Spring MVC

MR. Ashok

Face Book Group name: Ashok IT School
Email: ashok.javatraining@gmail.com

We spend our lives fiddling with countless devices and we spend many hours surfing the Internet and visiting all kinds of websites. Sometimes it's due to work-related reasons, although most of the time it's just for leisure or sheer entertainment. In recent years, it seems that any work can be done by a simple click. The app revolution and — by extension — the web app development revolution, has brought everyone closer to a world in which new technologies are growing by leaps and bounds.

To begin this, first we should start with what is web application?

A web-based application is any application that uses a website as the interface or front-end. Users can easily access the application from any computer connected to the Internet using a standard browser.

This contrasts with traditional desktop applications, which are installed on a local computer. For example, most of us are familiar with Microsoft Word, a common word-processing application that is a desktop application.

On the other hand, Google Docs is also a word-processing application but users perform all the functions using a web browser instead of using software installed on their computer. This means that it is a web-based application.

What are the business advantages of web applications?

Cost effective development: With web-based applications, users access the system via a uniform environment—the web browser. While the user interaction with the application needs to be thoroughly tested on different web browsers, the application itself needs only be developed for a single operating system.

There is no need to develop and test it on all possible operating system versions and configurations. This makes development and troubleshooting much easier and for web applications that use a Flash front end testing and troubleshooting is even easier.

Accessible anywhere: Unlike traditional applications, web systems are accessible anytime, anywhere and via any PC with an Internet connection. This puts the user firmly in charge of where and when they access the application.

It also opens up exciting, modern possibilities such as global teams, home working and real-time collaboration. The idea of sitting in front of a single computer and working in a fixed location is a thing of the past with web-based applications.

Easily customizable: The user interface of web-based applications is easier to customize than is the case with desktop applications. This makes it easier to update the look and feel of the application or to customize the presentation of information to different user groups.

Therefore, there is no longer any need for everyone to settle for using exactly the same interface at all times. Instead, you can find the perfect look for each situation and user.

Accessible for a range of devices: In addition to being customizable for user groups, content can also be customized for use on any device connected to the internet. This includes the likes of PDAs, mobile phones and tablets.

This further extends the user's ability to receive and interact with information in a way that suits them. In this way, the up to date information is always at the fingertips of the people who need it.

Improved interoperability: It is possible to achieve a far greater level of interoperability between web applications than it is with isolated desktop systems. For example, it is much easier to integrate a web-based shopping cart system with a web-based accounting package than it is to get two proprietary systems to talk to each other.

Because of this, web-based architecture makes it possible to rapidly integrate enterprise systems, improving workflow and other business processes. By taking advantage of internet technologies you get a flexible and adaptable business model that can be changed according to shifting market demands.

Easier installation and maintenance: With the web-based approach installation and maintenance becomes less complicated too. Once a new version or upgrade is installed on the host server all users can access it straight away and there is no need to upgrade the PC of each and every potential user.

Rolling out new software can be accomplished more easily, requiring only that users have up-to-date browsers and plugins. As the upgrades are only performed by an experienced professional to a single server the results are also more predictable and reliable.

Adaptable to increased workload: Increasing processor capacity also becomes a far simpler operation with web-based applications. If an application requires more power to perform tasks only the server hardware needs to be upgraded.

The capacity of web-based software can be increased by "clustering" or running the software on several servers simultaneously. As workload increases, new servers can be added to the system easily.

For example, Google runs on thousands of inexpensive Linux servers. If a server fails, it can be replaced without affecting the overall performance of the application.

Increased Security: Web-based applications are typically deployed on dedicated servers, which are monitored and maintained by experienced server administrators. This is far more effective than monitoring hundreds or even thousands of client computers as is the case with desktop applications.

This means that security is tighter and any potential breaches should be noticed far more quickly.

Flexible core technologies: Any of three core technologies can be used for building web-based applications, depending on the requirements of the application. The Java-based solutions (J2EE) from Sun Microsystems involve technologies such as JSP and Servlets.

The newer Microsoft .NET platform uses Active Server Pages, SQL Server and .NET scripting languages. The third option is the Open Source platform (predominantly PHP and MySQL), which is best suited to smaller websites and lower budget applications.

Conclusion

- Web-based applications are:
- Easier to develop
- More useful for your users
- Easier to install, maintain and keep secure

Spring Web MVC

Spring framework makes the development of web applications very easy by providing the Spring MVC module. Spring MVC module is based on two most popular design patterns, they are

1) **Front controller Design pattern**
2) **MVC Design pattern**

Let us have glance at these Design Patterns before starring Spring Web MVC.

Web is having more and more demand these days. Since the desire of the companies and organizations are increasing so the complexity and the performance of the web programming matters. Complexity with the different types of communication devices is increasing. The business is demanding applications using the web and many communication devices. So with the increase load of the data on the internet we have to take care of the architecture issue. To overcome these issues, we need to have idea about design models. There are two types of design models available in java, they are

1. **Model 1 Architecture**
2. **Model 2 (MVC) Architecture**

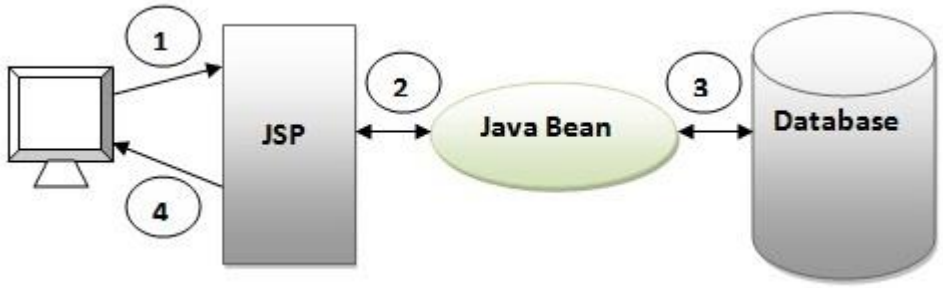
Model 1 Architecture

Servlet and JSP are the main technologies to develop the web applications.

Servlet was considered superior to CGI. Servlet technology doesn't create process, rather it creates thread to handle request. The advantage of creating thread over process is that it doesn't allocate separate memory area. Thus many subsequent requests can be easily handled by servlet.

Problem in Servlet technology Servlet needs to recompile if any designing code is modified. It doesn't provide separation of concern. Presentation and Business logic are mixed up.

JSP overcomes almost all the problems of Servlet. It provides better separation of concern, now presentation and business logic can be easily separated. You don't need to redeploy the application if JSP page is modified. JSP provides support to develop web Application using JavaBean, custom tags and JSTL so that we can put the business logic separate from our JSP that will be easier to test and debug.



As you can see in the above figure, there is picture which show the flow of the model1 architecture.

Browser sends request for the JSP page
JSP accesses Java Bean and invokes business logic
Java Bean connects to the database and get/save data
Response is sent to the browser which is generated by JSP

Advantage of Model 1 Architecture

- Easy and Quick to develop web application

Disadvantage of Model 1 Architecture

- Navigation control is decentralized since every page contains the logic to determine the next page. If JSP page name is changed that is referred by other pages, we need to change it in all the pages that leads to the maintenance problem.
- Time consuming you need to spend more time to develop custom tags in JSP. So that we don't need to use script let tag
- Hard to extend It is better for small applications but not for large applications.

Model 2 Architecture

Model View Controller (MVC) is a software architecture pattern, commonly used to implement user interfaces: it is therefore a popular choice for architecting web apps. In general, it separates out the application logic into three separate parts, promoting modularity and ease of collaboration and reuse. It also makes applications more flexible and welcoming to iterations.

The MVC pattern was first described in 1979 by Trygve Reenskaug, then working on Smalltalk at Xerox research labs. The original implementation is described in depth in the influential paper "Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller".

Smalltalk's MVC implementation inspired many other GUI frameworks such as:

- The NeXTSTEP and OPENSTEP development environments encourage the use of MVC. Cocoa and GNUstep, based on these technologies, also use MVC.
- Microsoft Foundation Classes (MFC) (also called Document/View architecture)
- Java Swing
- The Qt Toolkit (since Qt4 Release).
- XForms has a clear separation of model (stored inside the HTML head section) from the presentation (stored in the HTML body section). XForms uses simple bind commands to link the presentation to the model.

MVC stands for Model, View and Controller. MVC separates application into three components - Model, View and Controller.

Model: The Model is where data from the controller and sometimes the view is actually passed into, out of, and manipulated. Keeping in mind our last example of logging into your web-based email, the model will take the username and password given to it from the controller, check that data against the stored information in the database, and then render the view accordingly. For example, if you enter in an incorrect password, the model will tell the controller that it was incorrect, and the controller will tell the view to display an error message saying something to the effect of "Your username or password is incorrect."

Model is a data and business logic.

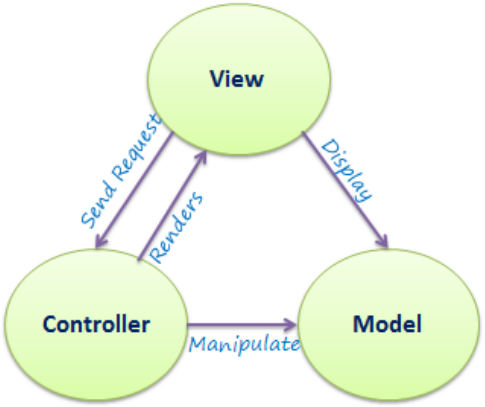
View: In a web-based application, **the view** is exactly what it sounds like: the visible interface that the user interacts with, displaying buttons, forms, and information. Generally speaking, the controller calls up the view after interacting with the model, which is what gathers the information to display in the particular view.

View is a User Interface.

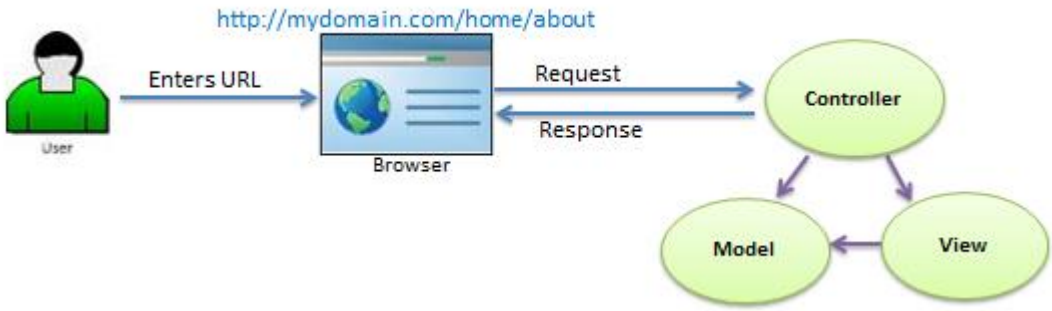
Controller: The Controller is essentially the traffic cop of the application, directing traffic to where it needs to go, figuring out which view it needs to load up, and interacting with the appropriate models. For example, when you go to login to your email on a website, the controller is going to tell the application that it needs to load the login form view. Upon attempting to login, the controller will load the model that handles logins, which will check if the username and password match what exists within the system. If successful, the controller will then pass you off to the first page you enter when logging in, such as your inbox. Once there, the inbox controller will further handle that request.

Controller is a request handler.

The following figure illustrates the interaction between Model, View and Controller



The following figure illustrates the flow of the user's request



As per the above figure, when the user enters a URL in the browser, it goes to the server and calls appropriate controller. Then, the Controller uses the appropriate View and Model and creates the response and sends it back to the user.

Though MVC comes in different flavors, the control flow generally works as follows:

- The user interacts with the user interface in some way (e.g., user presses a button)
- A controller handles the input event from the user interface, often via a registered handler or callback.
- The controller accesses the model, possibly updating it in a way appropriate to the user's action (e.g., controller updates user's shopping cart). Complex controllers are often structured using the command pattern to encapsulate actions and simplify extension.
- A view uses the model to generate an appropriate user interface (e.g., view produces a screen listing the shopping cart contents). The view gets its own data from the model. The model has no direct knowledge of the view. (However, the observer pattern can be used to allow the model to indirectly notify interested parties, potentially including views, of a change.)
- The user interface waits for further user interactions, which begins the cycle anew.

Conclusion

Now we know the basic concepts of the MVC pattern. Essentially, it allows for the programmer to isolate these very separate pieces of code into their own domain, which makes code maintenance and debugging much simpler than if all of these items were chunked into one massive piece. If I have a problem with an application not displaying an error message when it should, I have a very specific set of locations to look to see why this is not happening. First I would look at the "Login Controller" to see if it is telling the view to display the error. If that's fine, I would look at the "Login Model" to see if it is passing the data back to the controller to tell it that it needs to show an error. Then if that's correct, the last place it could be happening would be in the "Login View."

Front Controller design pattern

When the user access the view directly without going thought any centralized mechanism each view is required to provide its; own system services, often resulting in duplication of code and view navigations is left to the views, this may result in commingled view content and view navigation.

A centralized point of contact for handling a request may be useful, for example, to control and log a user's progress through the site. .

Introducing a controller as the initial point of contact for handling a request. The controller manages the handling of the request, including invoking security services such as authentication and authorization, delegating business processing, managing the choice of an appropriate view, handling errors, and managing the selection of content creation strategies.

The controller provides a centralized entry point that controls and manages Web request handling. By centralizing decision points and controls, the controller also helps reduce the amount of Java code, called scriptlets, embedded in the JavaServer Pages (JSP) page.

Centralizing control in the controller and reducing business logic in the view promotes code reuse across requests. It is a preferable approach to the alternative-embedding code in multiple views-because that approach may lead to a more error-prone, reuse-by-copy- and-paste environment.

Typically, a controller coordinates with a dispatcher component. Dispatchers are responsible for view management and navigation. Thus, a dispatcher chooses the next view for the user and vectors control to the resource. Dispatchers may be encapsulated within the controller directly or can be extracted into a separate component.

The responsibilities of the components participating in this patterns are:

Controller: The controller is the initial contact point for handling all requests in the system. The controller may delegate to a helper to complete authentication and authorization of a user or to initiate contact retrieval.

Dispatcher: A dispatcher is responsible for view management and navigation, managing the choice of the next view to present to the user, and providing the mechanism for vectoring control to this resource. A dispatcher can be encapsulated within a controller or can be a separate component working in coordination. The dispatcher provides either a static dispatching to the view or a more sophisticated dynamic dispatching mechanism. The dispatcher uses the RequestDispatcher object (supported in the servlet specification) and encapsulates some additional processing.

Helper: A helper is responsible for helping a view or controller complete its processing. Thus, helpers have numerous responsibilities, including gathering data required by the view and storing this intermediate model, in which case the helper is sometimes referred to as a value bean. Additionally, helpers may adapt this data model for use by the view. Helpers can service requests for data from the view by simply providing access to the raw data or by formatting the data as Web content. A view may work with any number of helpers, which are typically implemented as JavaBeans components (JSP 1.0+) and custom tags (JSP 1.1+). Additionally, a helper may represent a Command object, a delegate, or an XSL Transformer, which is used in combination with a stylesheet to adapt and convert the model into the appropriate form.

View: A view represents and displays information to the client. The view retrieves information from a model. Helpers support views by encapsulating and adapting the underlying data model for use in the display.

Front Controller centralizes control. A controller provides a central place to handle system services and business logic across multiple requests. A controller manages business logic processing and request handling. Centralized access to an application means that requests are easily tracked and logged. Keep in mind, though, that as control centralizes, it is possible to introduce a single point of failure. In practice, this rarely is a problem, though, since multiple controllers typically exist, either within a single server or in a cluster.

Front Controller improves manageability of security. A controller centralizes control, providing a choke point for illicit access attempts into the Web application. In addition, auditing a single entrance into the application requires fewer resources than distributing security checks across all pages.

Front Controller improves reusability. A controller promotes cleaner application partitioning and encourages reuse, as code that is common among components moves into a controller or is managed by a controller.

Benefits

The following lists the benefits of using the Front Controller pattern:

- ✓ Centralizes control logic
- ✓ Improves reusability
- ✓ Improves separation of concerns

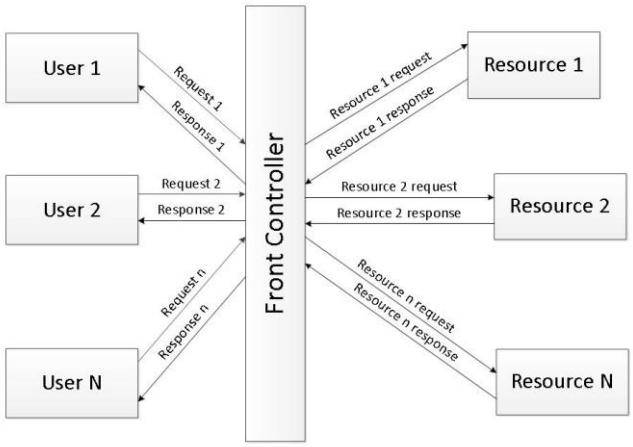
When to Use

You should use the Front Controller pattern to:

- ✓ Apply common logic to multiple requests
- ✓ Separate processing logic from view

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



Introduction to the spring web MVC framework

Spring's web MVC framework is designed around a **DispatcherServlet** that dispatches requests to handlers, with configurable handler mappings, view resolution, locale and theme resolution as well as support for upload files. The default handler is a very simple Controller interface, just offering a ModelAndView handleRequest (request, response) method. This can already be used for application controllers, but you will prefer the included implementation hierarchy, consisting of, for example AbstractController, AbstractCommandController and SimpleFormController. Application controllers will typically be subclasses of those. Note that you can choose an appropriate base class: If you don't have a form, you don't need a FormController. This is a major difference to Struts.

