Deloitte.

Spring MVC

Deloitte Technology Academy (DTA)

Agenda

</ />

Topic	Descriptions	Duration
Spring MVC	Process Flow, Components, Architecture, and Spring Framework Stereotype Annotations	X hours XX mins
Spring Handler–Adapters and Mappings	Handler Adapter, Handler Mappings, and Types of Handler mappings	X hours XX mins
View Resolvers	XmlViewResolver and InternalResourceViewResolver	X hours XX mins
Spring Validation	Spring MVC Validation, Server-Side Validation and Bean Validation API	X hours XX mins

Learning Objectives

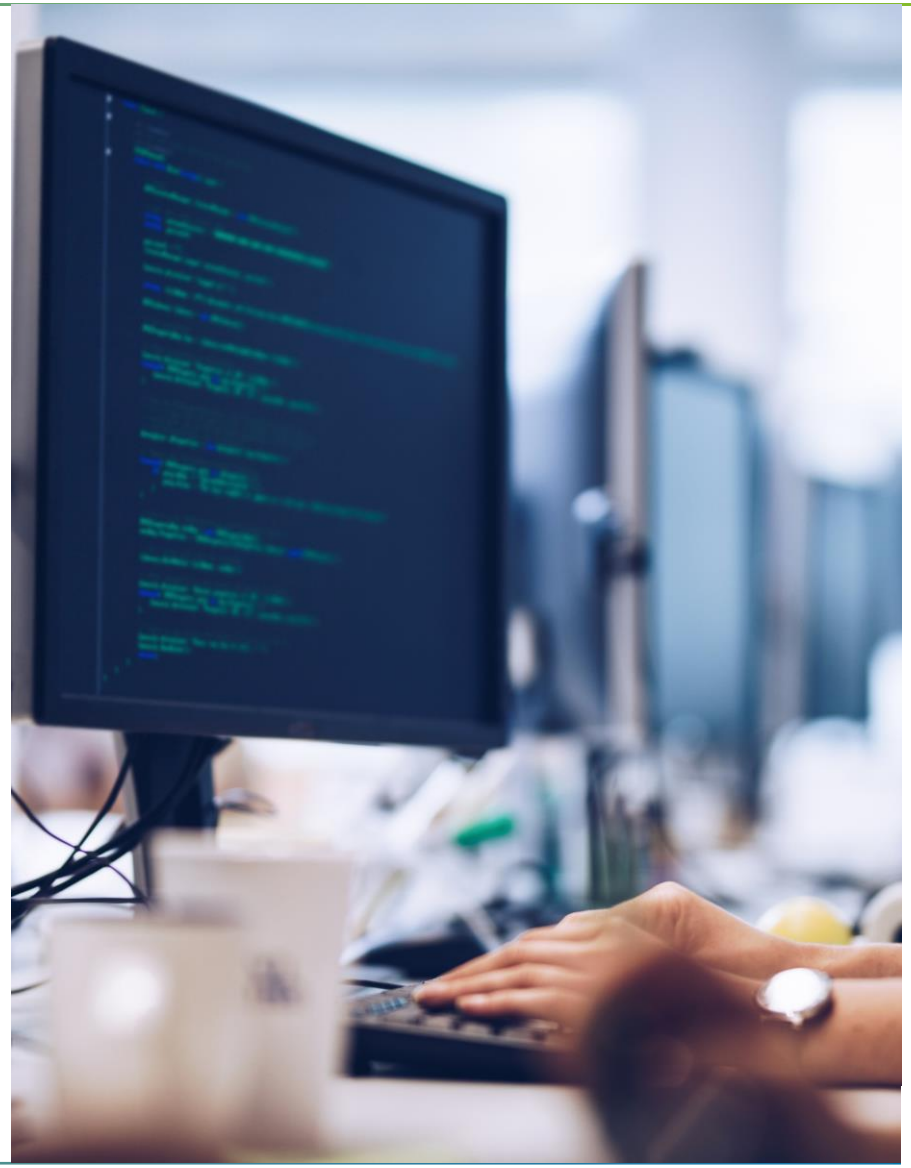
< / >

By the end of this session, you will be able to:

- Explain Spring MVC architecture and flow
- Define and use Stereotype Annotations
- Describe handler adapters and mappings
- Explain the usage of View Resolvers
- Describe performing Validations

</ >

Spring MVC



Introduction

< / >

Overview:

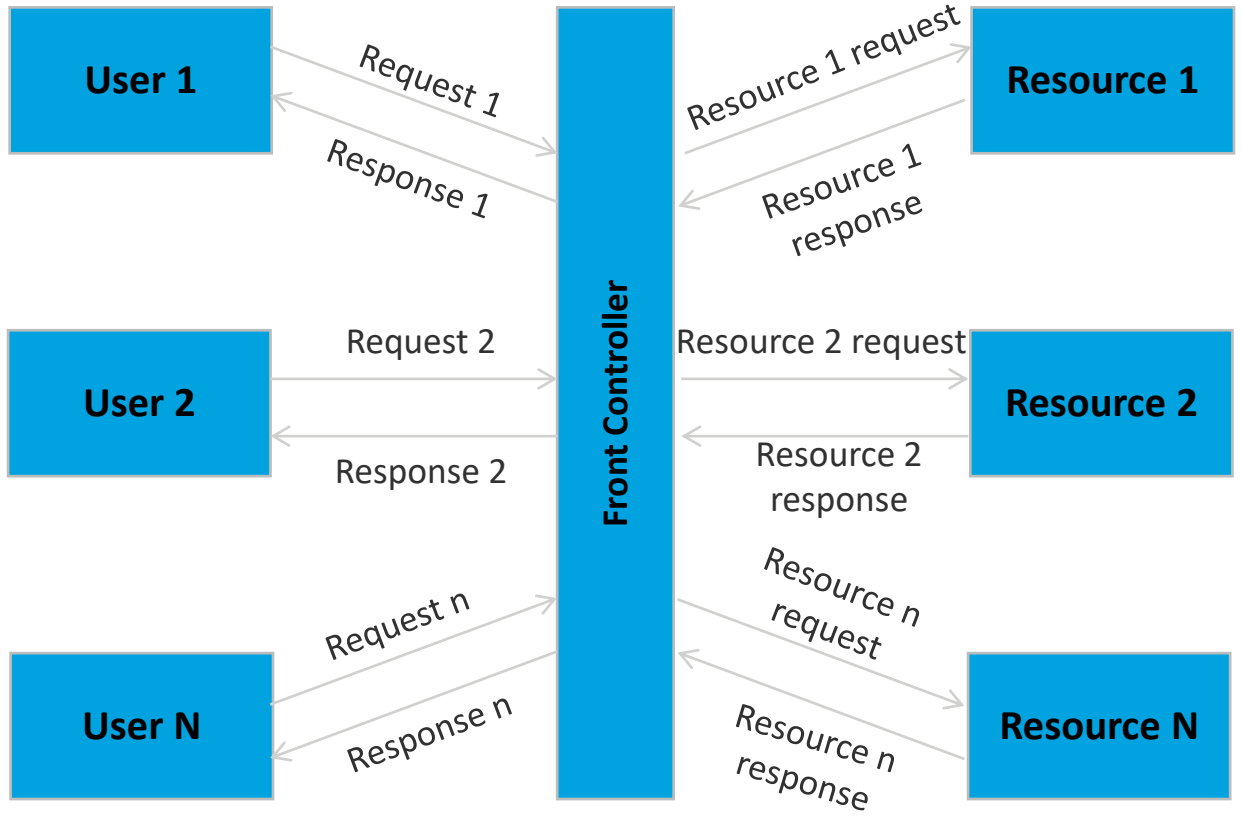
- Provides MVC architecture and ready components to develop flexible and loosely coupled Web Applications
- MVC pattern separates aspects of the application (input, business, and UI logic) while providing a loose coupling between them.
- An elegant solution to utilize MVC in the spring framework is provided with the help of DispatcherServlet

Spring	Spring MVC
• Spring Is an open-source framework created to address the complexity of enterprise application development.	• Spring MVC is a Java framework for building web applications. • The Model-View-Controller Design Pattern is followed. • All of the core spring framework basic features are implemented like Inversion of Control and Dependency Injection.

