayHi(Model model) {
13         model.addAttribute("message", "Hi There");
14         return new ModelAndView("index");
15     }
16
17     @RequestMapping(value = "/begin")
18     public ModelAndView beginLearning(Model model) {
19         model.addAttribute("message", "Let's begin learning");
20         return new ModelAndView("index");
21     }
22 }
23
24
```

A request from the user to the path **/hi**, this is the path that is relative to the root of the application, will render the view **index**. **index** is the logical name of the view, which is mapped to a physical view using the internal **viewResolver**. A request to the path **/begin**, this path is, once again, relative to the root of the application, will render the view **index** once again. Both of these handler methods render the same view. It's the message displayed in the view that is configured and is different.

```
index.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1" isElIgnored="false" %>
3 <html>
4 <head>
5   <meta charset="ISO-8859-1" content="text/html" http-equiv="Content-Type">
6 </head>
7 <title>Spring MVC Page</title>
8 </head>
9 <body>
10   <h1>Welcome to Spring MVC!</h1>
11   <span>We are going to explore a brand new world</span> ${message}
12 </body>
13 </html>
14
```

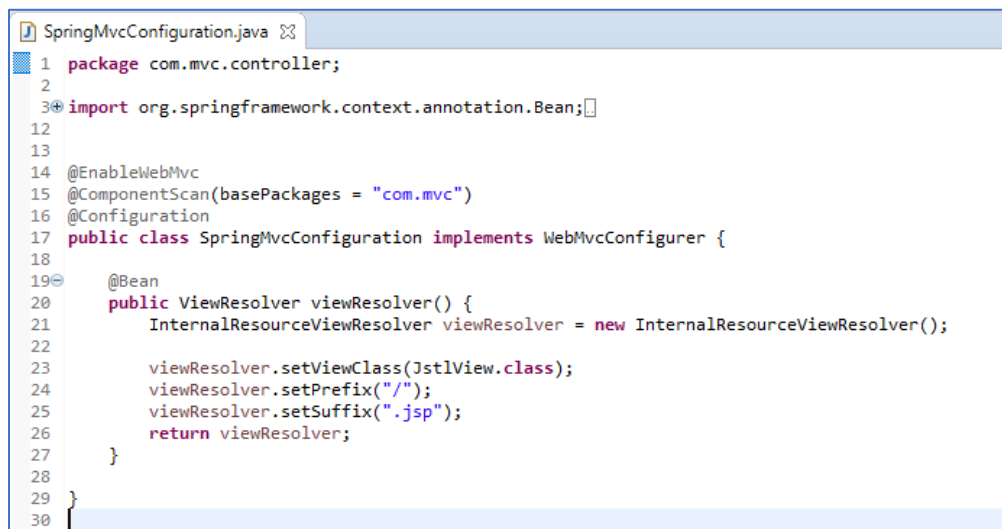
The first message is, Hi there!



The second message is, Let's begin learning.

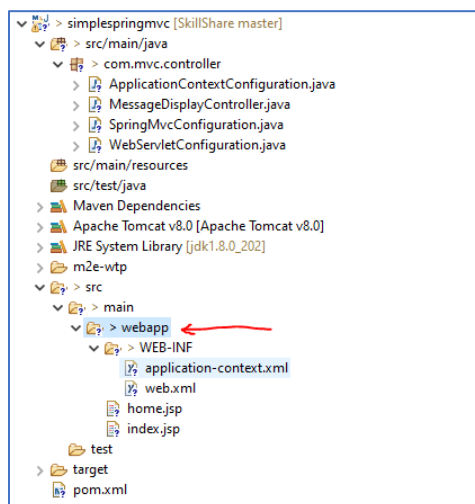


Now, let's head over to the SpringMvcConfiguration.java file. This is where we have specified the bean that is our viewResolver.



We've set the prefix of the view to be / and the suffix to be .jsp. We know that the view here is implemented using .jsp files, and all view files are relative to the root directory.

The root directory of our app is the webapp folder. Files placed in that root directory are at the root of the application.

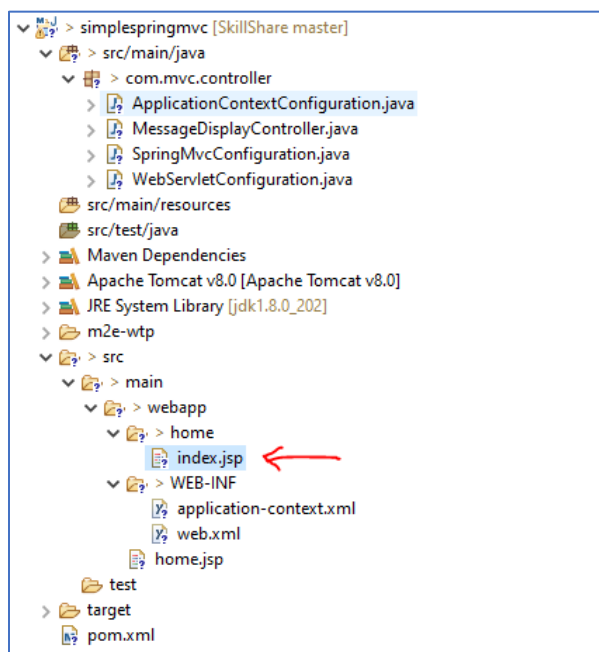


Getting Started with Spring Framework

Now, make a little change here to viewResolver. Instead of having the prefix just be /, I'm going to change the prefix to be /home/. This essentially tells the viewResolver to look for our view implementations in a folder called /home, relative to the root of the web application.