You can use any object as a command or form object - there's no need to implement an interface or derive from a base class. Spring's data binding is highly flexible, for example, it treats type mismatches as validation errors that can be evaluated by the application, not as system errors. So you don't need to duplicate your business objects' properties as Strings in your form objects, just to be able to handle invalid submissions, or to convert the Strings properly. Instead, it is often preferable to bind directly to your business objects. This is another major difference to Struts which is built around required base classes like Action and Action Form - for every type of action.

Compared to Web Work, spring has more differentiated object roles. It supports the notion of a Controller, an optional command or form object, and a model that gets passed to the view. The model will normally include the command or form object but also arbitrary reference data. Instead, a Web Work Action combines all those roles into one single object. Web Work does allow you to use existing business objects as part of your form, but only by making them bean properties of the respective Action class.

Finally, the same Action instance that handles the request is used for evaluation and form population in the view. Thus, reference data needs to be modeled as bean properties of the Action too. These are arguably too many roles for one object.

Spring's view resolution is extremely flexible. A Controller implementation can even write a view directly to the response, returning null as ModelAndView. In the normal case, a ModelAndView instance consists of a view name and a model Map, containing bean names and corresponding objects (like a command or form, containing reference data).

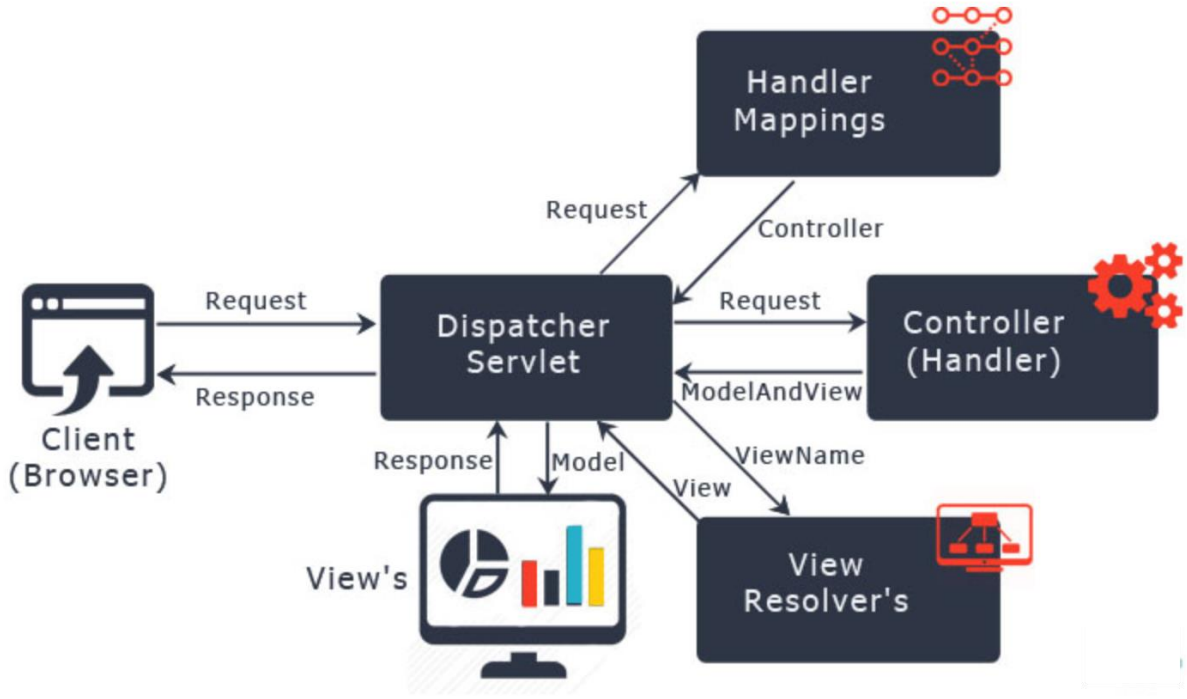
View name resolution is highly configurable, either via bean names, via a properties file, or via your own ViewResolver implementation. The abstract model Map allows for complete abstraction of the view technology, without any hassle. Any renderer can be integrated directly, whether JSP, Velocity, or any other rendering technology. The model Map is simply transformed into an appropriate format, such as JSP request attributes or a Velocity template model.

Features of Spring MVC

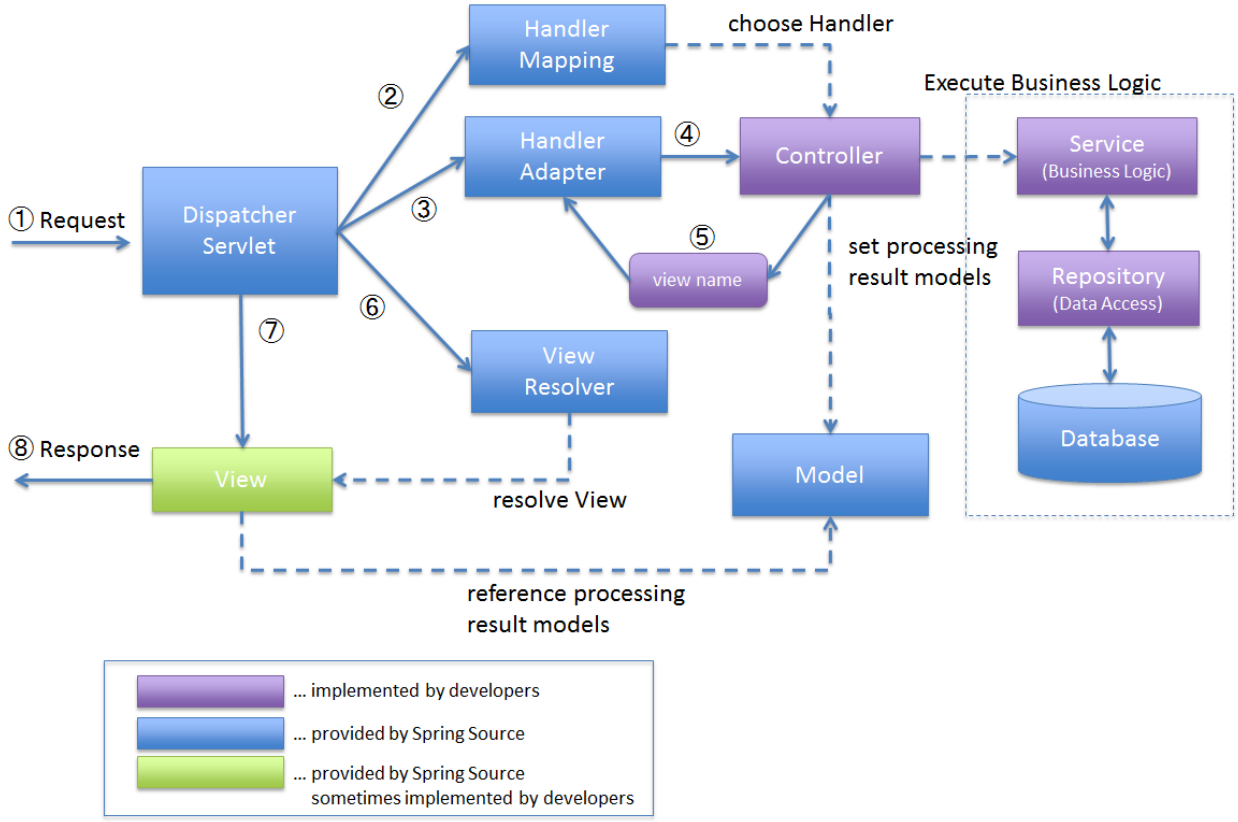
Spring's web module provides a wealth of unique web support features, including:

- **Clear separation of roles -** controller, validator, command object, form object, model object, DispatcherServlet, handler mapping, view resolver, etc. Each role can be fulfilled by a specialized object.
- Powerful and straightforward configuration of both framework and application classes as JavaBeans, including easy referencing across contexts, such as from web controllers to business objects and validators.
- **Adaptability, non-intrusiveness -** Use whatever controller subclass you need (plain, command, form, wizard, multi-action, or a custom one) for a given scenario instead of deriving from a single controller for everything.
- **Reusable business code -** no need for duplication. You can use existing business objects as command or form objects instead of mirroring them in order to extend a particular framework base class.
- **Customizable binding and validation -** type mismatches as application-level validation errors that keep the offending value, localized date and number binding, etc instead of String-only form objects with manual parsing and conversion to business objects.
- **Customizable handler mapping and view resolution -** handler mapping and view resolution strategies range from simple URL-based configuration, to sophisticated, purpose-built resolution strategies. This is more flexible than some web MVC frameworks which mandate a particular technique.
- **Flexible model transfer -** model transfer via a name/value Map supports easy integration with any view technology.
- **Customizable locale and theme resolution**, support for JSPs with or without Spring tag library, support for JSTL, support for Velocity without the need for extra bridges, etc.
- **A simple but powerful tag library** - That avoids HTML generation at any cost, allowing for maximum flexibility in terms of markup code.

Spring MVC Architecture



Spring MVC Architecture



Request Flow execution:

1. Client sends HTTP requests
2. Client's requests are intercepted by "DispatcherServlet", which tries to figure out appropriate handler mapping. DispatcherServlet consults Handler Mapping to identify Controller who is supposed to handle this request.
3. Based on Handler Mapper response Dispatcher Servlet invokes the Controller associated with the request. Now, the controller processes the requests invoking appropriate service method and returns the response/result (i.e.; ModelAndView instance which contains model data and view name) back to the DispatcherServlet.
4. Then DispatcherServlet sends the view name from ModelAndView instance to view resolver object.
5. **Time to show output to the user:** view name along with model data will render the response/result back to the user.

DispatcherServlet

Spring's web MVC framework is, like many other web MVC frameworks, a request-driven web MVC framework, designed around a servlet that dispatches requests to controllers and offers other functionality facilitating the development of web applications.

Like ordinary servlets, the DispatcherServlet is declared in the web.xml of your web application. Requests that you want the DispatcherServlet to handle, will have to be mapped, using a URL mapping in the same web.xml file like below

web.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    id="WebApp_ID" version="3.1">
    <display-name>SpringMVCApp1</display-name>

    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.htm</url-pattern>
    </servlet-mapping>
</web-app>
```

As per above configuration, all requests ending with .htm will be handled by the DispatcherServlet.

While Initializing the DispatcherServlet, it looks for the configuration file (DispatcherServletName-servlet.xml) for the web component beans declarations. Based on the above configuration it looks for dispatcher-servlet.xml file. Now loads all the bean declarations and creates a WebApplicationContainer with the web component beans.

Note: We can customize the [DispatcherServletName]-servlet.xml file name by configuring init-parameter to DispatcherServlet. Where param-name will be 'contextConfigLocation' and param-value will be path of the configuration file.

web.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    id="WebApp_ID" version="3.1">
    <display-name>SpringMVCApp1</display-name>

    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/Web-MVC-Beans.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.htm</url-pattern>
    </servlet-mapping>
</web-app>
```

But in typical web application we will have 2 types of components. They are

1) Web Components (Handler Mappers, Controller, View Resolvers, Themes etc)
2) Business Components (Service , Dao and DataSource etc)

It is not recommended to combine Business components with Web Components in Single Bean configuration file. So we will create another ApplicationContext for managing business components.

- WebComponents will be handled by WebApplicationContext
- Business components will be handled by ApplicationContext

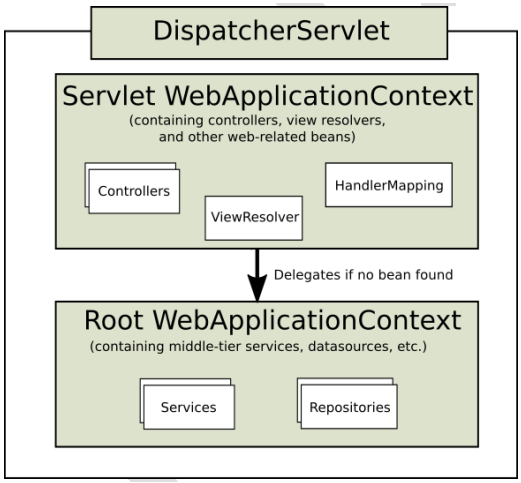
Configuring ApplicationContext

```
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/rootApplicationContext.xml</param-value>
    </context-param>
```

ContextLoaderListener

Performs the actual initialization work for the root application context. Reads a "contextConfigLocation" context-param and passes its value to the context instance, parsing it into potentially multiple file paths which can be separated by any number of commas and spaces, e.g. "WEB-INF/applicationContext1.xml, WEB-INF/applicationContext2.xml".

Typical Context hierarchy in Spring Web MVC



Handler Mapping

HandlerMapping is an interface that is implemented by all Objects that map the request to the corresponding Handler Object. The default Implementations used by the DispatcherServlet are BeanNameUrlHandlerMapping and DefaultAnnotationHandlerMapping.

Below are the some Handler mappers in Spring

1. **BeanNameUrlHandlerMapping**
2. **DefaultAnnotationHandlerMapping**
3. **ControllerBeanNameHandlerMapping**
4. **ControllerClassNameHandlerMapping**
5. **SimpleUrlHandlerMapping**
6. **RequestMappingHandlerMapping (This was introduced in spring 3.1)**

BeanNameURLHandlerMapping

BeanNameUrlHandlerMapping maps request URLs to beans with the same name.

Two ways to define bean name for BeanNameUrlHandlerMapping:

1. You can define static URL in bean name as we have do with the name "/hello.htm", whenever the url "/hello.htm" will be requested by browser, BeanNameUrlHandlerMapping will return respective url.
2. You can define "*" in bean name to hold any number of characters in between the name. As we have done with the bean name "/sayHello*". All the requests with URL starting with "/sayHello" will be parsed to this bean. We will see this in our example.

dispatcher-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:mvc="http://www.springframework.org/schema/mvc"
xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

    <bean name="/hello.htm" class="com.webapp.controller.HelloController"/>

    <bean name="/sayHello*" class="com.webapp.controller.HelloController"/>

    <bean id="urlHandler" class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>


    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/"/>
        <property name="suffix" value=".jsp"/>
    </bean>

</beans>
```

Note : BeanNameUrlHandlerMapping is the default url handler used by Spring MVC. That means if we do not specify any url handler in our Spring MVC configuration, Spring will automatically load this url handler and pass requested urls to BeanNameUrlHandlerMapping to get the controller bean to be executed.

SimpleURLHandlerMapping

This type of HandlerMapping is the simplest of all handler mappings which allows you specify URL pattern and handler explicitly.

There are two ways of defining SimpleUrlHandlerMapping, using <value> tag and <props> tag. SimpleUrlHandlerMapping has a property called mappings we will be passing the URL pattern to it.

```
<!-- Using Value Tag -->
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <value>
            /welcome.htm=welcomeController
            /welcome*=welcomeController
            /hell*=helloWorldController
            /helloWorld.htm=helloWorldController
        </value>
    </property>
</bean>

<!--  Using Prop Tag -->
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/welcome.htm">welcomeController</prop>
            <prop key="/welcome*">welcomeController</prop>
            <prop key="/helloworld">helloWorldController</prop>
            <prop key="/hello*">helloWorldController</prop>
            <prop key="/HELLOworld">helloWorldController</prop>
        </props>
    </property>
</bean>
```

Left Side of "=" is URL Pattern and right side is the id or name of the bean

ControllerClassNameHandlerMapping

ControllerClassHandlerMapping uses a convention to map the requested URL to the Controller. It will take the Controller name and converts them to lower case with a leading "/".

```
<bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping" />

<bean class="com.app.controller.HelloWorldController"></bean>
<bean class="com.app.controller.WelcomeController"></bean>

<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/" />
    <property name="suffix" value=".jsp" />
</bean>
```

As per above configuratoin

- helloworld is requested, the DispatcherServlet redirects it to the HelloWorldController.
- helloworld123 is requested, the DispatcherServlet redirects it to the HelloWorldController.
- welcome is requested, the DispatcherServlet redirects it to the WelcomeController.
- welcome123 is requested, the DispatcherServlet redirects it to the WelcomeController.
- helloWorld is requested, you will get 404 error as "W" is capitalized here.

RequestMappingHandlerMapping

This was introduced in spring 3.1. Earlier the DefaultAnnotationHandlerMapping decided which controller to use and the AnnotationMethodHandlerAdapter selected the actual method that handled the request. RequestMappingHandlerMapping does both the tasks. Therefore the request is directly mapped right to the method. The following types can be passed to method handlers.

MessageController.java

```java
package com.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class MessageController {

    @RequestMapping("/wish")
    public ModelAndView generateWishMsg() {
        String message = "Welcome to Ashok IT School..!!";

        ModelAndView mav = new ModelAndView();
        mav.setViewName("welcome");
        mav.addObject("msg", message);

        return mav;
    }

}
```

Controllers

In Spring MVC, we write a controller class to handle requests coming from the client. Then the controller invokes a business class to process business-related tasks, and then redirects the client to a logical view name which is resolved by the spring's dispatcher servlet in order to render results or output. That completes a round trip of a typical request-response cycle.

Spring MVC provides many abstract controllers, which is designed for specific tasks. Here is the list of anstract controllers that comes with the Spring MVC module:

1. SimpleFormController
2. AbstractController
3. AbstractCommandController
4. CancellableFormController
5. AbstractCommandController
6. MultiActionController
7. ParameterizableViewController
8. ServletForwardingController
9. ServletWrappingController
10. UrlFilenameViewController
11. AbstractController
12. AbstractCommandController
13. SimpleFormController
14. CancellableFormController etc.

Controller.java

```java
package com.controller;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;

public interface Controller {

    /**
     * Process the request and return a ModelAndView object which the
     * DispatcherServlet will render.
     */
    public ModelAndView handleRequest(HttpServletRequest req, HttpServletResponse res) throws Exception;

}
```

As you can see, the Controller interface requires a single method that should be capable of handling a request and returning an appropriate model and view. These three concepts are the basis for the Spring MVC implementation - *ModelAndView* and *Controller*. While the Controller interface is quite abstract, Spring offers a lot of controllers that already contain a lot of the functionality you might need. The Controller interface just defines the most common functionality required of every controller - handling a request and returning a model and a view.

Spring 2.5 introduced an annotation-based programming model for MVC controllers that uses annotations such as @RequestMapping, @RequestParam, @ModelAttribute, and so on. This annotation support is available for both Servlet MVC and Portlet MVC. Controllers implemented in this style do not have to extend specific base classes or implement specific interfaces. Furthermore, they do not usually have direct dependencies on Servlet or Portlet APIs, although you can easily configure access to Servlet or Portlet facilities.