Process Flow

Front Controller Design Pattern

- For all the incoming requests a single point of entry is enforced.
- A single piece of code handles all the requests.
- Processing the request is further delegated to further application objects.

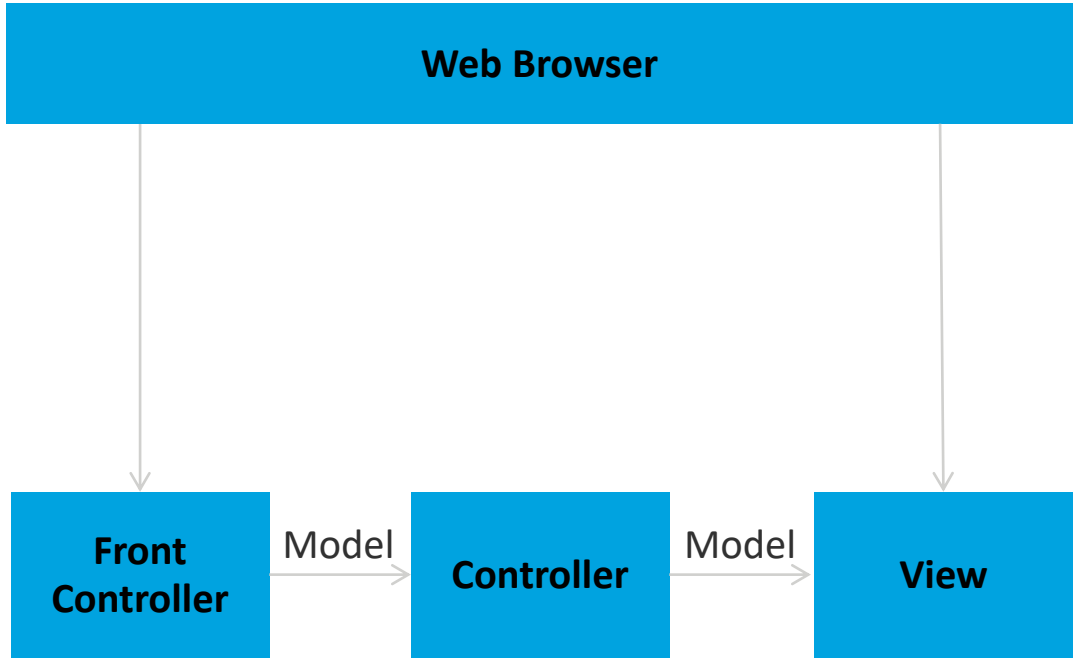


Process Flow (Cont.)

< / >

Spring Web Model-View-Controller

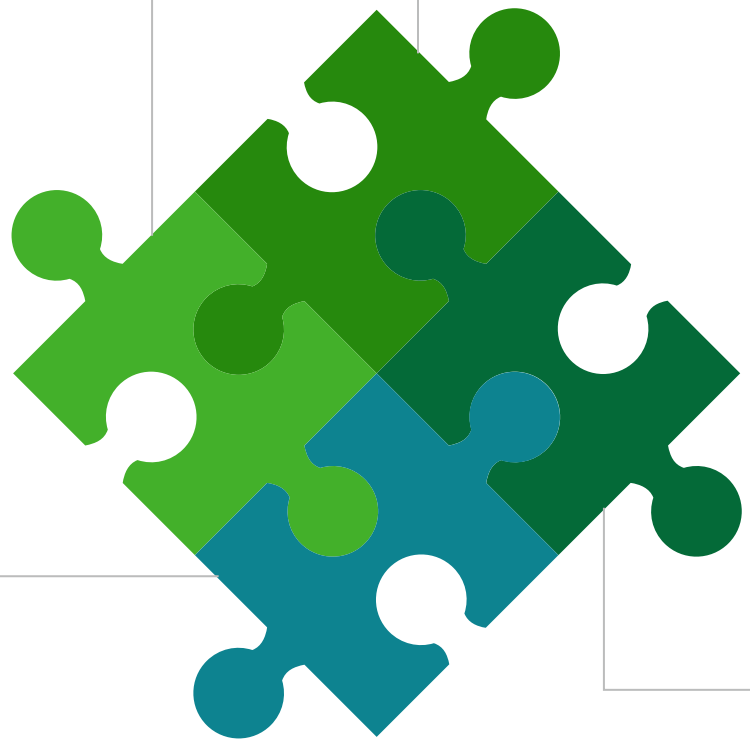
- The flow depicts the sequence of events as how the Spring MVC handles a user request.
- There can be only one Front Controller, any number of Models, Controllers, or Views.
- The above diagram is well explained in the next slide.



Spring MVC Components

Front Controller

- In Spring Web MVC, the Dispatcher Servlet class works as the Front Controller.
- It is responsible for managing the flow of the Spring MVC application.



Model

- Data of the application is contained in a model.
- A data can be a collection of objects or a single object.

View

- Provided information is represented in a particular format.
- To create a view page JSP+JSTL is generally used.

Controller

- Business logic of an application is contained in a controller.
- Here, the @Controller annotation marks the class as the controller.

Advantages

Separate of Concerns (roles/modules)

Separates each role, where Model object, controller, command object, ViewResolver, DispatcherServlet, Validator, etc., can be fulfilled by a specialized object.

Powerful Configuration

It provides a robust configuration both for framework and application classes that includes easy referencing across contexts such as from business objects to web controllers and validators.

Light-weight

It uses a lightweight servlet container to develop and deploy your application.

Rapid development

Parallel and fast development is facilitated by the Spring MVC.

Advantages (Cont.)

Reusable business code

You are allowed to use the existing business objects instead of having to create new objects.