```
SpringMvcConfiguration.java
1 package com.mvc.controller;
2
3 import org.springframework.context.annotation.Bean;
4
5
6
7
8
9
10
11
12
13
14 @EnableWebMvc
15 @ComponentScan(basePackages = "com.mvc")
16 @Configuration
17 public class SpringMvcConfiguration implements WebMvcConfigurer {
18
19     @Bean
20     public ViewResolver viewResolver() {
21         InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
22
23         viewResolver.setViewClass(JstlView.class);
24         viewResolver.setPrefix("/home/");
25         viewResolver.setSuffix(".jsp");
26         return viewResolver;
27     }
28
29 }
30 }
```

Now, that's not where our index.jsp file is, so create a new folder here under the webapp folder. Remember the webapp folder corresponds to the root of our web application. And drag index.jsp file so that it's now under the home folder.

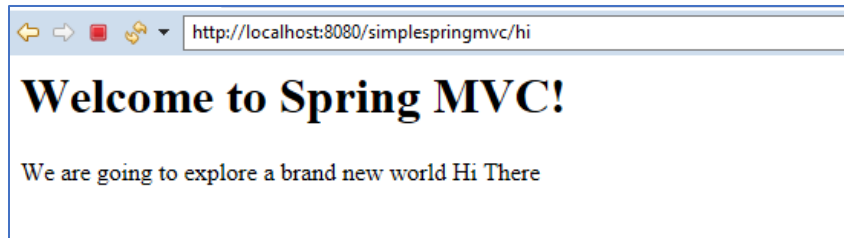


This tells the viewResolver to look for logical views under the home folder, and so when our controller maps to logical views, the right view will be found.

it's time to build and run our code. Run Maven Build, and once the build has succeeded, let's run our code on the Tomcat server. The first thing that you'll notice here when the web application is brought up on the browser window is that you get a 404- Not Found at the root of the web application. At the root path, which just corresponds to localhost:8080/simplespringmvc/, there is no index.jsp file to handle the request. This is what the Tomcat server expects by default. Since we moved the index.jsp file, we've got a status 404 here, file not found, but the relative paths are still mapped correctly.



Let's go to the /hi path, and this time, this will bring up Welcome to Spring MVC and Hi there!



Let's go to the /begin path and this is mapped correctly as well. Welcome to Spring MVC! Let's begin learning.



Summary



In this course, we introduce the Spring framework built on the design principle of inversion of control. And covered the fundamental architecture that Spring MVC offers to build web applications in a simple and intuitive manner. We started off by discussing the basic design principle of the Spring framework, namely inversion of control, which is achieved by our dependency injection. The basic building block of Spring we saw is the bean object. The Spring framework we saw is responsible for the entire life cycle of these bean objects. And these bean objects can be injected as dependencies into the objects that make up the components of our application.

We then moved on to understanding the Model-view-controller paradigm for building robust user interface based applications. The basic structure of a Spring MVC app is based on this pattern. Thus ensuring that developers follow best practices almost by default when they build their apps. We discussed the role of the DispatcherServlet in Spring MVC. And also the use of technologies such as JSP and JSTL.

We finally got hands on building our first Spring MVC application. We saw how we could specify our application context both programmatically, as well as using XML files. And we saw how Spring MVC specific Java annotations can be used to make our classes injectable beans.

Quiz

1. Question: What are Spring Beans?
POJOs which are default implementations available out of the box with Spring
The module which manages data access and integration in Spring
✓ Objects whose lifecycle is entirely managed by Spring
The container which manages dependency injection in Spring
2. Question: Which techniques using Spring will achieve inversion of control?
✓ Server locator pattern
Model-view-controller paradigm
✓ Factory pattern
✓ Dependency injection
Builder pattern
3. Question: What is Aspect Oriented Programming?
Using dependency injection to inject services into objects
✓ Adding functionality to existing code without changing the code itself
Splitting the code into modules, where each module is self-contained
Using annotations to specify features
4. Question: Which dependency management systems can be used with Spring?
Beam
✓ Maven
Spark
✓ Gradle
✓ Ivy
5. Question: Match the components to their role:
Holds the business logic of the application : **Controller**
Responsible for presenting application state to the user : **View**
Responsible for holding and updating application state : **Model**
6. Question: Which technologies does Spring MVC depend on?
✓ JSP
JMX
Hibernate
✓ Servlets
✓ JSTL
7. Question: What is the front controller in Spring MVC called?
View Resolver

Handler Mapping

✓ Dispatcher Servlet

JSP Scriptlet

8. Question: What is Apache Maven?

✓ A build management tool that helps manage Java dependencies for your project

A continuous integration tool that helps run integration tests for your daily build

A syntax highlighting tool that helps catch errors in your code

A web server that can be used to host web applications

9. Question: What are the contents of the pom.xml file?

Handler mappings and servlet configuration

Connection strings to connect to your database

Application specific configuration for Spring MVC

✓ Java dependencies for your project

10. Question: What is the name of the servlet that handles all the incoming web requests to your Spring MVC app?

Controller Servlet

Front Servlet

✓ Dispatcher Servlet

Application Servlet

11. Question: How do we clear the cache in the Tomcat server?

Right-click on your Eclipse project and choose "Update"

Stop and restart the server within Eclipse

✓ Right-click on the server within Eclipse and choose "Clean"

Click on the top Build menu within Eclipse and choose "Clean"

12. Question: Where should you place the XML file that holds your application context?

In src/test

In src/main

✓ In webapps/WEB-INF

In webapps/

13. Question: What annotation should you use to specify handler mappings in Spring MVC?

@Controller

✓ @RequestMapping

@HandlerMapping

@PathMapping

14. Question: Which folder is the root folder of your web application?

webapps/WEB-INF

src/main

src/test

✓ webapps/