```java
@Controller
public class HelloWorldController {

   @RequestMapping("/helloWorld")
   public String helloWorld(Model model) {

      //setting data to model obj (key-value)
      model.addAttribute("message", "Hello World!");

      //returing logical view name
      return "helloWorld";

   }
}
```

As you can see, the @Controller and @RequestMapping annotations allow flexible method names and signatures. In this particular example the method accepts a Model and returns a view name as a String, but various other method parameters and return values can be used as explained later in this section. @Controller and @RequestMapping and a number of other annotations form the basis for the Spring MVC implementation. This section documents these annotations and how they are most commonly used in a Servlet environment.

The @Controller annotation indicates that a particular class serves the role of a *controller*. Spring does not require you to extend any controller base class or reference the Servlet API. However, you can still reference Servlet-specific features if you need to.

The @Controller annotation acts as a stereotype for the annotated class, indicating its role. The dispatcher scans such annotated classes for mapped methods and detects @RequestMapping annotations (see the next section).
You can define annotated controller beans explicitly, using a standard Spring bean definition in the dispatcher's context. However, the @Controller stereotype also allows for autodetection, aligned with Spring general support for detecting component classes in the classpath and auto-registering bean definitions for them.

To enable autodetection of such annotated controllers, you add component scanning to your configuration. Use the *spring-context* schema as shown in the following XML snippet:

dispatcher-servlet.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

    <!-- To Enable Annotations -->
    <mvc:annotation-driven />

    <!-- To Scan Component classes -->
    <context:component-scan base-package="com.app.*" />

    <!-- View Resolver -->
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>

</beans>
```

Mapping Requests with @RequestMapping

You use the @RequestMapping annotation to map URLs such as /appointments onto an entire class or a particular handler method. Typically the class-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations narrowing the primary mapping for a specific HTTP method request method ("GET", "POST", etc.) or an HTTP request parameter condition.

```
package com.sb.apps.controller;

@Controller
public class EmpController {

	@Autowired(required = true)
	private EmpService service;

	@RequestMapping(value = "/", method = RequestMethod.GET)
	public String index(Model model) {
		model.addAttribute("e", new Employee());
		return "index";
	}

	@RequestMapping(value = "saveData", method = RequestMethod.POST)
	public String saveData(@ModelAttribute("e") Employee emp, Model model) {
		boolean flag = service.save(emp);
		if (flag) {
			model.addAttribute("msg", "Record inserted Successfully..");
		}
		return "index";
	}
}
```

```
@RequestMapping(value = "viewAll", method = RequestMethod.GET)
public String getAllEmps(@RequestParam(name = "p", defaultValue = "0") int pageNo, Model model) {
	try {
		List<Employee> emps = service.getAllEmps(pageNo);
			model.addAttribute("emps", emps);
	} catch (Exception e) {
		return "error";
	}

	return "display";
}
```

```
@RequestMapping(value = "deleteEmp", method = RequestMethod.GET)
public String delete(@RequestParam("eid") Integer eid) {
	Employee emp = new Employee();
	emp.setEmpId(eid);
	service.deleteEmp(emp);
	return "redirect:viewAll";
}
```

```
@RequestMapping(value = "editEmp", method = RequestMethod.GET)
public String edit(@RequestParam("eid") Integer eid, Model model) {
	Employee emp = service.getEmpById(eid);
	model.addAttribute("e", emp);
	return "index";
}
```

Views and resolving them

All MVC frameworks for web applications provide a way to address views. Spring provides view resolvers, which enable you to render models in a browser without tying you to a specific view technology. Out of the box, Spring enables you to use Java Server Pages, Velocity templates and XSLT views, The two interfaces which are important to the way Spring handles views are ViewResolver and View. The ViewResolverprovides a mapping between view names and actual views. The View interface addresses the preparation of the request and hands the request over to one of the view technologies.

ViewResolvers

As discussed all controllers in the spring web MVC framework, return a ModelAndViewinstance. Views in spring are addressed by a view name and are resolved by a view resolver. Spring comes with quite a few view resolvers. We'll list most of them

ViewResolver	Description
AbstractCach-ingViewResolver	An abstract view resolver which takes care of caching views. Often views need prepara-tion before they can be used, extending this view resolver provides caching of views.
XmlViewResolver	An implementation of ViewResolver that accepts a configuration file written in XML with the same DTD as Spring's bean factories. The default configuration file is /WEB-INF/views.xml.
Resource-BundleViewResolver	An implementation of ViewResolver that uses bean definitions in a ResourceBundle, spec-ified by the bundle basename. The bundle is typically defined in a properties file, located in the classpath. The default file name is views.properties.
UrlBasedViewResolver	A simple implementation of ViewResolver that allows for direct resolution of symbolic view names to URLs, without an explicit mapping definition. This is appropriate if your

ViewResolver	Description
	symbolic names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings.
InternalRe-sourceViewResolver	A convenience subclass of UrlBasedViewResolver that supports InternalResourceView (i.e. Servlets and JSPs), and subclasses like JstlView and TilesView. The view class for all views generated by this resolver can be specified via setViewClass. See UrlBasedViewResolver's javadocs for details.
VelocityViewResolver / FreeMarkerViewRe-solver	A convenience subclass of UrlBasedViewResolver that supports VelocityView (i.e. Velocity templates) or FreeMarkerView respectively and custom subclasses of them.

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

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.UrlBasedViewResolver">
  <property name="prefix">
        <value>/WEB-INF/jsp/</value>
  </property>
  <property name="suffix">
        <value>.jsp</value>
  </property>
</bean>
```

When returning test as a viewname, this view resolver will hand the request over to the RequestDispatcher that will send the request to /WEB-INF/jsp/test.jsp. When mixing different view technologies in a web application, you can use the ResourceBundleViewResolver:

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
  <property name="basename" value="views" />
  <property name="defaultParentView" value="parentView" />
</bean>
```

The ResourceBundleViewResolver inspects the ResourceBundle identified by the basename, and for each view it is supposed to resolve, it uses the value of the property [viewname].class as the view class and the value of the property[viewname].url as the view url. As you can see, you can identify a parent view, from which all views in the properties file sort of extend. This way you can specify a default view class, for example A note on caching - subclasses of AbstractCachingViewResolver cache view instances they have resolved. This greatly improves performance when using certain view technology. It's possible to turn off the cache, by setting the cacheproperty to false. Furthermore, if you have the requirement to be able to refresh a certain view at runtime (for example when a Velocity template has been modified), you can use the removeFromCache (String viewName, Locale loc) method.

Chaining View Resolvers

Spring supports more than just one view resolver. This allows you to chain resolvers and, for example, override specific views in certain circumstances. Chaining view resolvers is pretty straightforward - just add more than one resolver to your application context and, if necessary, set the order property to specify an order. Remember, the higher the order property, the later the view resolver will be positioned in the chain.

In the following example, the chain of view resolvers consists of two resolvers, a InternalResourceViewResolver (which is always automatically positioned as the last resolver in the chain) and an XmlViewResolver for specifying Excel views (which are not supported by the InternalResourceViewResolver):

```
<bean id="jspViewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
  <property name="prefix" value="/WEB-INF/jsp/"/>
  <property name="suffix" value=".jsp"/>
</bean>

<bean id="excelViewResolver" class="org.springframework.web.servlet.view.XmlViewResolver">
  <property name="order" value="1"/>
  <property name="location" value="/WEB-INF/views.xml"/>
</bean>

### views.xml
<beans>
  <bean name="report" class="org.springframework.example.ReportExcelView"/>
</beans>
```

If a specific view resolver does not result in a view, Spring will inspect the context to see if other view resolvers are configured. If there are additional view resolvers, it will continue to inspect them. If not, it will throw an Exception.
You have to keep something else in mind - the contract of a view resolver mentions that a view resolver *can* return null to indicate the view could not be found. Not all view resolvers do this however! This is because in some cases, the resolver simply cannot detect whether or not the view exists. For example, the InternalResourceViewResolver uses the RequestDispatcher internally, and dispatching is the only way to figure out if a JSP exists -this can only be done once. The same holds for the VelocityViewResolver and some others. Check the JavaDoc for the view resolver to see if you're dealing with a view resolver that does not report non-existing views. As a result of this, putting anInternalResourceViewResolver in the chain in a place other than the last, will result in the chain not being fully inspected, since the InternalResourceViewResolver will *always* return a view!

Redirecting to views

As has been mentioned, a controller normally returns a logical view name, which a view resolver resolves to a particular view technology. For view tehcnologies such as JSPs that are actually processed via the Servlet/JSP engine, this is normally handled via InternalResourceViewResolver/InternalResourceView which will ultimately end up issuing an internal forward or include, via the Servlet API's RequestDispatcher.forward() or RequestDispatcher.include(). For other view technologies, such as Velocity, XSLT, etc., the view itself produces the content on the response stream.
It is sometimes desireable to issue an HTTP redirect back to the client, before the view is rendered. This is desireable for example when one controller has been called with POSTed data, and the response is actually a delegation to another controller (for example on a successful form submission). In this case, a normal internal forward will mean the other controller will also see the same POST data, whih is potentially problematic if it can confuse it with other expected data Another reason to do a redirect before displaying the result is that this will eliminate the possiblity of the user doing a double submission of form data. The browser will have sent the initial POST, will have seen a redirect back and done a subsequent GET because of that, and thus as far as it is concerned, the current page does not reflect the result of a POST, but rather of a GET, so there is no way the user can accidentally re-POST the same data by doing a refresh. The refresh would just force a GET of the result page, not a resend of the initial POST data.

RedirectView

One way to force a redirect as the result of a controller response is for the controller to create and return an instance of Spring's RedirectView. In this case, DispatcherServlet will not use the normal view resolution mechanism, but rather as it has been given the (redirect) view already, will just ask it to do it's work.
The RedirectView simply ends up issuing an HttpServletResponse.sendRedirect() call, which will come back to the client browser as an HTTP redirect. All model attributes are simply exposed as HTTP query parameters. This does mean that the model must contain only objects (generally Strings or convertible to Strings) which can be readily converted to a string-form HTTP query parameter.
If using RedirectView, and the view is created by the Controller itself, it is generally always preferable if the redirect URL at least is injected into the Controller, so that it is not baked into the controller but rather configured in the context along with view names and the like.

The redirect: prefix

While the use of RedirectView works fine, if the controller itself is creating the RedirectView, there is no getting around the fact that the controller is aware that a redirection is happening. This is really suboptimal and couples things too tightly. The controller should not really care about how the response gets handled. It should generally think only in terms of view names, that have been injected into it.

The special redirect: prefix allows this to be achived. If a view name is returned which has the prefix redirect:, thenUrlBasedViewResolver (and all subclasses) will recognize this as a special indication that a redirect is needed. The rest of the view name will be treated as the redirect URL.

The net effect is the same as if the controller had returned a RedirectView, but now the controller itself can deal just in terms of logical view names. A logical view name such as redirect:/my/response/controller.html will redirect relative to the current servlet

context, while a name such as redirect:http://myhost.com/some/arbitrary/path.htmlwill redirect to an absolute URL. The important thing is that as long is this redirect view name is injected into the controller like any other logical view name, the controller is not even aware that redirection is happening.

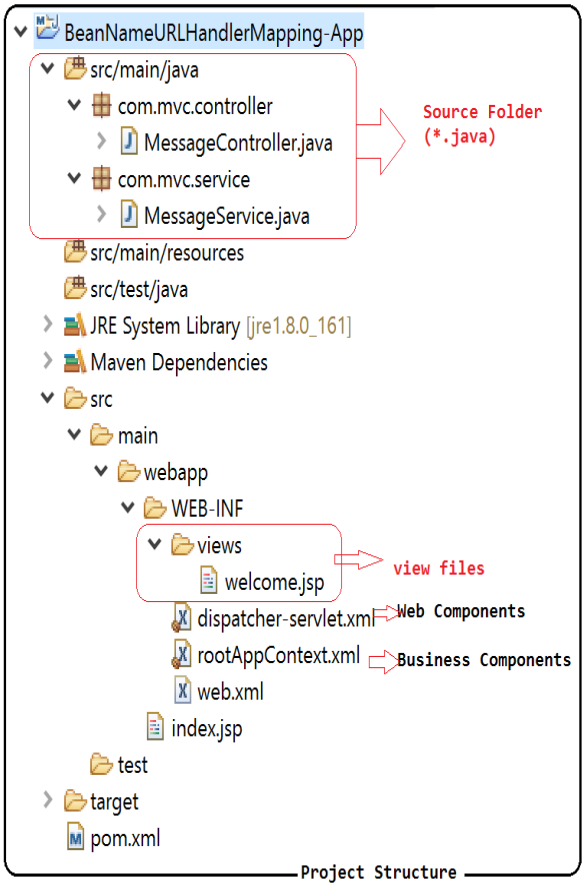
The forward: prefix

It is also possible to use a special forward: prefix for view names that will ultimately be resolved byUrlBasedViewResolver and subclasses. All this does is create an InternalResourceView (which ultimately does a RequestDispatcher.forward()) around the rest of the view name, which is considered a URL. Therefore, there is never any use in using this prefix when using InternalResourceViewResolver/InternalResourceView anyway (for JSPs for example), but it's of potential use when you are primarilly using another view technology, but want to still be able to in some cases force a forward to happen to a resource to be handled by the Servlet/JSP engine. Note that if you need to do this a lot though, you may also just chain multiple view resolvers. As with the redirect: prefix, if the view name with the prefix is just

injected into the controller, the controller does not have to be aware that anything special is happening in terms of handling the response.

BeanNameUrlHandlerMapping-Application

Step-1: Create Maven web application and configure Spring Web Mvc and java.servlet-api dependencies in pom.xml file like below



BeanNameURLHandlerMapping-App/pom.xml

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.web</groupId>
    <artifactId>BeanNameURLHandlerMapping-App</artifactId>
    <packaging>war</packaging>
    <version>0.0.1-SNAPSHOT</version>
    <name>BeanNameURLHandlerMapping-App Maven Webapp</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>4.3.6.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.0.1</version>
        </dependency>
    </dependencies>
    <build>
        <finalName>BeanNameURLHandlerMapping-App</finalName>
    </build>
</project>
```

pom.xml

Step-2: Create Controller and Service classes

MessageController.java

```java
package com.mvc.controller;

import javax.servlet.http.HttpServletRequest;

@Controller
public class MessageController extends AbstractController {

    public MessageController() {
        System.out.println("****MessageController:::Constructor****");
    }

    @Autowired(required = true)
    private MessageService msgService;

    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request, HttpServletResponse response)
            throws Exception {
        ModelAndView mav = new ModelAndView();

        mav.setViewName("welcome");

        mav.addObject("msg", msgService.getMessage());

        return mav;
    }
}
```

MessageController.java

MessageService.java

```java
package com.mvc.service;

import org.springframework.stereotype.Service;

@Service
public class MessageService {

    public MessageService() {
        System.out.println("****MessageService:::Constructor****");
    }

    public String getMessage() {
        return "Welcome to Spring MVC";
    }
}
```

MessageService.java

Step-3: Create view file to display as response to end user (here we are using jsp as view technology)

welcome.jsp

```jsp
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Spring MVC App</title>
</head>
<body>
    <h1>${msg}</h1>
</body>
</html>
```

welcome.jsp

Step-4: Create rootApplicationContext configuration file to configure Business components

rootAppContext.xml

```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:aop="http://www.springframework.org/schema/aop"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <!-- Declaring Business components -->
    <bean id="msgService" class="com.mvc.service.MessageService" />

</beans>
```

rootAppContext.xml

Step-5: Create dispatcher-servlet.xml to declare Web components

dispatcher-servlet.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

    <!-- To Enable Annotations -->
    <mvc:annotation-driven />

    <!-- To Scan Component classes -->
    <context:component-scan base-package="com.mvc.*" />

    <bean name="/welcome.htm" class="com.mvc.controller.MessageController" />

    <!-- View Resolver -->
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>

</beans>
```

dispatcher.servlet.xml

Step-6: Configure DispatcherServlet & rootAppContext file in web.xml file like below

web.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    id="WebApp_ID" version="3.1">
    <display-name>SpringMVCApp1</display-name>

    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/rootAppContext.xml</param-value>
    </context-param>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.htm</url-pattern>
    </servlet-mapping>