Easy to test

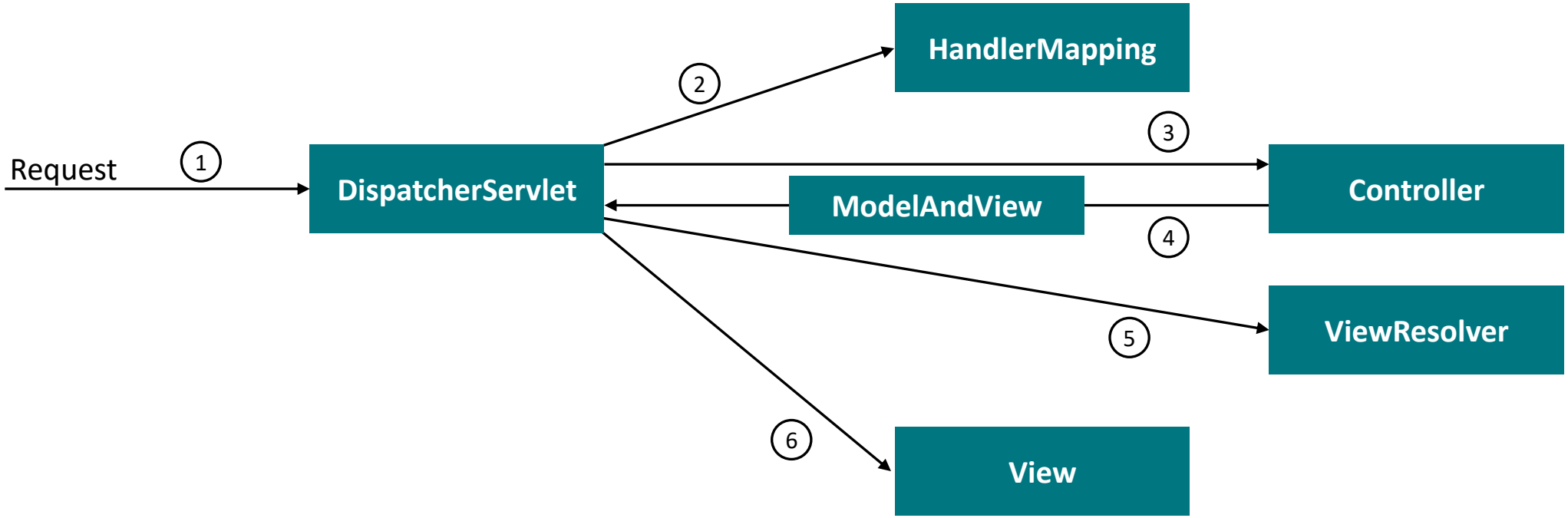
In spring, we create JavaBeans classes that allow us to inject test data using the setter methods.

Flexible Mapping

It provides specific annotations that easily redirect the page.

Architecture

- The DispatcherServlet intercepts all incoming requests and acts as the “front controller .”
- The DispatcherServlet receives handler mapping from the XML file and forwards the request to the controller.
- The controllers return an object of ModelAndView.
- The DispatcherServlet checks the entry of the view resolver in the XML file and invokes the specified view component.



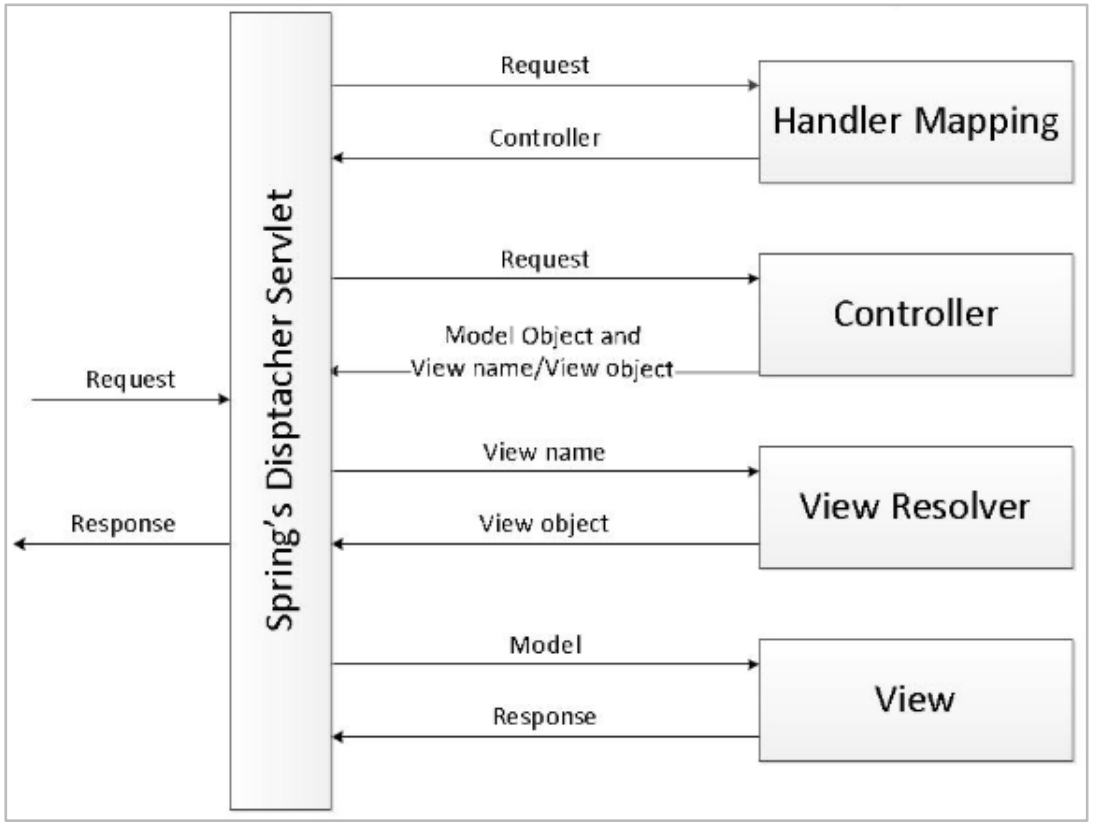
Architecture (Cont.)

</ />

- Spring’s MVC module is based on a front controller design pattern followed by the MVC design pattern.
- The single servlet named DispatcherServlet handles all incoming requests, which in Spring’s MVC module it acts as the front controller.
- The DispatcherServlet then refers to the HandlerMapping (from the XML file) to find a controller object which can handle the request and forwards the request to the controller.
- The request is then dispatched by the DispatcherServlet to the controller object so that it can perform the business logic to fulfill the user request.
- The view object and the model object (or a logical name of the view) are contained in an encapsulated object that is returned by the controller. In Spring’s MVC, the class ModelAndView represents this encapsulated object.

Architecture (Cont.)

- The DispatcherServlet refers to the ViewResolver if ModelAndView contain the logical name of the view to discover the actual View object based on the logical name.
- The DispatcherServlet checks the entry of the view resolver in the XML file and invokes the specified view component.
- The model object is then passed to the view object by DispatcherServlet, which is then rendered to the end user.



Spring Framework Stereotype Annotations

< / >

- Any class that fulfills a role within an application can be marked by sterotype annotations.
- This helps to at least greatly reduce or remove, the Spring XML configuration required for these components.
  - @Component
  - @Controller
  - @Service
  - @Repository

Spring Framework Stereotype Annotations (Cont.)

< / >

@Component

- To indicate a Spring component; it is used in classes.
- The @Component annotation marks the Java class as a component or bean so that Spring can add it into the application context by the component-scanning mechanism.
- @Component is common for any component managed by Spring.
- @Repository, @Service, and @Controller are variations of @Component for focused uses, such as in the persistence, service, and presentation layers, respectively.

@Controller

- To designate the class is a Spring controller @controller is used.
- To identify controllers for Spring MVC.
- In Spring MVC, you can easily make a controller class by prefixing @Controller before any class declaration.

Spring Framework Stereotype Annotations (Cont.)

< / >

@Repository

- For Java classes that directly access the database @Repository is used.
- For any class that has the role of Data Access Object or repository it works as a marker.
- There is an automatic translation feature for this annotation. For instance, when an exception occurs in the @Repository, there is no need to add a try-catch block instead there is a handler for that exception.
- Known as a repository in other technologies, this class serves as a Data Access Object (DAO) in the persistence layer of the application.
- All your DAO classes should be annotated with @Repository. DAO classes should be all of your database access logic.

@Service

- Is used on a class
- Performing calculations, executing business logic, and calling external APIs are some services in a Java class that @Service marks performs.
- This annotation is intended to be used in the service layer and is a specialize form of the @Component annotation.
- All your service classes should be annotated with @Service. Service classes should be used for all your business logic.

Spring Framework Stereotype Annotations (Cont.)