</web-app>
```

web.xml

Step-7: Deploy the application in JEE server

Once project deployed into Server below operations will be performed

- Server will try to load web.xml file from project WEB-INF folder.
- If web.xml is available, server checks for well-formness and validness of xml
- If xml is well-formed and valid, server will use jax-p api to parse web.xml content in xml style
- Server will create ServletContext object and will trigger contextCreated event
- If ContextLoaderListener is configured, then it will start IOC using ApplicationContext by passing context param value as input. This IoC will act as rootAppliationContext.

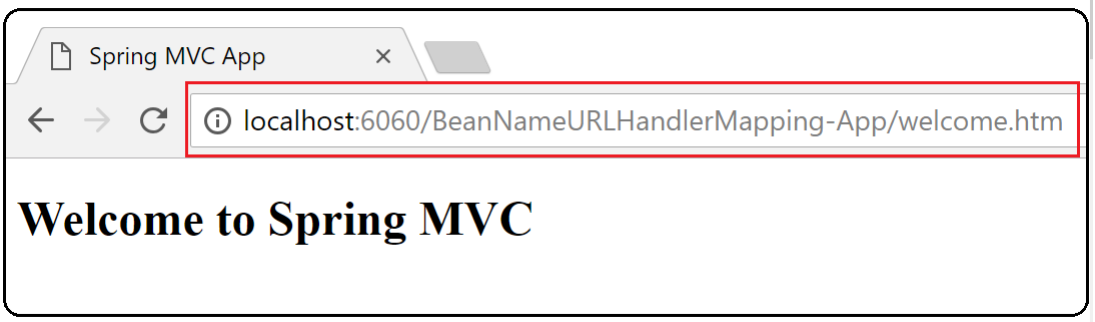
Note: context-param name should be contextConfigLocation and value will be path of the rootAppliationContext config file

- If the given file is available, IOC will load all business components configured in given bean spring bean configuration file and will instantiate all singleton beans.
- Then Server will check for Servlets which are having load-on-startup, if available container will instantiate that servlet and it will call init method and service method of that servlet.
- Note : In Spring application we will configure DispatcherServlet with load-on-startup to create WebApplicationContext before receiving first request itself
- DispatcherServlet init method will start WebApplicationContext by passing disparcher-servlet.xml file as input.
- WebApplicationContext will read web components configured in Bean Configuration and will instantiate all singleton beans.

Note: If no handler mapper configured, then DispatcherServlet will use BeanNameUrlHandlerMapper as a default handler mapper.

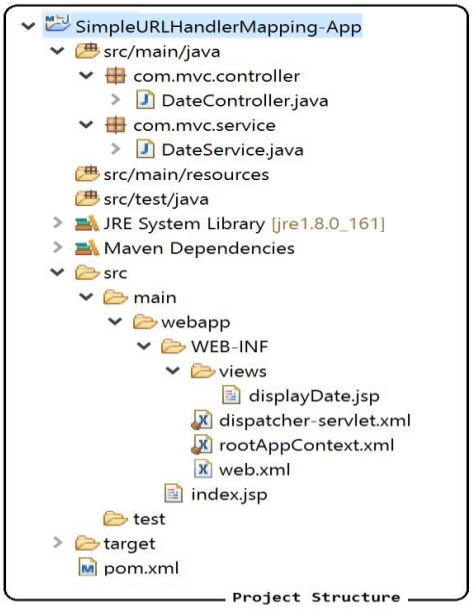
Once Deployment process is completed, RootAppliationContext and WebApplicationContext specific beans objects will be created and they will be ready to use.

Step-8: Access the application using below URL – welcome message will be displayed like below



SimpleUrlHandlerMapping-Application

Step-1: Create Maven web application and configure Spring Web Mvc and java.servlet-api dependencies in pom.xml file like below




```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>4.3.6.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.0.1</version>
    </dependency>
</dependencies>
```

pom.xml

Step-2: Create Controller and Service classes

DateController.java

```
package com.mvc.controller;

import javax.servlet.http.HttpServletRequest;

public class DateController extends AbstractController {

    @Autowired(required = true)
    private DateService dateService;

    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest req,
            HttpServletResponse res) throws Exception {
        ModelAndView mav = new ModelAndView();
        mav.setViewName("displayDate");
        mav.addObject("date", dateService.getTodaysDate());
        return mav;
    }

}
```

DateController.java

DateService.java

```
package com.mvc.service;

import java.util.Date;

@Service
public class DateService {

    public Date getTodaysDate() {
        return new Date();
    }
}
```

DateServie.java

Step-3: Create view file to display as response to end user (here we are using jsp as view technology)

displayDate.jsp

```jsp
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Simple URL Handler Mapping App</title>
</head>
<body>
    <h1>Today's Date is : ${date}</h1>
</body>
</html>
```

displayDate.jsp

Step-4: Create rootAppliationContext & WebApplicationContext configuration file to configure Business components & web Components

```xml
    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="/date.htm">dateController</prop>
            </props>
        </property>
    </bean>

    <bean id="dateController" class="com.mvc.controller.DateController" />


    <!-- View Resolver -->
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>
```

dispatcher-servlet.xml

```xml

    <!-- Declaring Business components -->
    <bean id="msgService" class="com.mvc.service.DateService" />

```

rootAppContext.xml

Step-6: Configure DispatcherServlet & rootAppContext file in web.xml file like below

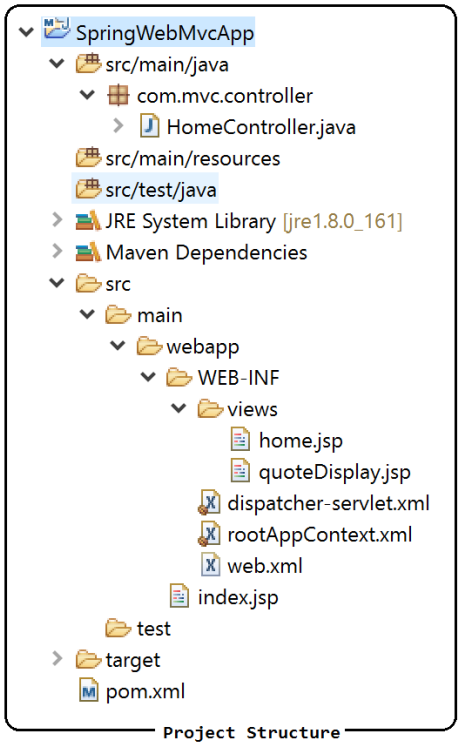
```xml
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/rootAppContext.xml</param-value>
    </context-param>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.htm</url-pattern>
    </servlet-mapping>
```

web.xml

Step-7: Deploy the application in JEE server and access Controller using below URL.

Spring Web MVC Application using @Controller and @RequestMapping

Step-1: Create Maven web project and configure maven dependencies in project pom.xml file



Project Structure

```
12  <dependencies>
13      <dependency>
14          <groupId>junit</groupId>
15          <artifactId>junit</artifactId>
16          <version>3.8.1</version>
17      </dependency>
18      <dependency>
19          <groupId>org.springframework</groupId>
20          <artifactId>spring-webmvc</artifactId>
21          <version>4.3.6.RELEASE</version>
22      </dependency>
23      <dependency>
24          <groupId>javax.servlet</groupId>
25          <artifactId>javax.servlet-api</artifactId>
26          <version>3.0.1</version>
27      </dependency>
28  </dependencies>
```

Maven Dependencies

Step-2: Create controller class using @Controller annotations (This class acts as Request Handler for client requests). Map request to controller method using @RequestMapping annotation like below

HomeController.java

```java
package com.mvc.controller;

import java.text.DateFormat;

@Controller
public class HomeController {

    @RequestMapping(value = "/home.htm", method = RequestMethod.GET)
    public String home(Locale locale, Model model) {

        Date date = new Date();
        DateFormat dateFormat = DateFormat.getDateTimeInstance(DateFormat.LONG, DateFormat.LONG, locale);

        String formattedDate = dateFormat.format(date);

        model.addAttribute("serverTime", formattedDate);
        model.addAttribute("msg", "Welcome to Ashok IT School..!!");

        return "home";
    }

    @RequestMapping(value = "/getDailyQuote.htm", method = RequestMethod.GET)
    public String getDailyQuote(Model model) {
        model.addAttribute("quote", "Don't Drink and Drive..!!");
        return "quoteDisplay";
    }

}
```

HomeController.java

Step-3: Create view files (Using JSP as view Technology)

home.jsp

```jsp
<%@ page session="false"%>
<html>
<head>
<title>Home</title>
</head>
<body>
    <h1>${msg}</h1>
    <P>The time on the server is ${serverTime}.</P>
</body>
</html>
```

home.jsp

quoteDisplay.jsp

```jsp
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Spring MVC App</title>
</head>
<body>
    <h1>Today's Quote : ${quote}</h1>
</body>
</html>
```

quoteDisplay.jsp

Step-4: Configure Web components in WebApplicationContext configuration file

dispatcher-servlet.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

    <mvc:annotation-driven />

    <context:component-scan base-package="com.mvc.controller" />

    <!-- View Resolver -->
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>

</beans>
```

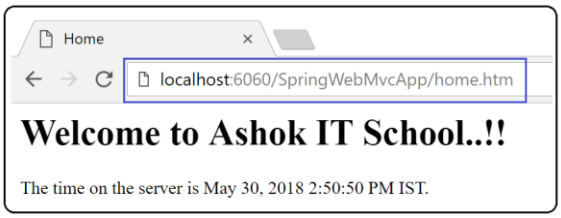
dispatcher-servlet.xml

Step-5: Configuring DispatcherServlet and rootAppliationContext in web.xml file

```xml
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/rootAppContext.xml</param-value>
    </context-param>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.htm</url-pattern>
    </servlet-mapping>
```

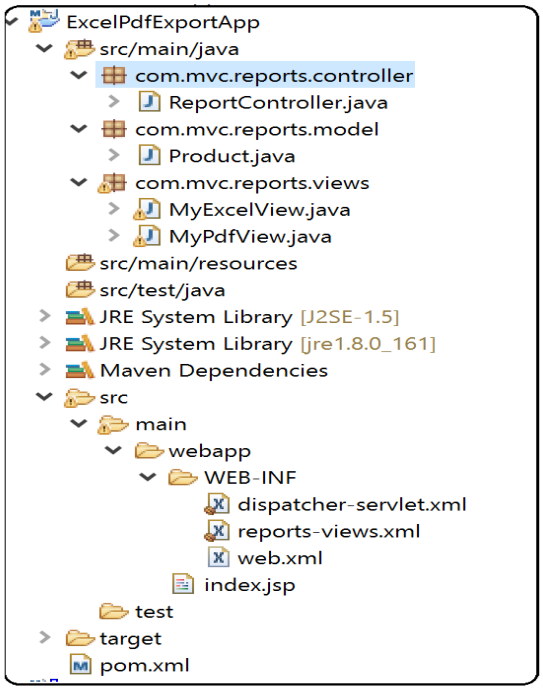
web.xml

Step-6: Deploy the application in Run server and access the controller method using http request



Working with Excel and PDF views

Step-1: Create Maven project and add poi and itext pdf dependencies like below




```xml
<dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.0.1</version>
</dependency>

<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>4.3.6.RELEASE</version>
</dependency>

<dependency>
        <groupId>org.apache.poi</groupId>
        <artifactId>poi</artifactId>
        <version>3.17</version>
</dependency>

<dependency>
        <groupId>com.lowagie</groupId>
        <artifactId>itext</artifactId>
        <version>1.4.8</version>
</dependency>
```

pom.xml

Step-2: Create Model class to present the data

```
package com.mvc.reports.model;

public class Product {

	private Integer pid;
	private String pname;

	public Product(int pid, String name) {
		this.pid = pid;
		this.pname = name;
	}

	//setters & getters
	//toString()
}
```

Product.java

```
package com.mvc.reports.controller;

import java.util.ArrayList;
import java.util.List;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

import com.mvc.reports.model.Product;

@Controller
public class ReportController {

	@RequestMapping("/excel")
	public ModelAndView generateExcel() {
		List<Product> pList = getProductsData();
		return new ModelAndView("excelView", "products", pList);
	}

	@RequestMapping("/pdf")
	public ModelAndView generatePdf() {
		List<Product> pList = getProductsData();
		return new ModelAndView("pdfView", "products", pList);
	}

	private List<Product> getProductsData() {
		List<Product> pList = new ArrayList<Product>();
		pList.add(new Product(501, "Keyboard"));
		pList.add(new Product(502, "Mouse"));
		pList.add(new Product(503, "Hard disk"));
		return pList;
	}

}
```

ReportController.java

```java
package com.mvc.reports.views;
public class MyExcelView extends AbstractXlsView {

	@Override
	protected void buildExcelDocument(Map<String, Object> map, Workbook book,
HttpServletRequest req,
			HttpServletResponse res) throws Exception {

		Sheet sheet = book.createSheet("Products Details");

		Row headerRow = sheet.createRow(0);

		headerRow.createCell(0).setCellValue("S.No");
		headerRow.createCell(1).setCellValue("PID");
		headerRow.createCell(2).setCellValue("PName");

		List<Product> pList = (List) map.get("products");

		if (!pList.isEmpty()) {
			int rowIndex = 1;
			for (Product p : pList) {
				Row dataRow = sheet.createRow(rowIndex);
				dataRow.createCell(0).setCellValue(rowIndex);
				dataRow.createCell(1).setCellValue(p.getPid());
				dataRow.createCell(2).setCellValue(p.getPname());
				rowIndex++;
			}
		}
	}
}
```

MyExcelView.java

```java
package com.mvc.reports.views;

public class MyPdfView extends AbstractPdfView {
	@Override
	protected void buildPdfDocument(Map map, Document doc, PdfWriter writer, HttpServletRequest request,
			HttpServletResponse response) throws Exception {

		List<Product> pList = (List) map.get("products");

		Paragraph p = new Paragraph("Product Details");
		p.setAlignment("center");

		Table t = new Table(3);
		t.setAlignment("center");

		t.addCell("S.No");
		t.addCell("Product ID");
		t.addCell("Product Name");

		if (!pList.isEmpty()) {
			int rowIndex = 1;
			for (Product prod : pList) {
				t.addCell(rowIndex + "");
				t.addCell(prod.getPid() + " ");
				t.addCell(prod.getPname() + " ");
				rowIndex++;
			}
		}
		doc.add(p);
		doc.add(t);
	}
}
```

MyPdfView.java

```xml
<mvc:annotation-driven />

<context:component-scan base-package="com.mvc.reports.controller" />

<bean class="org.springframework.web.servlet.view.XmlViewResolver">
	<property name="location" value="/WEB-INF/reports-views.xml" />
	<property name="order" value="0" />
</bean>

<bean id="viewResolver"
	class="org.springframework.web.servlet.view.InternalResourceViewResolver">
	<property name="prefix" value="/WEB-INF/views/" />
	<property name="suffix" value=".jsp" />
	<property name="order" value="1" />
</bean>
```

dispatcher-Servlet.xml

```xml
<bean id="excelView" class="com.mvc.reports.views.MyExcelView" />
<bean id="pdfView" class="com.mvc.reports.views.MyPdfView" />
```

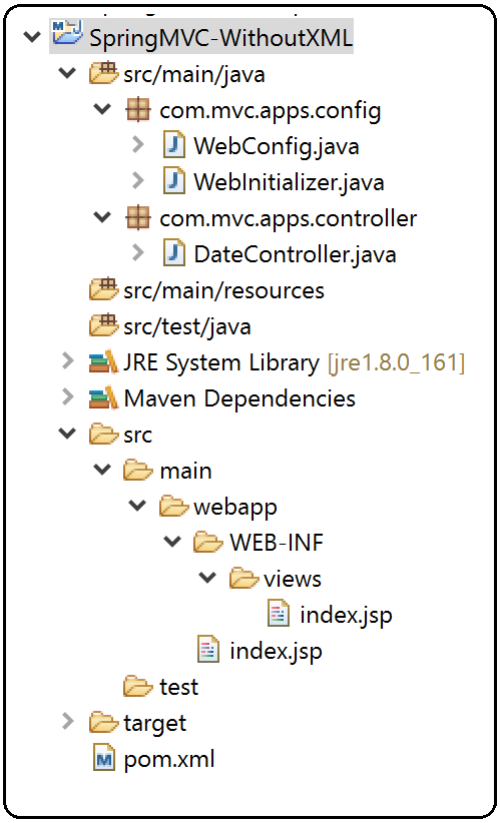
reports-views.xml

```jsp
<html>
<body>
	<a href="excel">Excel</a>
	<br />

	<a href="pdf">PDF</a>
</body>
</html>
```

index.jsp

Spring MVC Application with zero XML

Step-1: Create a Maven web application with below dependencies in Pom.xml for the Required Libraries




```
<properties>
        <failOnMissingWebXml>false</failOnMissingWebXml>
</properties>

<dependencies>
        <dependency>
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
                <version>3.8.1</version>
        </dependency>

        <dependency>
                <groupId>javax.servlet</groupId>
                <artifactId>javax.servlet-api</artifactId>
                <version>3.0.1</version>
        </dependency>

        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-webmvc</artifactId>
                <version>4.3.6.RELEASE</version>
        </dependency>
</dependencies>
```

pom.xml

This Property needs to be configured in the absence of web.xml file

Step-2: Creating a WebConfig Class

As we want to do Java-based configuration, we will create a class called SpringConfig, where we will register all Spring-related beans using Spring's Java-based configuration style.

This class will replace the need to create a SpringApplicationContext.xml file, where we use two important tags

- <context:component-scan/>
- <mvc:annotation-driven/>

* Please note that this class has to extend the org.springframework.web.servlet.config.annotation.WebMvc-ConfigurerAdapter class

* Please note that here, we will use three different annotations at the top level. They will serve the purpose of the XML-based tags used earlier.

XML Tag	Annotation	Description
<context:component-scan/>	@ComponentScan()	Scan starts from base package and registers all controllers, repositories, service, beans, etc.
<mvc:annotation-driven/>	@EnableWebMvc	Enable Spring MVC-specific annotations like @Controller
Spring config file	@Configuration	Treat as the configuration file for Spring MVC-enabled applications.

```
package com.mvc.apps.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = { "com.mvc.apps.controller" })
public class WebConfig {

	@Bean
	public InternalResourceViewResolver createViewResolver() {
		InternalResourceViewResolver vr = new InternalResourceViewResolver();
		vr.setPrefix("/WEB-INF/views/");
		vr.setSuffix(".jsp");
		return vr;
	}
}
```

WebConfig.java

Step 3: Replacing Web.xml

Create another class, which will replace our traditional web.xml. We use Servlet 3.0 and extend the org.springframework.web.WebApplicationInitializer class.

Here we provide our SpringConfig class and add DispatcherServlet, which acts as the FrontController of the Spring MVC application. SpringConfig class is the source of Spring beans, before which we used contextConfiglocation.

```
package com.mvc.apps.config;

import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class WebInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {
	@Override
	protected Class<?>[] getRootConfigClasses() {
		return null;
	}

	@Override
	protected Class<?>[] getServletConfigClasses() {
		return new Class[] { WebConfig.class };
	}

	@Override
	protected String[] getServletMappings() {
		return new String[] { "/" };
	}
}
```

WebInitializer.java

Step 4: Create a Controller Class

Now we will create a Controller class, which will take a parameter from request URL and greet a message in the browser.

```
package com.mvc.apps.controller;

import java.util.Date;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HelloController {

	@RequestMapping("/hello")
	public String display(Model model) {
		model.addAttribute("msg", "Hello, This is produced by HelloController..!!");
		return "index";
	}
}
```

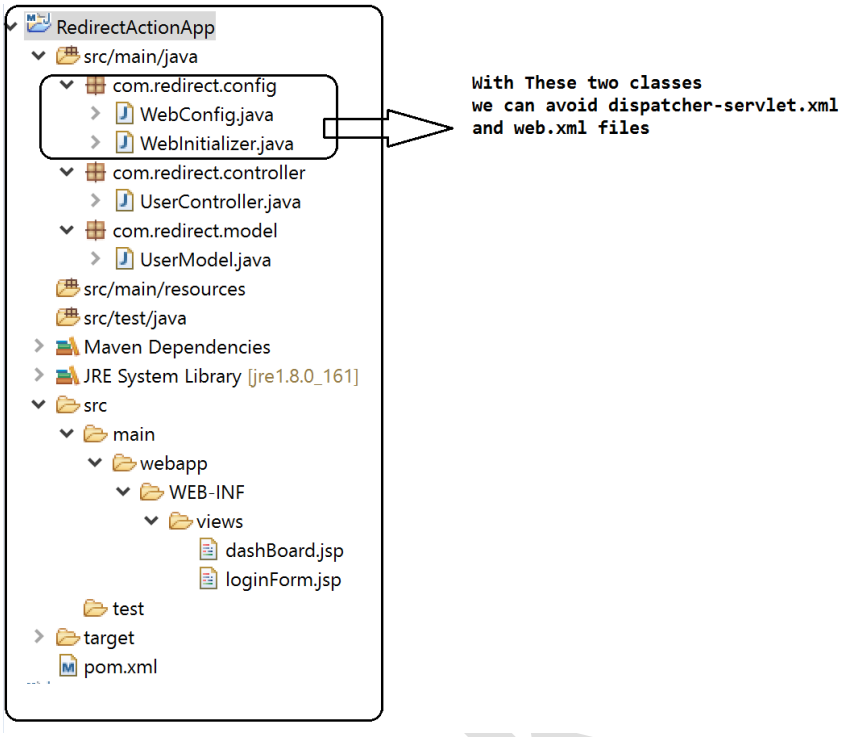
HelloController.java

Step 5: Create a JSP Page to Show the Message

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
	pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Spring App</title>
</head>
<body>
	Response : ${msg}
</body>
</html>
```

index.jsp

LogingForm with Redirection – App




```
package com.redirect.config;


@Configuration
@EnableWebMvc
@ComponentScan(basePackages = { "com.redirect.controller" })
public class WebConfig {

	@Bean
	public InternalResourceViewResolver createViewResolver() {
		InternalResourceViewResolver vr = new InternalResourceViewResolver();
		vr.setPrefix("/WEB-INF/views/");
		vr.setSuffix(".jsp");
		return vr;
	}
}
```

```
import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class WebInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {
	@Override
	protected Class<?>[] getRootConfigClasses() {
		return null;
	}

	@Override
	protected Class<?>[] getServletConfigClasses() {
		return new Class[] { WebConfig.class };
	}

	@Override
	protected String[] getServletMappings() {
		return new String[] { "/" };
	}
}
```

```java
public class UserModel {

	private String uname;
	private String pwd;

	//setters & getters
}
```

```java
package com.redirect.controller;

@Controller
public class UserController {

	@RequestMapping("/loginForm")
	public String displayLoginForm(Model model) {
		model.addAttribute("u", new UserModel());
		return "loginForm";
	}

	@RequestMapping(value = "/login", method = RequestMethod.POST)
	public String login(@ModelAttribute UserModel um, Model model) {
		if (um.getUname().equals("admin") && um.getPwd().equals("admin123")) {
			// Login successfull
			return "redirect:/buildDashboard";
		} else {
			// login failure
			model.addAttribute("u", new UserModel());
			model.addAttribute("errMsg", "Invalid Credentials");
			return "loginForm";
		}
	}

	@RequestMapping("/buildDashboard")
	public String dashboard(Model model) {
		model.addAttribute("msg", "Reports generating...!!");
		return "dashBoard";
	}
}
```

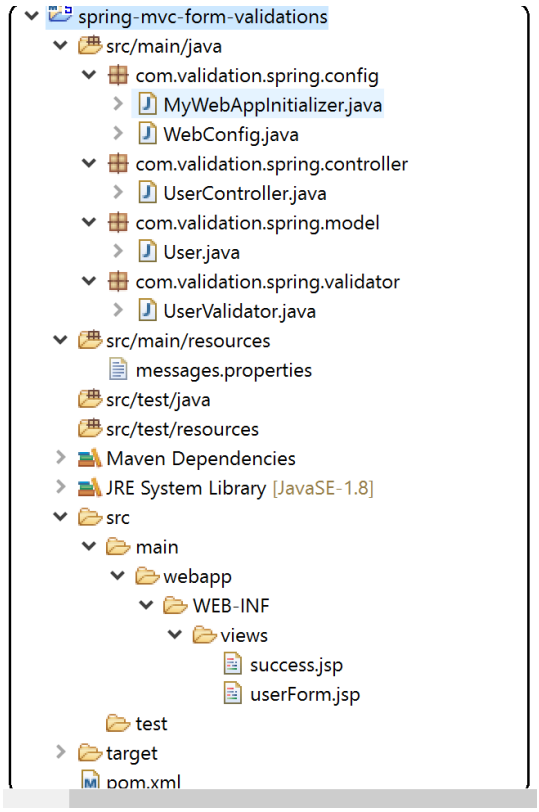
UserController.java

```jsp
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
	<h2>Login Here</h2>
	<b style="color: red">${errMsg}</b>
	<form:form action="login" method="post" modelAttribute="u">
		<table>
			<tr>
				<td>Username :</td>
				<td><form:input path="uname" /></td>
			</tr>
			<tr>
				<td>Password :</td>
				<td><form:password path="pwd" /></td>
			</tr>
			<tr>
				<td></td>
				<td><input type="submit" value="Login" /></td>
			</tr>
		</table>
	</form:form>
</body>
</html>
```

login.jsp

Spring MVC Application with Form validations

Step-1: Create a Maven web application with below dependencies in pom.xml for the Required Libraries



Step-2: Creating a WebConfig Class

WebConfig.java

```java
package com.validation.spring.config;

import org.springframework.context.MessageSource;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = { "com.validation.spring.controller", "com.validation.spring.validator" })
public class WebConfig extends WebMvcConfigurerAdapter {

    @Bean
    public InternalResourceViewResolver resolver() {
        InternalResourceViewResolver resolver = new InternalResourceViewResolver();
        resolver.setViewClass(JstlView.class);
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        return resolver;
    }

    @Bean
    public MessageSource messageSource() {
        ResourceBundleMessageSource source = new ResourceBundleMessageSource();
        source.setBasename("messages");
        return source;
    }
}
```

Step 3: Replacing Web.xml

MyWebAppInitializer.java

```java
package com.validation.spring.config;

import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[] {};
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] { WebConfig.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

Step-4 : Create messages.properties file for validation messages

messages.properties

```
user.name.empty = Enter a valid name.
user.email.empty = Enter a valid email.
user.email.invalid = Invalid email! Please enter valid email.
user.gender.empty = Select gender.
user.languages.empty = Select at least one language.
```

Step-5: Create a Model class for holding form data

User.java

```java
package com.validation.spring.model;

import java.util.List;

public class User {
    private String name;
    private String email;
    private String gender;
    private List<String> languages;

    // setters & getters

}
```

Step-6: Create Validator class by implementing Validator interface

UserValidator.java

```java
package com.validation.spring.validator;

import java.util.regex.Pattern;

@Component
public class UserValidator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return User.class.equals(clazz);
    }

    @Override
    public void validate(Object obj, Errors err) {

        ValidationUtils.rejectIfEmpty(err, "name", "user.name.empty");
        ValidationUtils.rejectIfEmpty(err, "email", "user.email.empty");
        ValidationUtils.rejectIfEmpty(err, "gender", "user.gender.empty");
        ValidationUtils.rejectIfEmpty(err, "languages", "user.languages.empty");

        User user = (User) obj;

        Pattern pattern = Pattern.compile("^[A-Z0-9._%+-]+@[A-Z0-9.-]+\\.[A-Z]{2,6}$", Pattern.CASE_INSENSITIVE);
        if (!(pattern.matcher(user.getEmail()).matches())) {
            err.rejectValue("email", "user.email.invalid");
        }
    }
}
```

Step 7: Create a Controller Class

UserController.java

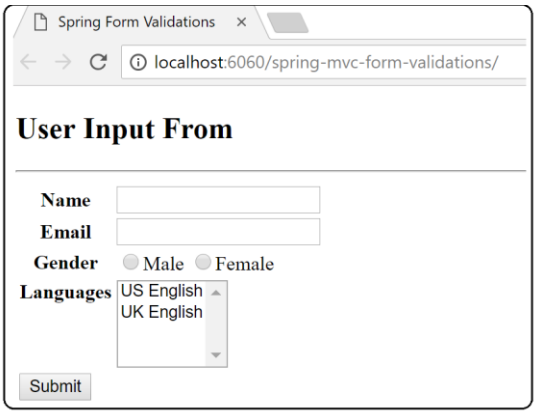
```java
package com.validation.spring.controller;

import java.util.Locale;

@Controller
public class UserController {

    @Autowired
    private UserValidator userValidator;

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.addValidators(userValidator);
    }

    @GetMapping("/")
    public String userForm(Locale locale, Model model) {
        model.addAttribute("user", new User());
        return "userForm";
    }

    /*
     * Save user object
     */
    @PostMapping("/saveUser")
    public String saveUser(@ModelAttribute("user") @Validated User user, BindingResult result, Model model) {
        if (result.hasErrors()) {
            return "userForm";
        }
        return "success";
    }
}
```

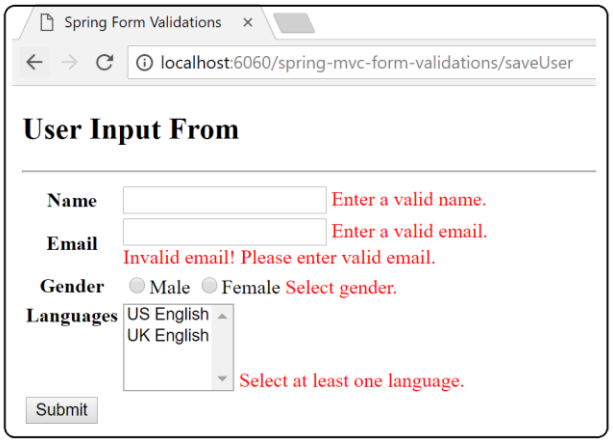
Step-8: Create jsp form for user input

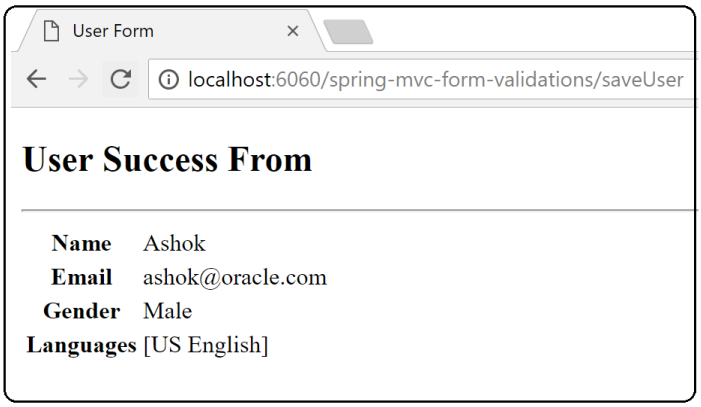
```
<style type="text/css">
.error {
        color: red;
}
</style>
</head>
<body>
        <h2>User Input From</h2>
        <hr />
        <form:form action="saveUser" method="post" modelAttribute="user">
                <table>
                        <tr>
                                <th>Name</th>
                                <td><form:input path="name" /> <form:errors path="name" cssClass="error" /></td>
                        </tr>
                        <tr>
                                <th>Email</th>
                                <td><form:input path="email" /> <form:errors path="email" cssClass="error" /></td>
                        </tr>
                        <tr>
                                <th>Gender</th>
                                <td><form:radiobutton path="gender" value="Male" label="Male" />
                                        <form:radiobutton path="gender" value="Female" label="Female" /> <form:errors
                                                path="gender" cssClass="error" /></td>
                        </tr>
                <tr>
                        <th valign="top">Languages</th>
                        <td><form:select path="languages" multiple="true">
                                        <form:option value="US English">US English</form:option>
                                        <form:option value="UK English">UK English</form:option>
                            </form:select>
                            <form:errors path="languages" cssClass="error" />
                        </td>
                </tr>
                <tr>
                        <td><button type="submit">Submit</button></td>
                </tr>
        </table>
</form:form>
```

userForm.jsp

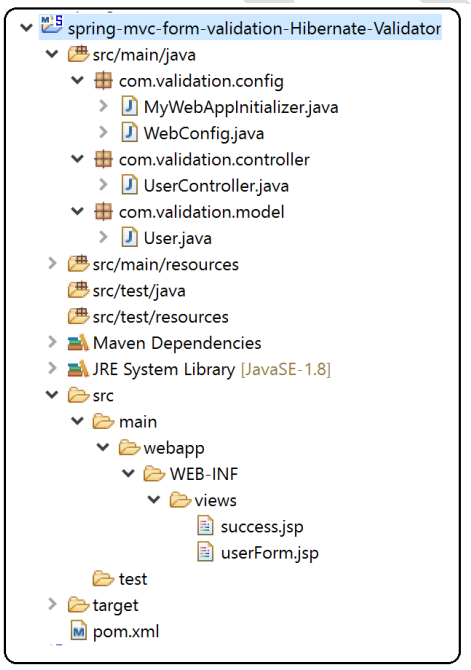
Step-9: Deploy the application and test it







Form Validations using Hibernate Validator – Application



```
<properties>
	<failOnMissingWebXml>false</failOnMissingWebXml>
</properties>
<dependencies>
	<!-- Spring MVC Dependency -->
	<dependency>
		<groupId>org.springframework</groupId>
		<artifactId>spring-webmvc</artifactId>
		<version>4.3.7.RELEASE</version>
	</dependency>

	<!-- Hibernate Validator -->
	<dependency>
		<groupId>org.hibernate</groupId>
		<artifactId>hibernate-validator</artifactId>
		<version>5.4.1.Final</version>
	</dependency>

	<!-- JSTL Dependency -->
	<dependency>
		<groupId>javax.servlet.jsp.jstl</groupId>
		<artifactId>javax.servlet.jsp.jstl-api</artifactId>
		<version>1.2.1</version>
	</dependency>

	<dependency>
		<groupId>taglibs</groupId>
		<artifactId>standard</artifactId>
		<version>1.1.2</version>
	</dependency>

	<!-- Servlet Dependency -->
	<dependency>
		<groupId>javax.servlet</groupId>
		<artifactId>javax.servlet-api</artifactId>
		<version>3.1.0</version>
		<scope>provided</scope>
	</dependency>

	<!-- JSP Dependency -->
	<dependency>
		<groupId>javax.servlet.jsp</groupId>
		<artifactId>javax.servlet.jsp-api</artifactId>
		<version>2.3.1</version>
		<scope>provided</scope>
	</dependency>

</dependencies>
```

pom.xml

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = { "com.validation.controller" })
public class WebConfig extends WebMvcConfigurerAdapter {

	@Bean
	public InternalResourceViewResolver resolver() {
		InternalResourceViewResolver resolver = new InternalResourceViewResolver();
		resolver.setViewClass(JstlView.class);
		resolver.setPrefix("/WEB-INF/views/");
		resolver.setSuffix(".jsp");
		return resolver;
	}

	@Bean
	public MessageSource messageSource() {
		ResourceBundleMessageSource source = new ResourceBundleMessageSource();
		source.setBasename("messages");
		return source;
	}

	@Override
	public Validator getValidator() {
		LocalValidatorFactoryBean validator = new LocalValidatorFactoryBean();
		validator.setValidationMessageSource(messageSource());
		return validator;
	}

}
```

WebConfig.java

```
package com.validation.config;

import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

	@Override
	protected Class<?>[] getRootConfigClasses() {
		return new Class[] {};
	}

	@Override
	protected Class<?>[] getServletConfigClasses() {
		return new Class[] { WebConfig.class };
	}

	@Override
	protected String[] getServletMappings() {
		return new String[] { "/" };
	}
}
```

MyWebAppInitializer.java

```
user.name.empty = Name entered is invalid. It must be
between {2} and {1} characters.
user.email.invalid = Invalid email! Please enter valid email.
user.gender.empty = Select gender.
user.languages.empty = Select at least one language.
```

messages.properties