< / >

@RestController

- The @RestController annotation indicates the class as a controller so that every method returns a domain object in place of a view.
- Once the class is annotated, there is no need to add @ResponseBody to all the RequestMapping methods. As a result, no need to use view-resolvers or send HTML in response. You send the domain object as an HTTP response in the format the users understand, like JSON.
- @RestController = @Controller + @ResponseBody
- To generate RESTful web services using Spring MVC, Spring RestController annotation is used. This annotation ensures that request data is mapped to the defined request handler method.
- Once the handler method generates the response body, it converts it to JSON or XML response.

Interceptor

- To intercept client requests and process them, Spring Interceptors are used.
- Spring MVC Interceptor is especially useful for intercepting the HTTP Request and processing it before passing it to the controller handler.
- We can create our Spring interceptor by either implementing org.springframework.web.servlet.HandlerInterceptor interface or by overriding abstract class org.springframework.web.servlet.handler.HandlerInterceptorAdapter that provides the base implementation of HandlerInterceptor interface.

Spring Framework Stereotype Annotations (Cont.)

</ />

@Configuration

- Is used on classes that define beans.
- @Configuration is an analog for an XML configuration file. It is configured using Java classes.
- A Java class annotated with @Configuration is a configuration by itself and will have methods to instantiate and configure the dependencies.

```
@Configuartion
public class DataConfig {
    @Bean
    public DataSource source() {
        DataSource source = new OracleDataSource();
        source.setURL();
        source.setUser();
        return source;
    }
    @Bean
    public PlatformTransactionManager manager() {
        PlatformTransactionManager manager = new BasicDataSourceTransactionManager();
        manager.setDataSource(source());
        return manager;
    }
}
```

Dispatcher Servlet—Configuration

</ />

Example 1: DispatcherServlet—Config

- Using a URL mapping in the web.xml file, map the requests that you want the DispatcherServlet to manage.
- Example: Declaration and Mapping for HelloWeb DispatcherServlet

```
<web-app id = "WebApp_ID" version = "2.4"
   xmlns = "http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://java.sun.com/xml/ns/j2ee
   http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

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

   <servlet>
      <servlet-name>HelloWeb</servlet-name>
      <servlet-class>
         org.springframework.web.servlet.DispatcherServlet
      </servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
      <servlet-name>HelloWeb</servlet-name>
      <url-pattern>*.jsp</url-pattern>
   </servlet-mapping>
</web-app>
```

web.xml location	WebContent/WEB-INF directory
Upon initialization of HelloWeb	Framework will load the application context from a file named [servlet-name]-servlet.xml located in the WebContent/WEB-INF directory.
In this case	Our file will be HelloWeb-servlet.xml.
<servlet-mapping> tag	Specifies which URLs will be managed by which DispatcherServlet.
HTTP requests ending with .jsp	Will be handled by the HelloWeb DispatcherServlet.

Dispatcher Servlet—Configuration (Cont.)

< / >

Configuration for HelloWeb-servlet.xml file, placed in web application's WebContent/WEB-INF directory:

```
<beans xmlns = "http://www.springframework.org/schema/beans"
   xmlns:context = "http://www.springframework.org/schema/context"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   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">

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

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

[servlet-name]-servlet.xml	Used to generate the beans defined and override the definitions of any beans defined with the same name in the global scope.
<context:component-scan...> tag	Used to trigger Spring MVC annotation scanning that enables using annotations like @Controller and @RequestMapping etc.,
InternalResourceViewResolver	Comes with rules for resolving view names. According to the above defined rule, a logical view named hello is delegated to a view implementation located at /WEB-INF/jsp/hello.jsp

Dispatcher Servlet—Configuration (Cont.)

Custom DispatcherServlet—Config

If you don’t prefer the default filename as [servlet-name]-servlet.xml and location as WebContent/WEB-INF, you can modify them as you like by adding the servlet listener ContextLoaderListener in your web.xml file as shown:

```
<web-app...>

   <!-------- DispatcherServlet definition goes here----->
   ....
   <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/HelloWeb-servlet.xml</param-value>
   </context-param>

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

</web-app>
```

Dispatcher Servlet—Configuration (Cont.)

Example 2: DispatcherServlet—Config

- DispatcherServlet works as the front controller in the MVC module; it manages all user requests.
- The servlet must be configured like other servlets in the application's web deployment descriptor file—web.xml.
- We call this servlet—myLibraryAppFrontController.
- URI pattern in the servlet mapping section—*.htm
- As a result, all request that match the URI pattern will be managed by myLibraryAppFrontController.

```
<web-app xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
id="WebApp_ID"
version="2.5"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
   <display-name>HelloWorldWeb</display-name>
   <servlet>
      <servlet-name>helloWordWebController</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>helloWordWebController</servlet-name>
      <url-pattern>*.htm</url-pattern>
   </servlet-mapping>
   <welcome-file-list>
      <welcome-file>welcome.htm</welcome-file>
   </welcome-file-list>
</web-app>
```

Defining a Controller

</ />

@RequestMapping is one of the most common annotations used in Spring Web applications.

- It maps HTTP requests to handler methods of MVC and REST controllers.
- While configuring Spring MVC, you need to specify the mappings between requests and handler methods.
- For configuring the mapping of web requests, use the @RequestMapping annotation.

DispatcherServlet	Delegates web request to Controllers to execute functionality specific to it
@Controller	Indicates that a particular class serves the role of a controller
@RequestMapping	Maps a URL to an entire class or a particular handler method

Defining a Controller (Cont.)

</>

- The value attribute specifies the URL to which the handler method is mapped. The method attribute specifies the service method to manage HTTP GET requests.
- Required business logic must be defined inside a service method.
- On the basis of the business logic, you can develop a model within this method. You can use setter different model attributes. These attributes will be used by the view to generate the result. The given example creates a model with the "message" attribute.
- A defined service method can return a String containing the name of the view to be used to render the model. "hello" is returned as a logical view name in this example.

@Controller	Indicates the class as a Spring MVC controller.
@RequestMapping	Specifies that all handling methods on this controller are relative to the /hello path.
@RequestMapping(method = RequestMethod.GET)	Is used to declare theprintHello() method as the controller's default service method to handle HTTP GET request.
You can define another method to handle any POST request at the same URL.	

You can write the previous controller in another form where you can add additional attributes in **@RequestMapping** as follows:

```
@Controller
public class HelloController {
   @RequestMapping(value = "/hello", method = RequestMethod.GET)
    public String printHello(ModelMap model) {
        model.addAttribute("message", "Hello Spring MVC Framework!");
        return "hello";
    }
}
```

Steps to Develop a Simple Spring MVC Application (Cont.)

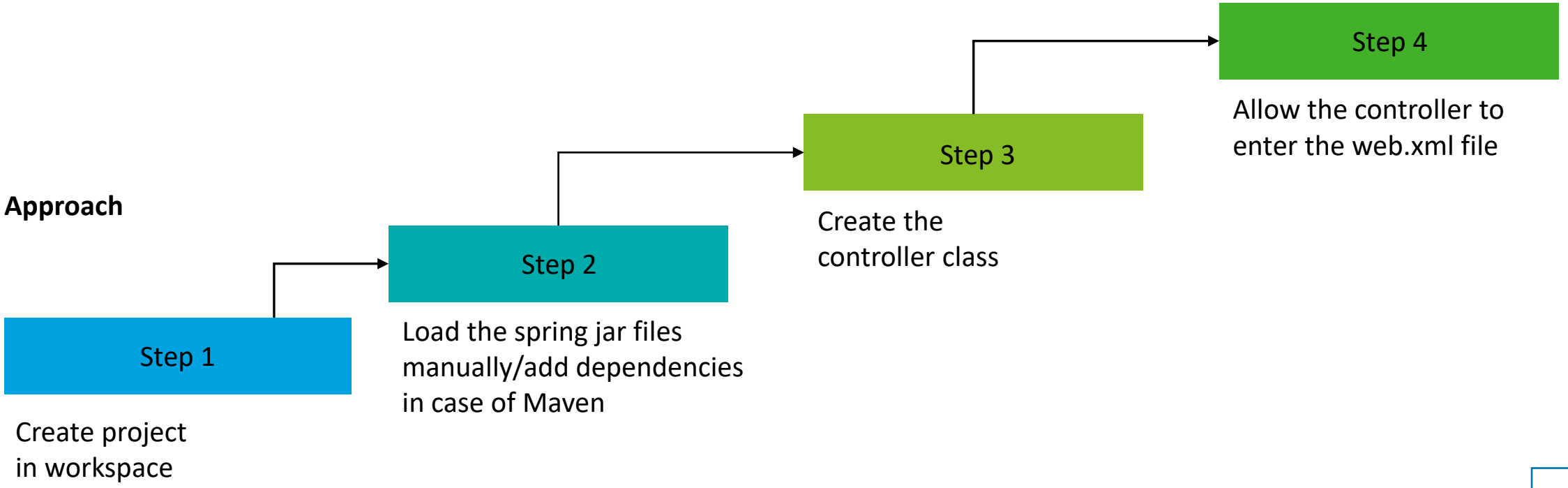
< / >

UseCase

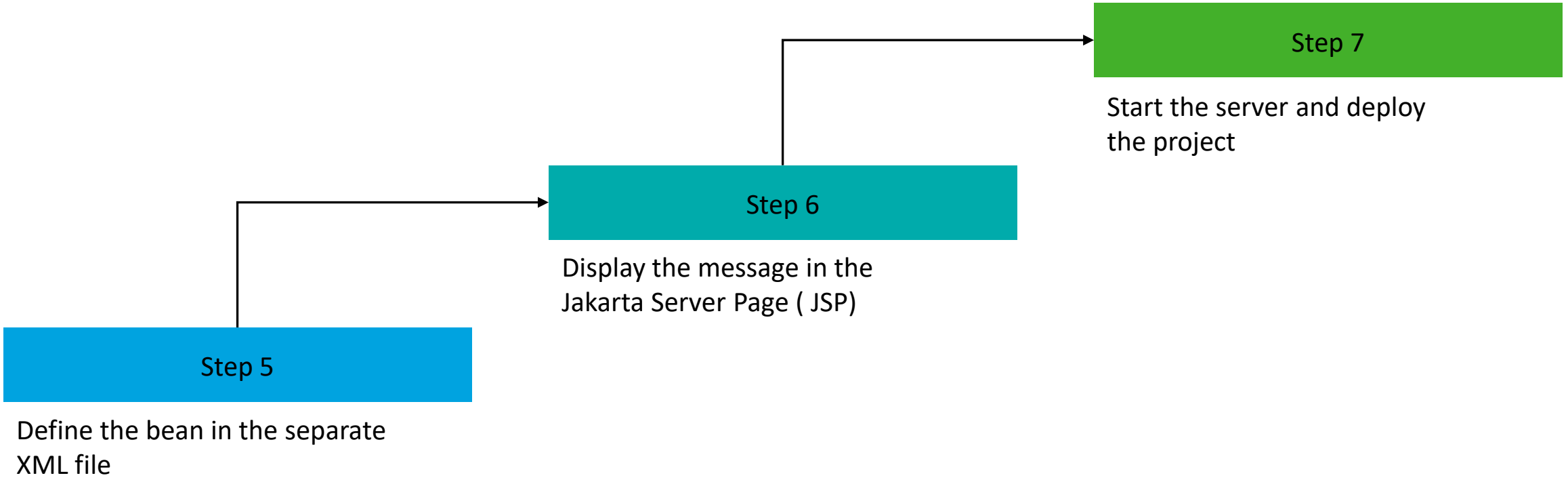
Develop a Simple Spring MVC Application that displays output in the browser
Required JAR files(add manually)/Maven dependencies:

- Spring Core jar files
- Spring Web jar files
- JSP + JSTL jar files (depends on view technology)

Approach



Steps to Develop a Simple Spring MVC Application (Cont.)



Steps to Develop a Simple Spring MVC Application (Cont.)

</ />

1. Configure pom.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.example</groupId>
  <artifactId>SpringMVC</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>SpringMVC Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>

<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>

  </dependencies>
  <build>
    <finalName>SpringMVC</finalName>
  </build>
</project>
```

2. Create Controller Class

To create the controller class, we are using two annotations:

- **@Controller**: Used to mark this class as Controller.
- **@RequestMapping:** Used to map the class with the specified URL name.

```
package com.example;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class HelloController {
@RequestMapping("/")
    public String display()
    {
        return "index";
    }
}
```

Steps to Develop a Simple Spring MVC Application (Cont.)

</ />

3. Controller Entry in web.xml

- Specify the servlet class DispatcherServlet that acts as the front controller
- All the incoming request for the html file will be forwarded to the DispatcherServlet.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://java.sun.com/xml/ns/javaee"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         id="WebApp_ID" version="3.0">
  <display-name>SpringMVC</display-name>
   <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>
          org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

4. Define Bean in xml file

- Specify the View components in this important configuration file.
- The context:component-scan element defines where DispatcherServlet searches Controllers.

```
<?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.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <!-- Provide support for component scanning -->
    <context:component-scan base-package="com.example" />

    <!--Provide support for conversion, formatting and validation -->
    <mvc:annotation-driven/>

</beans>
```

Steps to Develop a Simple Spring MVC Application (Cont.)