```
public class User {

	@Size(max = 20, min = 3, message = "{user.name.empty}")
	private String name;

	@Email(message = "{user.email.invalid}")
	private String email;

	@NotEmpty(message = "{user.gender.empty}")
	private String gender;

	@NotEmpty(message = "{user.languages.empty}")
	private List<String> languages;

	//setters & getters
}
```

User.java

```
@Controller
public class UserController {

	@GetMapping("/")
	public String userForm(Locale locale, Model model) {
		model.addAttribute("user", new User());
		return "userForm";
	}

	@PostMapping("/saveUser")
	public String saveUser(@ModelAttribute("user") @Valid User user, BindingResult result, Model model) {

		if (result.hasErrors()) {
			return "userForm";
		}

		return "success";
	}
}
```

UserController.java

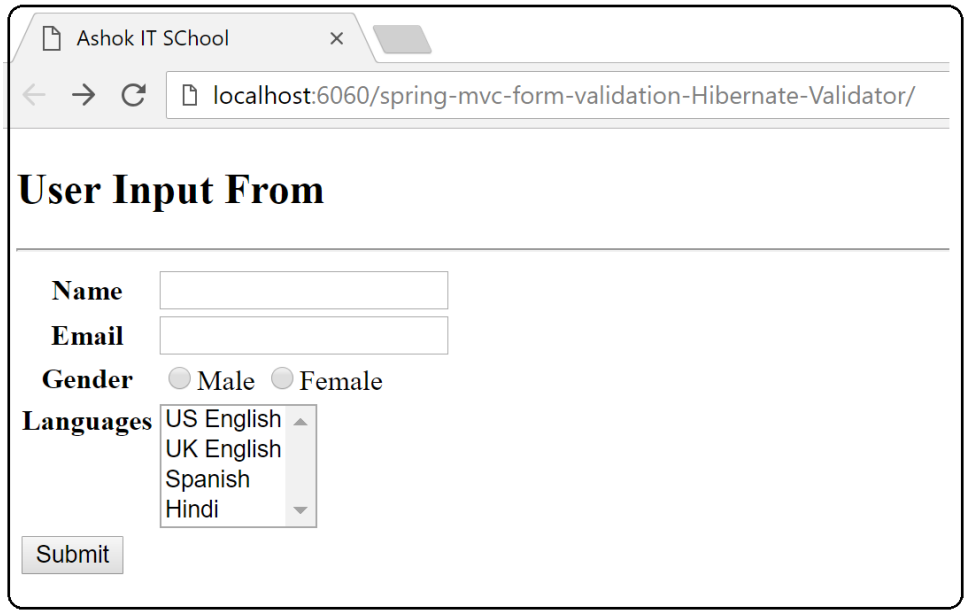
```
<body>
	<h2>User Input From</h2>
	<hr />
	<form:form action="saveUser" method="post" modelAttribute="user">
		<table>
			<tr>
				<th>Name</th>
				<td><form:input path="name" /> <form:errors path="name" cssClass="error" /></td>
			</tr>
			<tr>
				<th>Email</th>
				<td><form:input path="email" /> <form:errors path="email"  cssClass="error" /></td>
			</tr>
			<tr>
				<th>Gender</th>
				<td><form:radiobutton path="gender" value="Male" label="Male" />
					<form:radiobutton path="gender" value="Female" label="Female" />
					<form:errors   path="gender" cssClass="error" />
				</td>
			</tr>

		</table>
	</form:form>
</body>
```

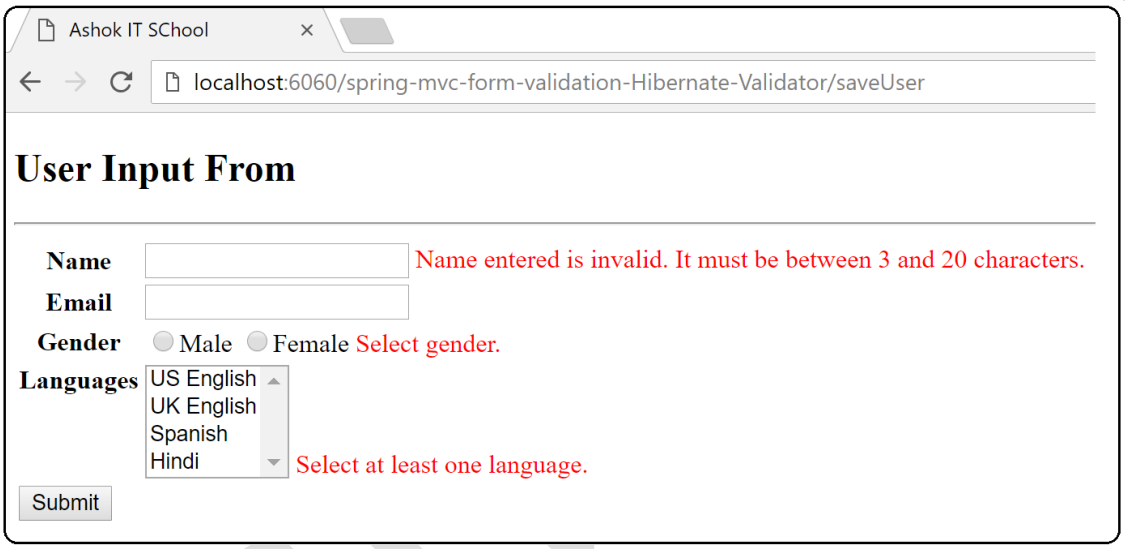
userForm.jsp

```
<tr>
	<th valign="top">Languages</th>
	<td><form:select path="languages" multiple="true">
			<form:option value="US English">US English</form:option>
			<form:option value="UK English">UK English</form:option>
			<form:option value="Spanish">Spanish</form:option>
			<form:option value="Hindi">Hindi</form:option>
	</form:select> <form:errors path="languages" cssClass="error" />
	</td>
</tr>
<tr>
	<td><button type="submit">Submit</button></td>
</tr>
```

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Ashok IT School</title>
</head>
<body>
 <h2>User Success From</h2>
 <hr />

 <table>
  <tr>
   <th>Name</th>
   <td>${user.name}</td>
  </tr>
  <tr>
   <th>Email</th>
   <td>${user.email}</td>
  </tr>
  <tr>
   <th>Gender</th>
   <td>${user.gender}</td>
  </tr>
  <tr>
   <th valign="top">Languages</th>
   <td>${user.languages}</td>
  </tr>
 </table>
</body>
</html>
```

success.jsp

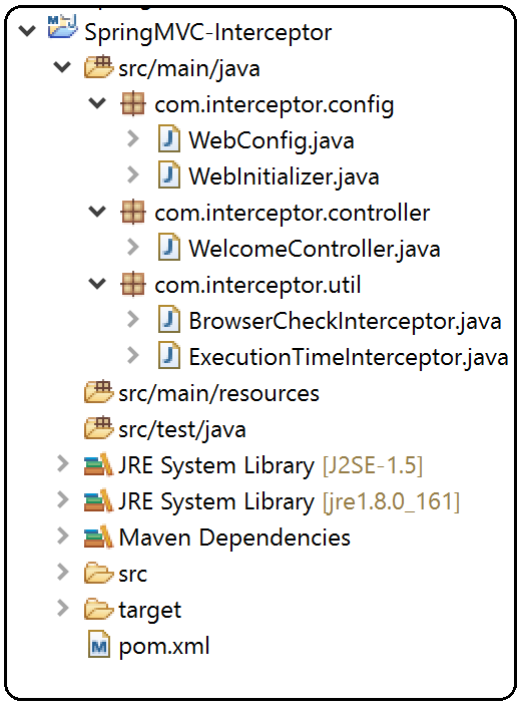




Adding HandlerInterceptors

Spring's handler mapping mechanism has a notion of handler interceptors, that can be extremely useful when you want to apply specific functionality to certain requests, for example, checking for a principal.
Interceptors located in the handler mapping must implement HandlerInterceptor from the org.springframework.web.servlet package. This interface defines three methods, one that will be called *before* the actual handler will be executed, one that will be called *after* the handler is executed, and one that is called *after the complete request has finished*. These three methods should provide enough flexibility to do all kinds of pre- and post-processing.

The preHandle method returns a boolean value. You can use this method to break or continue the processing of the execution chain. When this method returns true, the handler execution chain will continue, when it returns false, the DispatcherServlet assumes the interceptor itself has taken care of requests (and, for example, rendered an appropriate view) and does not continue executing the other interceptors and the actual handler in the execution chain.



Note: Below is the Web Configuration class (added Interceptors to Interceptor Registry)

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = { "com.interceptor.controller" })
public class WebConfig extends WebMvcConfigurerAdapter {

	@Bean
	public ViewResolver createViewResolver() {
		InternalResourceViewResolver res = new InternalResourceViewResolver();
		res.setPrefix("/WEB-INF/views/");
		res.setSuffix(".jsp");
		return res;
	}

	@Override
	public void addInterceptors(InterceptorRegistry registry) {
		registry.addInterceptor(new ExecutionTimeInterceptor());
		registry.addInterceptor(new BrowserCheckInterceptor());
	}
}
```

WebConfig.java

Note: Below is the Initializer class (This class is replacement for web.xml)

```
public class WebInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

	@Override
	protected Class<?>[] getRootConfigClasses() {
		return null;
	}

	@Override
	protected Class<?>[] getServletConfigClasses() {
		return new Class[] { WebConfig.class };
	}

	@Override
	protected String[] getServletMappings() {
		return new String[] { "/" };
	}
}
```

WebInitializer.java

Note: Below is the Browser check Interceptor, if request comes from Google Chrome it will not process the request

```
public class BrowserCheckInterceptor extends HandlerInterceptorAdapter {

	@Override
	public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
			throws Exception {
		String browserName = request.getHeader("user-agent");

		if (browserName.contains("Chrome")) {
			response.sendRedirect("invalidBrowser.jsp");
			return false;
		}
		return true;
	}
}
```

BrowserCheckInterceptor.java

Note: Below interceptor to print time taken to process each request

```
public class ExecutionTimeInterceptor extends HandlerInterceptorAdapter {

	@Override
	public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
			throws Exception {
		long startTime = System.currentTimeMillis();
		request.setAttribute("start", startTime);
		return true;
	}

	@Override
	public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
			ModelAndView modelAndView) throws Exception {
		long endTime = System.currentTimeMillis();
		long startTime = (Long) request.getAttribute("start");
		long diff = endTime - startTime;
		System.out.println("Time taken : " + diff);
	}

	@Override
	public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)
			throws Exception {
		System.out.println("after completion");
	}
}
```

ExecutionTimeInterceptor.java

```
@Controller
public class WelcomeController {

	@RequestMapping("/welcome")
	public String welcome(Model model) {
		model.addAttribute("msg", "Welcome to My Project...!!");
		return "welcomeFile";
	}

	@RequestMapping("/wish")
	public String wish(Model model) {
		model.addAttribute("msg", "Good Morning...!!");
		for (int i = 0; i <= 100; i++) {
			// TODO:
		}
		return "wishFile";
	}
}
```
WelcomeController.java

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
	pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
	<h1>${msg}</h1>
</body>
</html>
```
welcomeFile.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
	pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
	<h1>Please use only IE browser to access this project..!!</h1>
</body>
</html>
```
invalidBrowser.jsp

Spring Form Tag Library

One of the view technologies you can use with the Spring Framework is Java Server Pages (JSPs). The Spring Web MVC framework provides a set of tags in the form of a tag library, which is used to construct views (web pages). The Spring Web MVC integrates spring's form tag library. Spring's form tag accesses to the command object, and also it refers to the data our spring controller deals with. A **Command** object can be defined as a JavaBean that stores user input, usually entered through HTML form is called the **Command** object or **Model** Object. The spring form tag makes it easier to develop, maintain, and read JSPs. The spring form tags are used to construct user interface elements such as text and buttons. Spring form tag library has a set of tags such as **<form>** and **<input>**. Each form tag provides support for the set of attributes of its corresponding HTML tag counterpart, which allows a developer to develop UI components in JSP or HTML pages.

Configuration – spring-form.tld

The spring form tag library comes bundled in **spring-webmvc.jar**. The **springform.tld** is known as **Tag Library Descriptor** (**tld**) file, which is available in a web application and generates HTML tags. The spring form tag library must be defined at the top of the JSP page. The following directive needs to be added to the top of your JSP pages, in order to use spring form tags from this library:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

Here, **form** is the tag name prefix, which will be used for the tags from this spring form tag library in JSP pages.

The following table shows few important tags of the spring form tag library:

Tag	Description
form:form	Generates the HTML **<form>** tag. It has the name attribute that specifies the command object that the inner tags should bind to.
form: input	Represents the HTML input text tag.
form:password	Represents the HTML input password tag.
form:radiobutton	Represents the HTML input radio button tag.
form: checkbox	Represents the HTML input checkbox tag.
form:select	Represents the HTML select list tag.
form:options	Represents the HTML options tag.
form:errors	Represents the HTML span tag. It also generates span tag from the error created as a result of validations.

The difference between Html tags and Spring form tags

Tags	HTML	Spring
Input	`<input type = "text">`	`<form:input>`

Radiobutton	`<input type="radiobutton">`	`<form:radiobutton>`
		
		`<form:radiobuttons>`
		Specify list of values for radiobuttons in one shot

Checkbox	`<input type="checkbox">`	`<form:checkbox>`
		
		`<form:checkboxes>`
		Specify list of values for checkboxes in one shot

Password	`<input type="password">`	`<form:password>`

Select and option	`<select name="course">`
	`<option value="java">java</option>`
	`<option value="spring">spring</option>`
	`</select>`
		`<form:select name="course">`
		`<form:option value="java" label="java"/>`
		`<form:option value="spring" label="spring"/>`
		`</form:select>`
		
		`<form:options>`
		Specify the list of options in one shot

Text area	`<input type="textarea">`	`<form:textarea>`
Hidden	`<input type="hidden">`	`<form:hidden>`

The <form: form> tag

The **<form: form>** tag is used to generate an HTML **<form: form>** tag in any of the JSP pages of a Spring application. It is used for binding the inner tags, which means that all the other tags are the nested tags in the **<form:form>** tag.

Using this **<form: form>** tag, the inner tags can access the **Command** object, which resides in the JSP's **PageContext** class.

```
<form:form method="POST" modelAttribute="user" action="register">
        ……..
</form: form>
```

The commandName & modelAttribute attribute

The attributes **commandName** and **modelAttribute** on the form:form tag do primarily the same thing, which is to map the form's fields to an Object of some type in the Controller. I believe modelAttribute is the preferred method, and commandName is only there for backwards compatibility.

The <form: input>tag: The **<form: input>** tag is used for entering the text by the user in any of the JSP pages of the spring web application. The following code snippet shows the use of the **<form: input>** tag in JSP pages:

```
<form:form method="POST" modelAttribute="user" action="register">
    <table>
      <tr>
        <td>Enter your name:</td>
        <td><form:input path="name" /></td>
      </tr>
      <tr>
        <td>Enter your mail:</td>
        <td><form:input path="email" /></td>
      </tr>
    </table>
</form:form>
```

The path attribute

The <form: input> tag renders an HTML <input type="text"/> element. The path attribute is the most important attribute of the input tag. This path attribute binds the input field to the form-backing object's property.

Let's take an example, if user is assigned as modelAttribute of the enclosing <form/> tag, then the path attribute of the input tag will be given as name or email. It should be noted that the User class contains getter and setter for name and email properties.

The <form:checkbox> tag

The **<form:checkbox>** tag is same as the HTML **<input>** tag, which is of the checkbox type.

```
<form:checkbox path="skills" value="Excel" label="Excel"/>
<form:checkbox path="skills" value="Word" label="Word"/>
<form:checkbox path="skills" value="Powerpoint"  label="Powerpoint"/>
```

In the preceding code snippet, the User class property called skills is used in the checkbox option. If the skills checkbox is checked, the User class's skills property is set accordingly.

The <form: radiobutton> tag

The **<form: radiobutton>** tag is used to represent the HTML **<input>** tag with the **radio** type in any of the JSP pages of a spring web application. It is used when there are many tag instances having the same property with different values, and only one radio value can be selected at a time. The following code snippet shows how we use the **<form:radiobutton>** tag in JSP pages:

```
<tr>
  <td>Gender:</td>
  <td>
    Male: <form:radiobutton path="gender" value="male"  label="male"/>
    Female: <form:radiobutton path="gender" value="female"  label="female"/>
  </td>
</tr>
```

The <form: password> tag

The **<form: password>** tag is used to represent the HTML **<input>** tag of the **password** type, in any of the JSP pages of a spring web application. By default, the browser does not show the value of the password field. We can show the value of the password field by setting the **showPassword** attribute to **true**. The following code snippet shows how we use the **<form: password>** tag in the JSP page:

```
<tr>
  <td>Password :</td>
  <td>
     <form:password path="password" />
  </td>
</tr>
```

The <form: select> tag

the **<form: select>** tag is used to represent an HTML **<select>** tag in any of the JSP pages of a Spring application. We use the **<form: select>** tag for binding the selected option with its value. The **<form: option>** tag is the nested tag in the **<form:select>** tag. The following code snippet shows how we use the **<select>** tag in the JSP pages:

```
<tr>
  <td>Department</td>
  <td>
     <form:select path="department" items="${departmentMap}"  />
  </td>
</tr>
```

Note: Here departmentMap is collection with multiple keys and values

The <form:option> tag

The **<form: option>** tag is used to represent an HTML **<option>** tag in any of the JSP pages of a Spring application. This tag is used when we need to add all the options to be **<form: select>** tag. Here, we need to add all the options inside the **<form: select>** tag. The following code snippet shows how to use the **<option>** tag in a JSP page:

```
<tr>
  <td>Department</td>
  <td><form:select path="department">
    <form:option value="technical" label="Technical" />
    <form:option value="non_technical" label="Non Technical" />
    <form:option value="r&d" label="R & D" />
  </form:select></td>
</tr>
```

The alternative code is below (with collection object)

```
<tr>
  <td>Department</td>
  <td><form:select path="department">
    <form:options items="${departmentMap}" />
  </form:select></td>
</tr>
```

The <form: textarea> tag

The **<form: textarea>** tag is used to represent an HTML **<textarea>** tag in any of the JSP pages of a Spring application. The following code snippet shows how to use the **<form: textarea>** tag in a JSP pages:

```
<td>Remarks :</td>
<td>
  <form:textarea path="remarks" rows="3"  cols="20"></form:textarea>
</td>
```

The <form: hidden> tag

The **<form: hidden>** tag is used to represent an HTML hidden field in a JSP page of a Spring application. The following code snippet shows how we use the **<hidden>** tag in JSP pages:

```
<form:hidden path="empId" />
```

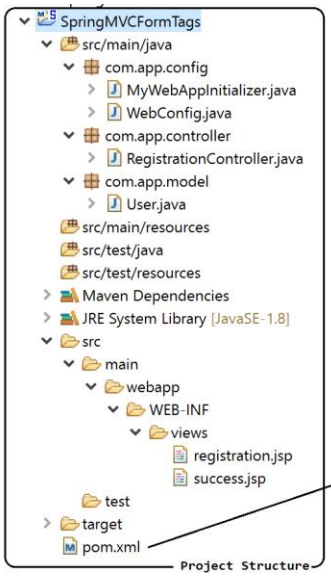
The <form: errors> tag

The **<form: errors>** tag is used to represent an HTML errors in a JSP of a Spring application. This tag is used for accessing the error defined in an **org. springframework.validation.Validator** interface. For example, if we want to submit a form, and find all the validator error related to the name and password fields in that form, we have to define all the validation errors related to these fields in a **Validator** class, which must implements the **Validator** interface as follows:

```
<form:form method="POST" modelAttribute="user" action="register">
    <table>
      <tr>
        <td>Enter your name:</td>
        <td><form:input path="name" /></td>
        <td><form:errors path="name" cssStyle="color: #ff0000;"/></td>
      </tr>
      <tr>
        <td>Enter your mail:</td>
        <td><form:input path="email" /></td>
        <td><form:errors path="email" cssStyle="color: #ff0000;" /></td>
      </tr>
      <tr>
        <td>Select your gender</td>
        <td><form:radiobuttons path="gender" items="${genders}" /></td>
        <td><form:errors path="gender" cssStyle="color: #ff0000;"/></td>
      </tr>
    </table>
</form>
```

Registration Form using Spring Form Tags

Step 1: Create one Maven Project like below



Project Structure

```xml
<properties>
        <failOnMissingWebXml>false</failOnMissingWebXml>
</properties>
<dependencies>
        <!-- Spring MVC Dependency -->
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-webmvc</artifactId>
                <version>4.3.7.RELEASE</version>
        </dependency>

        <!-- Servlet Dependency -->
        <dependency>
                <groupId>javax.servlet</groupId>
                <artifactId>javax.servlet-api</artifactId>
                <version>3.1.0</version>
                <scope>provided</scope>
        </dependency>

  </dependencies>
```

```xml
<!-- JSTL Dependency -->
<dependency>
        <groupId>javax.servlet.jsp.jstl</groupId>
        <artifactId>javax.servlet.jsp.jstl-api</artifactId>
        <version>1.2.1</version>
</dependency>
<dependency>
        <groupId>taglibs</groupId>
        <artifactId>standard</artifactId>
        <version>1.1.2</version>
</dependency>

<!-- JSP Dependency -->
<dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>javax.servlet.jsp-api</artifactId>
        <version>2.3.1</version>
</dependency>
```

pom.xml

Step-2: Creating a WebConfig Class

WebConfig.java

```java
package com.app.config;

import org.springframework.context.annotation.Bean;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = { "com.app.controller" })
public class WebConfig extends WebMvcConfigurerAdapter {

    @Bean
    public InternalResourceViewResolver resolver() {
        InternalResourceViewResolver resolver = new InternalResourceViewResolver();
        resolver.setViewClass(JstlView.class);
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        return resolver;
    }
}
```

WebConfig.java

Step-3: Create WebInitializer.java class

WebConfig.java | MyWebAppInitializer.java

```java
package com.app.config;

import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[] {};
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] { WebConfig.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

MyWebAppInitializer.java

Step 4: Create User.java (model class) like below

User.java

```java
package com.app.model;

import java.util.List;

public class User {

    private String name;

    private String email;

    private String gender;

    private String password;

    private String passwordConfirm;

    private List<String> courses;

    private List<String> batches;

    private String hiddenMsg;

    //setters && getters

}
```

User.java

Step-5 : Create RegistrationController class to handle the request

In the controller we have added method to display the Registration page and success page on submit.
We have also written a method to initialize the form with all the required values.

RegistrationController.java

```java
package com.app.controller;

import java.util.ArrayList;

@Controller
public class RegistrationController {

    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String displayUserPage(Model model) {
        User user = new User();
        user.setHiddenMsg("Ashok IT School");
        model.addAttribute("user", user);
        //to set form default values into model
        initializeFormValues(model);
        return "registration";
    }

    @RequestMapping(value = "/register", method = RequestMethod.POST)
    public String displayUserDetails(@ModelAttribute User user, Model model) {
        model.addAttribute("user", user);
        return "success";
    }

    private void initializeFormValues(Model model) {
        List<String> courses = new ArrayList<String>();
        courses.add("JSE"); courses.add("J2EE"); courses.add("Spring"); courses.add("Hibernate");
        courses.add("RESTful Services");
        model.addAttribute("courses", courses);

        List<String> genders = new ArrayList<String>();
        genders.add("Male"); genders.add("Female");
        model.addAttribute("genders", genders);

        List<String> batches = new ArrayList<String>();
        batches.add("Morning"); batches.add("Afternoon");
        batches.add("Evening");

        model.addAttribute("batches", batches);

    }
}
```

Step 6: Create registration.jsp page to display registration form with input fields like below

- path is the spring form tag attribute used to bind the form field with the model class.
- <form: errors> tag is used to specify error to be displayed for the corresponding field.
- We can specify the list of strings directly using items as we used in
  < form: checkboxes> and < form:select>
- List specified with items is used to display multiple values and path is used to bind the selected value into the model class variable.

----------registration.jsp----------

```jsp
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<html>
<head>
<title>Registraion</title>
</head>
<body>
	<h2>Register Here</h2>
	<hr />
	<form:form method="POST" modelAttribute="user" action="register">
		<table>
			<tr>
				<td>Enter your name:</td>
				<td><form:input path="name" /></td>

				<td><form:errors path="name" cssStyle="color: #ff0000;" /></td>
```

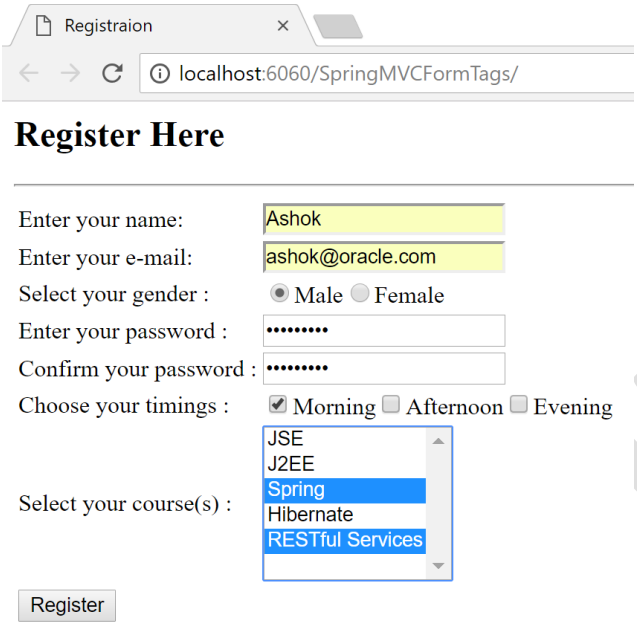
```
                </tr>
                <tr>
                        <td>Enter your e-mail:</td>
                        <td><form:input path="email" /></td>
                        <td><form:errors path="email" cssStyle="color: #ff0000;" /></td>
                </tr>
                <tr>
                        <td>Select your gender :</td>
                        <td><form:radiobuttons path="gender" items="${genders}" /></td>
                        <td><form:errors path="gender" cssStyle="color: #ff0000;" /></td>
                </tr>
                <tr>
                        <td>Enter your password :</td>
                        <td><form:password path="password" showPassword="true" /></td>
                        <td><form:errors path="password" cssStyle="color: #ff0000;" /></td>
                <tr>
                        <td>Confirm your password :</td>
                        <td><form:password path="passwordConfirm" showPassword="true" /></td>
                        <td><form:errors path="passwordConfirm"
                                        cssStyle="color: #ff0000;" /></td>
                </tr>
                <tr>
                        <td>Choose your timings :</td>
                        <td><form:checkboxes path="batches" items="${batches}" /></td>
                        <td><form:errors path="batches" cssStyle="color: #ff0000;" /></td>
                </tr>
                <tr>
                        <td>Select your course(s) :</td>
                        <td><form:select path="courses" size="6">
                                        <form:options items="${courses}" />
                                </form:select></td>
                        <td><form:errors path="courses" cssStyle="color: #ff0000;" /></td>
                </tr>
                <tr>
                        <form:hidden path="hiddenMsg" />
                </tr>
                <tr>
                        <td><input type="submit" name="submit" value="Register"></td>
                </tr>
                <tr>
            </table>
        </form:form>
</body>
```

Step 7 : Create success.jsp page to display captured form data as a response

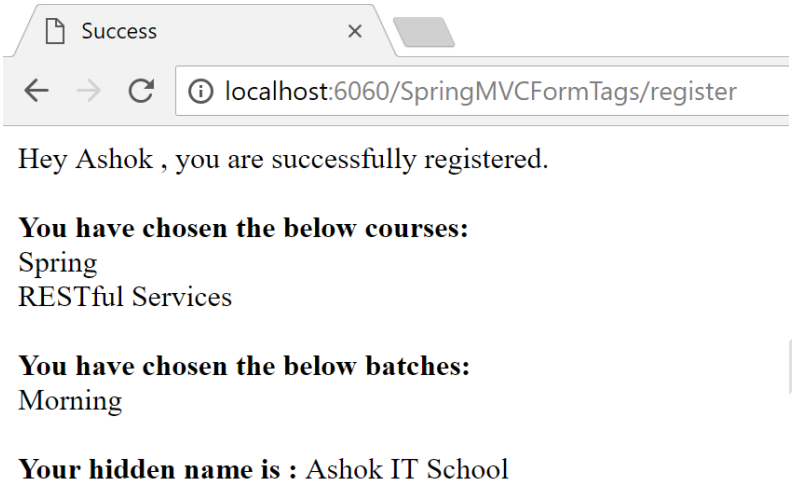
success.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Success</title>
</head>
<body>
    Hey ${user.name} , you are successfully registered.
    <br />
    <b>You have chosen the below courses: </b>
    <br>
    <c:forEach var="course" items="${user.courses}">
        <c:out value="${course}" />
        <br>
    </c:forEach>
    <b>You have chosen the below batches:</b>
    <c:forEach var="batch" items="${user.batches}">
        <c:out value="${batch}" />
        <br>
    </c:forEach>
    <br>
    <b>Your hidden name is : </b> ${user.hiddenMsg}
</body>
</html>
```

Step 8: Deploy the application into server and access using below URL and fill the form

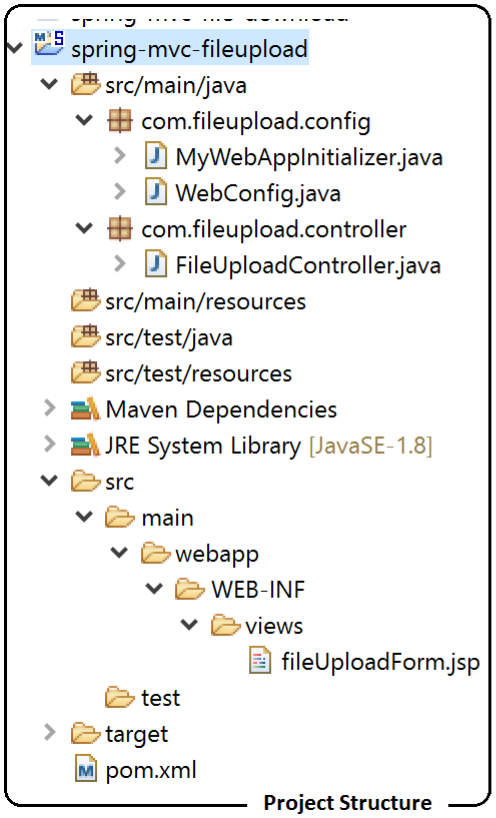


Step 9: Click on Register button, below screen will be displayed



Spring MVC File(s) Uploading - Application

Step-1: Create Maven Project and add dependencies



Project Structure

```
    <properties>
        <failOnMissingWebXml>false</failOnMissingWebXml>
    </properties>
    <dependencies>
        <!-- Spring MVC Dependency -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>4.3.6.RELEASE</version>
        </dependency>

        <!-- Servlet Dependency -->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.1.0</version>
        </dependency>

        <!-- JSP Dependency -->
        <dependency>
            <groupId>javax.servlet.jsp</groupId>
            <artifactId>javax.servlet.jsp-api</artifactId>
            <version>2.3.1</version>
        </dependency>

    </dependencies>
```

Maven Dependencies

Step-2: Create WebConfig class

```
package com.fileupload.config;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = { "com.fileupload.controller" })
public class WebConfig extends WebMvcConfigurerAdapter {

	@Bean
	public InternalResourceViewResolver resolver() {
		InternalResourceViewResolver resolver = new InternalResourceViewResolver();
		resolver.setViewClass(JstlView.class);
		resolver.setPrefix("/WEB-INF/views/");
		resolver.setSuffix(".jsp");
		return resolver;
	}

	@Bean
	public MultipartResolver multipartResolver() {
		StandardServletMultipartResolver multipartResolver = new StandardServletMultipartResolver();
		return multipartResolver;
	}

}
```

WebConfig.java

Step-3: Create Web Initializer class

```
package com.fileupload.config;

public class MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {
	@Override
	protected Class<?>[] getRootConfigClasses() {
		return new Class[] {};
	}
	@Override
	protected Class<?>[] getServletConfigClasses() {
		return new Class[] { WebConfig.class };
	}

	@Override
	protected String[] getServletMappings() {
		return new String[] { "/" };
	}
	@Override
	protected void customizeRegistration(Dynamic registration) {

		// Parameters:-
		// location - the directory location where files will be stored
		// maxFileSize - the maximum size allowed for uploaded files
		// maxRequestSize - the maximum size allowed for multipart/form-data
		// requests
		// fileSizeThreshold - the size threshold after which files will be
		// written to disk
		MultipartConfigElement multipartConfig = new MultipartConfigElement("C:\\Desktop\\File", 1048576, 10485760, 0);
		registration.setMultipartConfig(multipartConfig);
	}
}
```

MyWebAppInitializer.java

Step-4: Create Controller class to handle file upload request

FileUploadController.java

```java
package com.fileupload.controller;

import java.io.BufferedOutputStream;