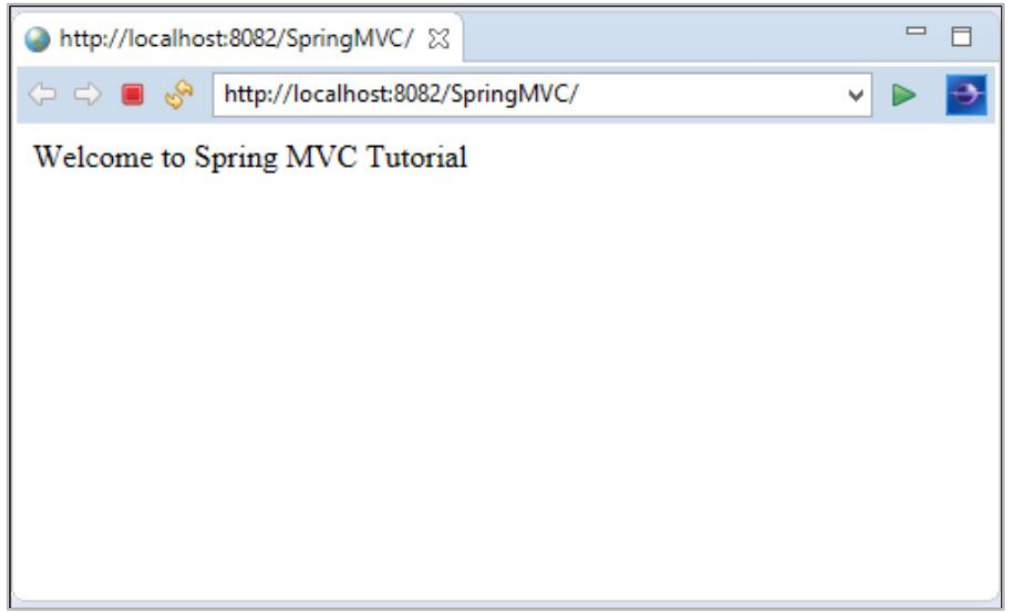
</>

5. Create JSP page

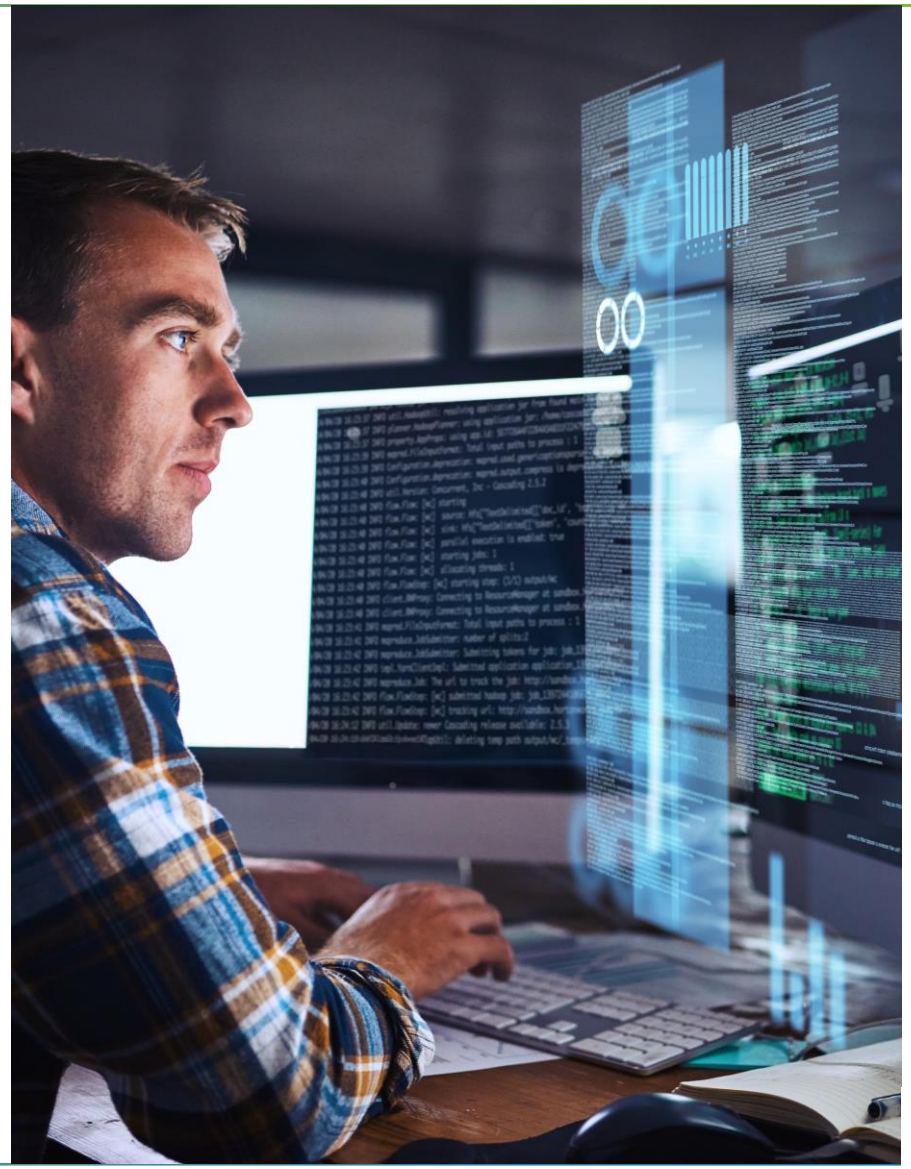
This is the simple JSP page, displaying the message returned by the Controller:

```
<html>
    <body>
        <p>
            Welcome to Spring MVC Tutorial
        </p>
    </body>
</html>
```

6. Output



Spring Handler—Adapters and Mappings



Handler Adapters

</>

- It is basically an interface that facilitates the handling of HTTP requests in a very flexible manner in Spring MVC.
- It's used along with HandlerMapping that maps a method to a specific URL.
- The DispatcherServlet then uses a HandlerAdapter to call this method.
- The Dispatcher servlet doesn't call the method directly—it serves as a bridge between itself and the handler objects resulting in a loose coupling design.
- The methods available in this HandlerAdapter interface are:
  - `handle()`
  - `getLastModified()`

```
public interface HandlerAdapter {
    boolean supports(Object handler);

    ModelAndView handle(
      HttpServletRequest request,
      HttpServletResponse response,
      Object handler) throws Exception;

    long getLastModified(HttpServletRequest request, Object handler);
}
```

- You can use the supports API to check whether a specific handler instance is supported.
- Call this method calling the handle() method of this interface to check whether the handler instance is supported or not.
- The handle API is used to manage a specific HTTP request. This method's job is to invoke the handler by sharing the HttpServletRequest and HttpServletResponse objects as parameters.
- Then, the handler executes the application logic and returns a ModelAndView object. It is then processed by the DispatcherServlet.

Handler Adapters (Cont.)