@Controller
public class FileUploadController {

    @GetMapping("/")
    public String fileUploadForm(Model model) {
        return "fileUploadForm";
    }
    // Handling single file upload request
    @PostMapping("/singleFileUpload")
    public String singleFileUpload(@RequestParam("file") MultipartFile file, Model model) throws IOException {
        // Save file on system
        if (!file.getOriginalFilename().isEmpty()) {
            File f = new File("C:\\Desktop\\File\\SingleFileUpload", file.getOriginalFilename());
            FileOutputStream fis = new FileOutputStream(f);
            BufferedOutputStream outputStream = new BufferedOutputStream(fis);
            outputStream.write(file.getBytes());
            outputStream.flush();
            outputStream.close();
            model.addAttribute("smsg", "File uploaded successfully.");
        } else {
            model.addAttribute("smsg", "Please select a valid file..");
        }

        return "fileUploadForm";
    }
```

FileUploadController.java

```java
// Handling multiple files upload request
@PostMapping("/multipleFileUpload")
public String multipleFileUpload(@RequestParam("file") MultipartFile[] files, Model model) throws IOException {
    // Save file on system
    for (MultipartFile file : files) {
        if (!file.getOriginalFilename().isEmpty()) {
            FileOutputStream fos = new FileOutputStream(new File("C:\\Desktop\\File\\MultiFileUpload", file.getOriginalFilename()));
            BufferedOutputStream outputStream = new BufferedOutputStream(fos);
            outputStream.write(file.getBytes());
            outputStream.flush();
            outputStream.close();
        } else {
            model.addAttribute("mmsg", "Please select at least one file..");
            return "fileUploadForm";
        }
    }
    model.addAttribute("mmsg", "Multiple files uploaded successfully.");
    return "fileUploadForm";
}
```

To Support Multiple files at a time

FileController method to upload multiple files

Step-5: Deploy the application, below screen will be displayed where we can upload single or multiple files

Spring application with Rest Integration

Introduction

RESTful web services are built to work best on the Web. Representational State Transfer (REST) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web service induce desirable properties, such as performance, scalability, and modifiability that enable services to work best on the Web. In the REST architectural style, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs), typically links on the Web. The resources are acted upon by using a set of simple, well-defined operations. The REST architectural style constrains an architecture to a client/server architecture and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol.

The following principles encourage RESTful applications to be simple, lightweight, and fast:

Resource identification through URI: A RESTful web service exposes a set of resources that identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery. See the @Path Annotation and URI Path Templates for more information.

Uniform interface: Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE. PUT creates a new resource, which can be then deleted by using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource. See Responding to HTTP Methods and Requests for more information.

Self-descriptive messages: Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others. Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control. See Responding to HTTP Methods and Requests and Using Entity Providers to Map HTTP Response and Request Entity Bodies for more information.

Stateful interactions through hyperlinks: Every interaction with a resource is stateless; that is, request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction. See Using Entity Providers to Map HTTP Response and Request Entity Bodies and “Building URIs” in the JAX-RS Overview document for more information

RESTful Web Services utilize the features of the HTTP Protocol to provide the API of the Web Service. It uses the HTTP Request Types to indicate the type of operation:

GET: Retrieve / Query of existing records.

POST: Creating new records.

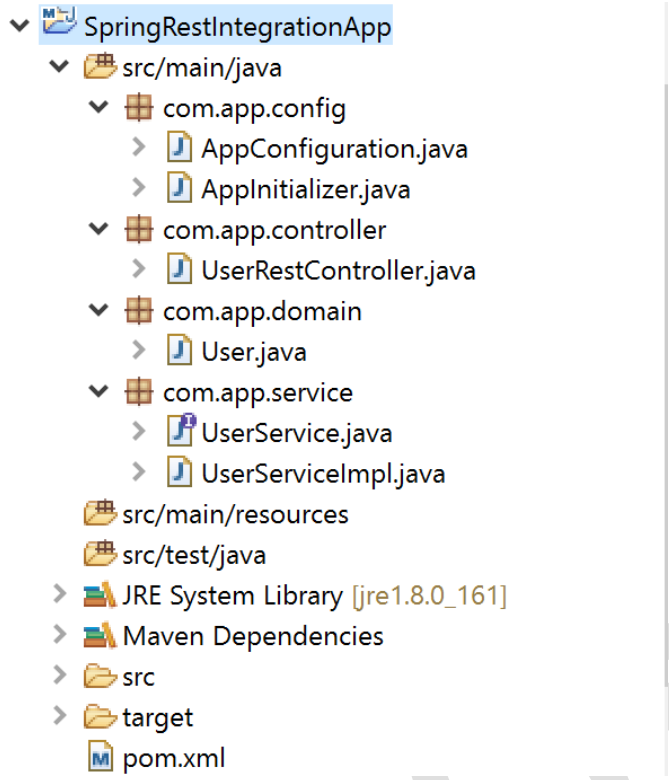
DELETE: Removing records.

PUT: Updating existing records.

Using these 4 HTTP Request Types a RESTful API mimics the CRUD operations (Create, Read, Update & Delete). REST is stateless, each call the to a RESTful Web Service is completely stand-alone, it has no knowledge of previous requests.

Below REST application performs CURD operations using Spring Service class

Step 1: Create Maven Web Project



Step 2: Configure maven dependencies in project pom.xml file

```
<properties>
	<springframework.version>4.3.0.RELEASE</springframework.version>
	<jackson.library>2.7.5</jackson.library>
</properties>
<dependencies>
	<dependency>
		<groupId>org.springframework</groupId>
		<artifactId>spring-core</artifactId>
		<version>${springframework.version}</version>
	</dependency>
	<dependency>
		<groupId>org.springframework</groupId>
		<artifactId>spring-web</artifactId>
		<version>${springframework.version}</version>
	</dependency>
	<dependency>
		<groupId>org.springframework</groupId>
		<artifactId>spring-webmvc</artifactId>
		<version>${springframework.version}</version>
	</dependency>
	<dependency>
		<groupId>javax.servlet</groupId>
		<artifactId>javax.servlet-api</artifactId>
		<version>3.1.0</version>
	</dependency>
	<dependency>
		<groupId>com.fasterxml.jackson.core</groupId>
		<artifactId>jackson-databind</artifactId>
		<version>${jackson.library}</version>
	</dependency>
</dependencies>
```

Step 3: Create Domain class for Storing and Retrieving the data (User.java)

User.java

```
package com.app.domain;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "user")
@XmlAccessorType(XmlAccessType.FIELD)
public class User {

    private Integer userid;
    private String username;
    private String email;
    private String gender;

    //setters & getters

    //toString( )

}
```

Step 4: Create UserService.java class to perform Business operations

```
package com.app.service;

@Service(value = "service")
public class UserServiceImpl implements UserService {

	private static Map<Integer, User> usersData = new HashMap<Integer, User>();

	public boolean add(User user) {
		if (usersData.containsKey(user.getUserid())) {
			return false;
		} else {
			usersData.put(user.getUserid(), user);
			return true;
		}
	}
	public User get(String uid) {
		System.out.println(usersData);
		if (usersData.containsKey(Integer.parseInt(uid))) {
			return usersData.get(Integer.parseInt(uid));
		}
		return null;
	}
	public boolean update(String uid, User user) {
		if (usersData.containsKey(Integer.parseInt(uid))) {
			usersData.put(Integer.parseInt(uid), user);
			return true;
		}
		return false;
	}
	public boolean delete(String uid) {
		if (usersData.containsKey(Integer.parseInt(uid))) {
			usersData.remove(usersData.get(Integer.parseInt(uid)));
			return true;
		}
		return false;
	}
}
```

UserServiceImpl.java

Step 5: Create RestController (UserRestController.java)

```
package com.app.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

import com.app.domain.User;
import com.app.service.UserService;

@RestController
public class UserRestController {

	@Autowired(required = true)
	private UserService service;

	@RequestMapping(value = "/add", method = RequestMethod.POST, consumes = { "application/xml", "applica-
tion/json" })
	public @ResponseBody String addUser(@RequestBody User user) {
		boolean isAdded = service.add(user);
		if (isAdded) {
			return "User Added successfully";
		} else {
			return "Failed to Add the User..!";
		}
	}
```

```
	@RequestMapping(value = "/get", produces = { "application/xml", "application/json" }, method = Request-
Method.GET)
	@ResponseBody
	public User getUserById(@RequestParam(name = "uid") String uid) {
		System.out.println("Getting User with User Id : " + uid);
		User user = service.get(uid);
		return user;
	}

	@RequestMapping(value = "/update", method = RequestMethod.PUT,
			consumes = { "application/xml", "application/json" }
	)
	public @ResponseBody String update(@RequestParam("uid") String uid, @RequestBody User user) {
		boolean isAdded = service.update(uid, user);
		if (isAdded) {
			return "User updated successfully";
		} else {
			return "Failed to update the User..!";
		}
	}

	@RequestMapping(value = "/delete", method = RequestMethod.DELETE)
	public @ResponseBody String delete(@RequestParam("uid") String uid) {
		boolean isAdded = service.delete(uid);
		if (isAdded) {
			return "User Deleted successfully";
		} else {
			return "Failed to Delete the User..!";
		}
	}

	public void setService(UserService service) {
		this.service = service;
	}
}
```

Step 6: Create AppConfig and AppInitiazer classes

AppConfiguration.java

AppConfiguration.java

```
package com.app.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.app.*")
public class AppConfiguration {

}
```

AppInitializer.java

AppInitializer.java

```
package com.app.config;

import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class AppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[] { AppConfiguration.class };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return null;
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/rest/*" };
    }

}
```

Step 7: Deploy the Application into server (I have used Apache Tomcat 8.0)

Step 8: Test the Application using POSTMAN plugin in Google Chrome

Below Screen shots shows you how test Rest Resources using POSTMAN plugin

Testing Add User

URI: http://localhost:6060/SpringRestIntegrationApp/rest/add

Method Type: POST

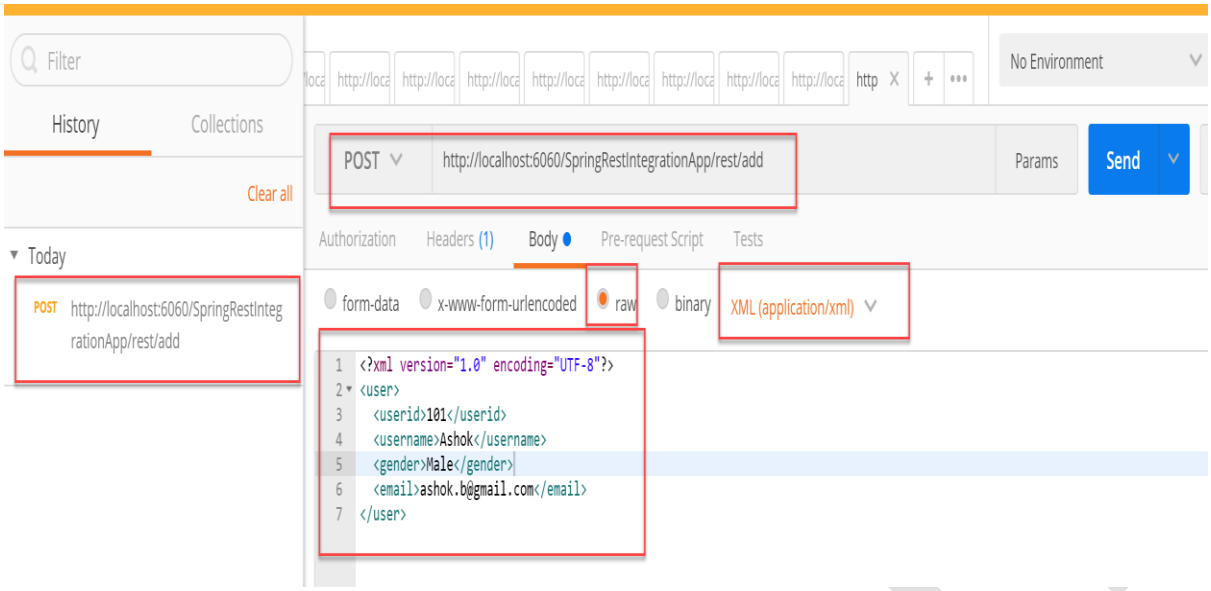
Consumes: {application/xml, application/json}

Produces: text/plain

Request Body Data: In XML Format

```
<? xml version="1.0" encoding="UTF-8"?>
<user>
        <userid>101</userid>
        <username>Ashok</username>
        <gender>Male</gender>
        <email>ashok.b@gmail.com</email>
</user>
```

POSTMAN Screenshot



Testing Fetch User

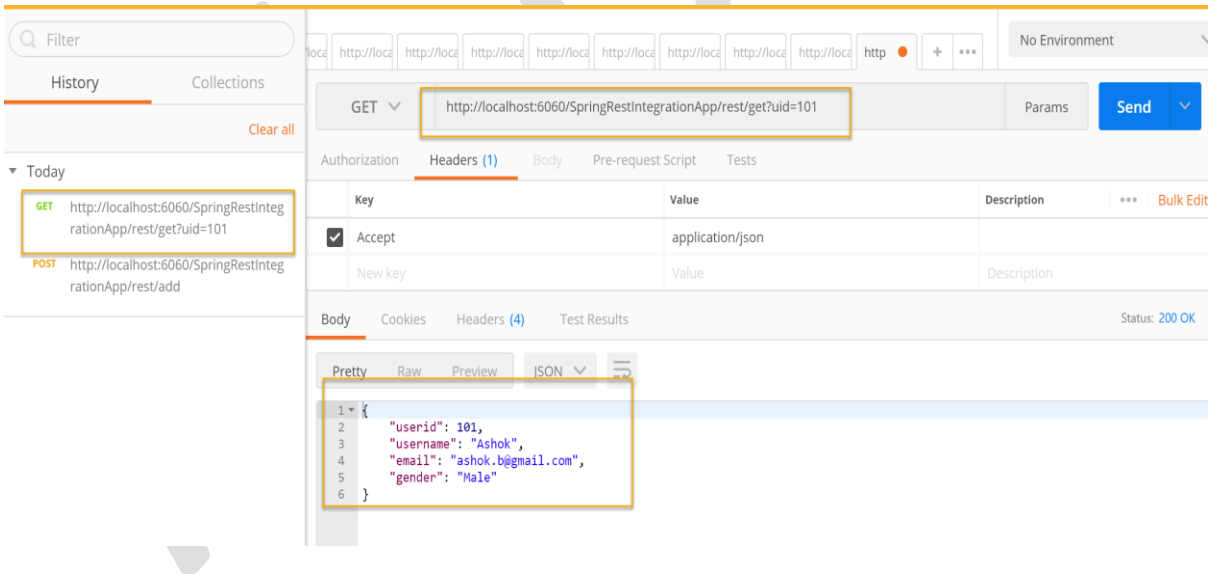
URI: http://localhost:6060/SpringRestIntegrationApp/rest/get?uid=101

Method Type: GET

Input: Request Parameter (? uid=101)

Produces: {application/xml, application/json}

POSTMAN Screenshot



Testing Update User

URL: http://localhost:6060/SpringRestIntegrationApp/rest/update?uid=101

Method Type: PUT

Input in URL: User Id (Request Parameter)? uid=101

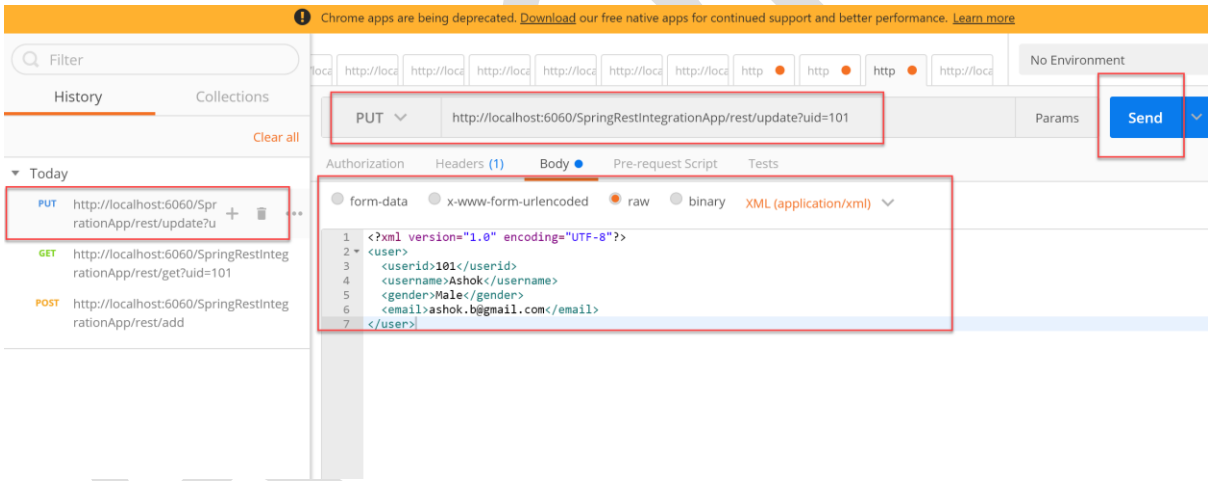
Consumes: {application/xml, application/json}

Produces: text/plain

Request Body Data: in XML format

```
<? xml version="1.0" encoding="UTF-8"?>
<user>
        <userid>101</userid>
        <username>Ashok</username>
        <gender>Male</gender>
        <email>ashok.b@gmail.com</email>
</user>
```

POSTMAN Screenshot



Testing Delete User

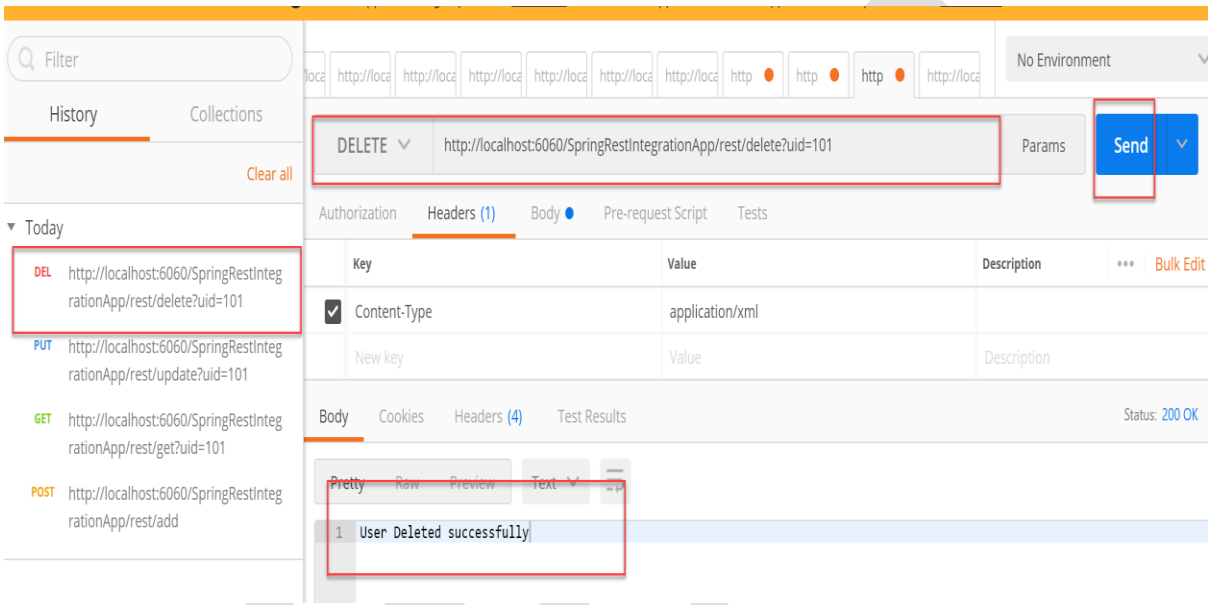
URL: http://localhost:6060/SpringRestIntegrationApp/rest/delete?uid=101

Method Type: DELETE

Input: Request Parameter (? uid=101)

Produces: text/plain

POSTMAN Screenshot



====0000====