</>

Maven Dependency

- Maven dependency that needs to be added to pom.xml

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>4.3.4.RELEASE</version>
</dependency>
```

Types of Handler Adapter

</ />

SimpleControllerHandlerAdapter

- Default handler adapter registered by Spring MVC
- Deals with classes implementing the Controller interface and is used to forward a request to a controller object

SimpleServletHandlerAdapter

Allows any Servlet to work with DispatcherServlet to handle the request. It forwards the request from DispatcherServlet to the appropriate Servlet class by calling its service() method.

AnnotationMethodHandlerAdapter

This executes methods annotated with @RequestMapping annotation.

RequestMappingHandlerAdapter

AnnotationMethodHandlerAdapter was deprecated and replaced by RequestMappingHandlerAdapter from Spring 3.1. It's used with RequestMappingHandlerMapping class that runs methods annotated with @RequestMapping.

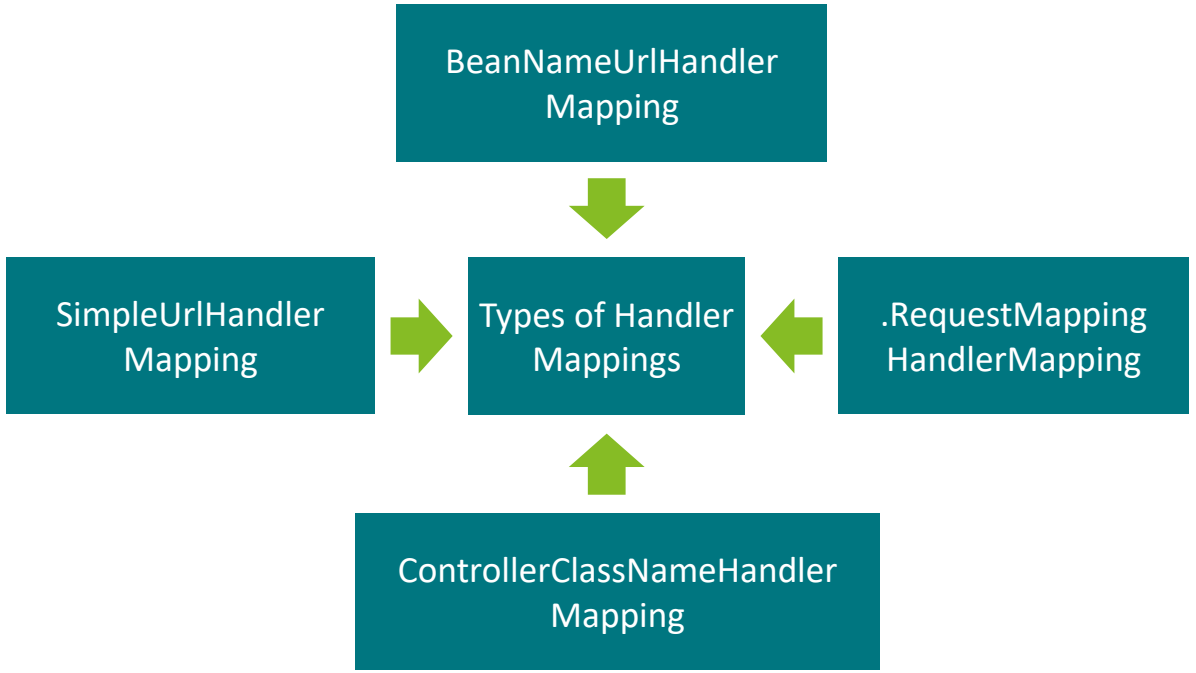
HttpRequestHandlerAdapter

This handler adapter is used to process HttpRequests.

Handler Mappings

</ />

- In Spring MVC, the DispatcherServlet works as front controller—receiving all incoming HTTP requests and processing them.
- The processing occurs when requests are passed to the relevant component using handler mappings.
- HandlerMapping is an interface that defines a mapping between requests and handler objects.
- The handler execution chain consists of the handler that matches the incoming request, and optionally, the list of interceptors applied for the request.
- Dispatcher servlet then runs the handlers and any associated handler interceptor.
- All handler mappings classes implement the interface: `org.springframework.web.servlet.HandlerMapping`



Types of Handler Mappings

< / >

BeanNameUrlHandlerMapping

- By default, DispatcherServlet uses:
  - BeanNameUrlHandlerMapping and DefaultAnnotationHandlerMapping
- How it works:
  - This implementation of handler mapping matches the URL of the incoming request with the name of the controller beans.
  - The matching bean is then used as the controller for the request.
  - This is the default handler mapping used by the Spring's MVC module, i.e., if the dispatcher servlet doesn't find any handler mapping bean defined in the context of Spring's application, then it uses BeanNameUrlHandlerMapping.
  - An example of pattern mapping is mapping requests to "/foo*" to beans with names starting with "/foo" like "/foo2/" or "/fooOne/."

Types of Handler Mappings (Cont.)

</>

- Let us assume we have three web pages in our application. The URL of the pages:
  - http://servername:portnumber/ApplicationContext/welcome.htm
  - http://servername:portnumber/ApplicationContext/listBooks.htm
  - http://servername:portnumber/ApplicationContext/displayBookContent.htm

- Define controllers in Spring's application context such that the name of the controller matches the URL of the request. XML configuration file

```
- net.example.frameworks.spring.mvc.controller.WelcomeController
- net.example.frameworks.spring.mvc.controller.ListBooksController
- net.example.frameworks.spring.mvc.controller.DisplayBookTOCController
```

Types of Handler Mappings (Cont.)

< / >

SimpleUrlHandlerMapping

- The BeanNameUrlHandlerMapping puts a restriction on controller bean names so that they match incoming request's URL.
- SimpleUrlHandlerMapping removes the restriction and maps the controller beans to request URL through property "mappings." It is the most flexible HandlerMapping implementation.
  - key of the <prop> element is the URL pattern of the incoming request.
  - value of the <prop> element is the name of the controller bean, which will perform the business logic to fulfill the request.
- SimpleUrlHandlerMapping is one of the simplest and most commonly used handler mappings which allows you to specify URL pattern and handler explicitly
- It enables direct and declarative mapping between bean instances and URLs or between bean names and URLs.
- SimpleUrlHandlerMapping has a property called mappings. We pass the URL pattern to it.
- Eg: It maps requests "/simpleUrlWelcome" and "/*/simpleUrlWelcome" to the "welcome" bean:

```
<bean
 id="myHandlerMapping"
 class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
   <property name="mappings">
      <props>
         <prop key="/welcome.htm">welcomeController</prop>
         <prop key="/listBooks.htm">listBooksController</prop>
         <prop key="/displayBookTOC.htm">displayBookTOCController</prop>
      </props>
   </property>
</bean>
<bean name="welcomeController"
 class="net.codejava.frameorks.spring.mvc.controller.WelcomeController"/>
<bean name="listBooksController"
 class="net.codejava.frameorks.spring.mvc.controller.ListBooksController"/>
<bean name="displayBookTOCController"
 class="net.codejava.frameorks.spring.mvc.controller.DisplayBookTOCController"/>
```

Types of Handler Mappings (Cont.)

</>

RequestMappingHandlerMapping

- The RequestMappingHandlerMapping is used to maintain the request URI's mapping to the handler.
- Once the handler is in place, the DispatcherServlet sends the request to the suitable handler adapter, which then invokes the handlerMethod().

1

Let's define a simple controller class:

```
@Controller
public class RequestMappingHandler {

    @RequestMapping("/requestName")
    public ModelAndView getEmployeeName() {
        ModelAndView model = new ModelAndView("Greeting");
        model.addObject("message", “Abc");
        return model;
    }
}
```

2

There are 2 different ways of configuring this adapter depending on whether the application uses Java-based configuration or XML based configuration.

Let's look at the first way using Java configuration:

```
@ComponentScan("com.baeldung.spring.controller")
@Configuration
@EnableWebMvc
public class ServletConfig implements WebMvcConfigurer {
    @Bean
    public InternalResourceViewResolver jspViewResolver() {
        InternalResourceViewResolver bean = new InternalResourceViewResolver();
        bean.setPrefix("/WEB-INF/");
        bean.setSuffix(".jsp");
        return bean;
    }
}
```

3

If the application uses XML configuration, then there are two different approaches for configuring this handler adapter in web application context XML. Let us take a look at the first approach defined in the file *spring-servlet_RequestMappingHandlerAdapter.xml*:

```
<beans ...>
    <context:component-scan base-package="com.baeldung.spring.controller" />

    <bean
      class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"/>

    <bean
      class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"/>

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

Types of Handler Mappings (Cont.)

< / >

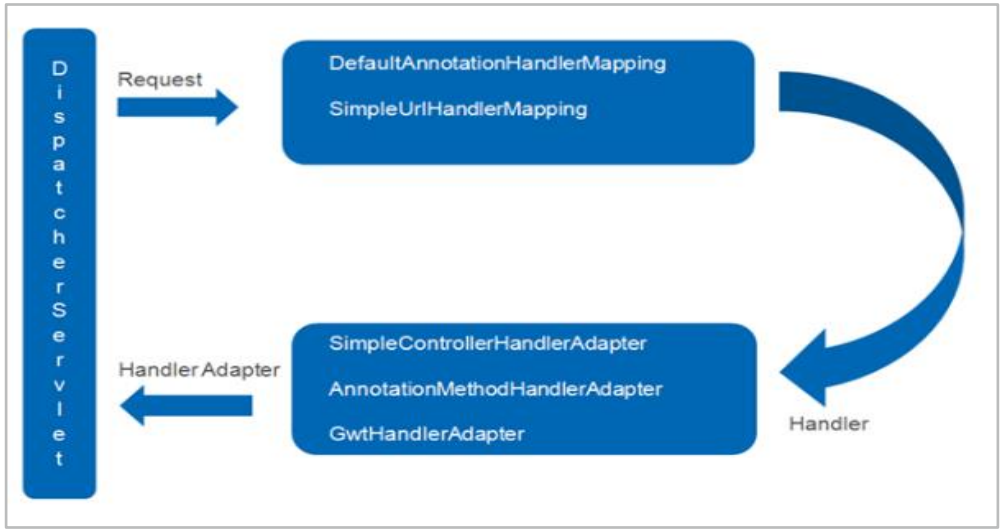
ControllerClassNameHandlerMapping(removed in Spring 5)

- The ControllerClassNameHandlerMapping maps the URL to a registered controller bean (or a controller annotated with the @Controller annotation) that has or starts with the same name.

Uses ControllerClassNameHandlerMapping

For example, “WelcomeController” would return as mapping to “/welcome*,” i.e., to any URL that starts with “welcome.”

HandlerMapping and HandlerAdapter



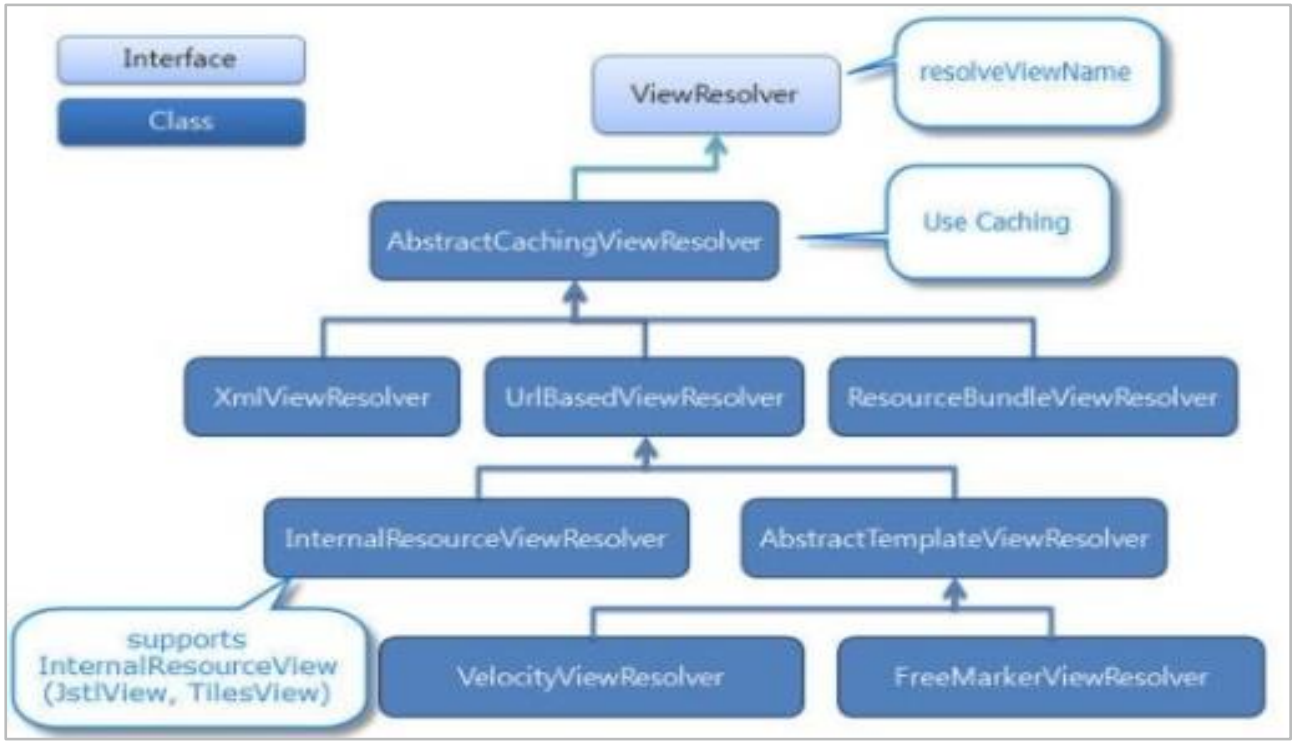
</ / >

View Resolvers

Overview

- In Spring MVC, view resolvers enable you to render models in a browser without linking the implementation to a particular view technology such as JSP, Velocity, XML, etc.
- There are two interfaces important to how Spring handles views: ViewResolver and View.
- The ViewResolver maps view names and actual views.
- The Spring framework comes with several view resolvers, e.g., InternalResourceViewResolver, XmlViewResolver, ResourceBundleViewResolver, and a few others.

View Resolver Hierarchy



View Resolver Hierarchy

< / >

AbstractCachingViewResolver	Abstract view resolver caches views. Views often need to be prepared before they can be used; extending this view resolver allows caching.
XmlViewResolver	Accepts configuration file which is written in XML with same DTD as XML bean factories, implement ViewResolver. /WEB-INF/views.xml is the default configuration file.
ResourceBundleViewResolver	Implement ViewResolver that uses bean definitions in a resource bundle specified by the bundle base name. Usually, we define the bundle in a properties file located in the classpath. views.properties is the default file name.
UrlBasedViewResolver	Simple ViewResolver interface which affects the resolution of view names which are logical to URLs without mapping definition. It is suitable if our logical names match the names of view resources.

View Resolver Hierarchy (Cont.)

InternalResourceViewResolver	Convenient subclass of UrlBasedViewResolver that supports InternalResourceView (in effect, Servlets and JSPs) and subclasses such as JstlView and TilesView. You can specify the view class for all views generated by this resolver by using setViewClass(..).
VelocityViewResolver	Convenient subclass of UrlBasedViewResolver that supports VelocityView (in effect, Velocity templates) or FreeMarkerView, respectively, and custom subclasses of them.
ContentNegotiatingView Resolver	Implementation of the ViewResolver interface that resolves a view based on the request file name or Accept header.

View Resolver Hierarchy (Cont.)

< / >

InternalResourceViewResolver

- The InternalResourceViewResolver is:
  - Implementation of ViewResolver interface
  - Extends UrlBasedViewResolver class
- Maps the jsp, servlet and jstl. It uses prefixes and suffixes to prepare the final view page URL, configured in the *-servlet.xml file.

How to use:
Copy this entry inside *-servlet.xml file

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

- Now, all the views names returned from your controller class will map inside WEB-INF/pages with suffix as .jsp.
- It's recommended to put all your view components inside the WEB-INF folder for security purposes.
- Putting viewing components inside the WEB-INF folder will hide them to direct access from the URL and allow only the controller to access the viewing components.

View Resolver Hierarchy (Cont.)

< / >

XmlViewResolver

- The XmlViewResolver is also an
  - Implementation of ViewResolver interface
  - Uses bean definitions from the dedicated XML file.
- XML file
  - Default Name: views.xml
  - Default location: class path
- You can map your view name with views inside views.xml file. However, XmlViewResolver will not support internationalization.

How to use:
Create views.xml file inside class path with below mapping entries

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="home"
        class="org.springframework.web.servlet.view.JstlView">
        <property name="url" value="/WEB-INF/pages/xml-home.jsp" />
    </bean>
</beans>
```

Now add below entry into your *-servlet.xml file

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

View Resolver Hierarchy (Cont.)

</>

ResourceBundleViewResolver

- The ResourceBundleViewResolver is an
  - Implementation of ViewResolver interface
  - Uses bean definition from ResourceBundle specified by the bundle basename.
- Bundle defined in a properties file,
  - Default bundle basename: views.properties.
  - Default location is in class path
- Will also
  - helps to achieve Internationalization (it can potentially be adapted to various languages and regions without engineering changes)
  - to return excel/pdf file as a view instead of jsp, jstl

How to use:

Create a views.properties file in your class path with the given entries to return views in an Excel file format instead of jsp, jstl.

```
excel.(class)=com.javamakeuse.springmvc.util.ExcelBuilderForResourceBundleViewResolver
```

Now add below entry in your *-servlet.xml file

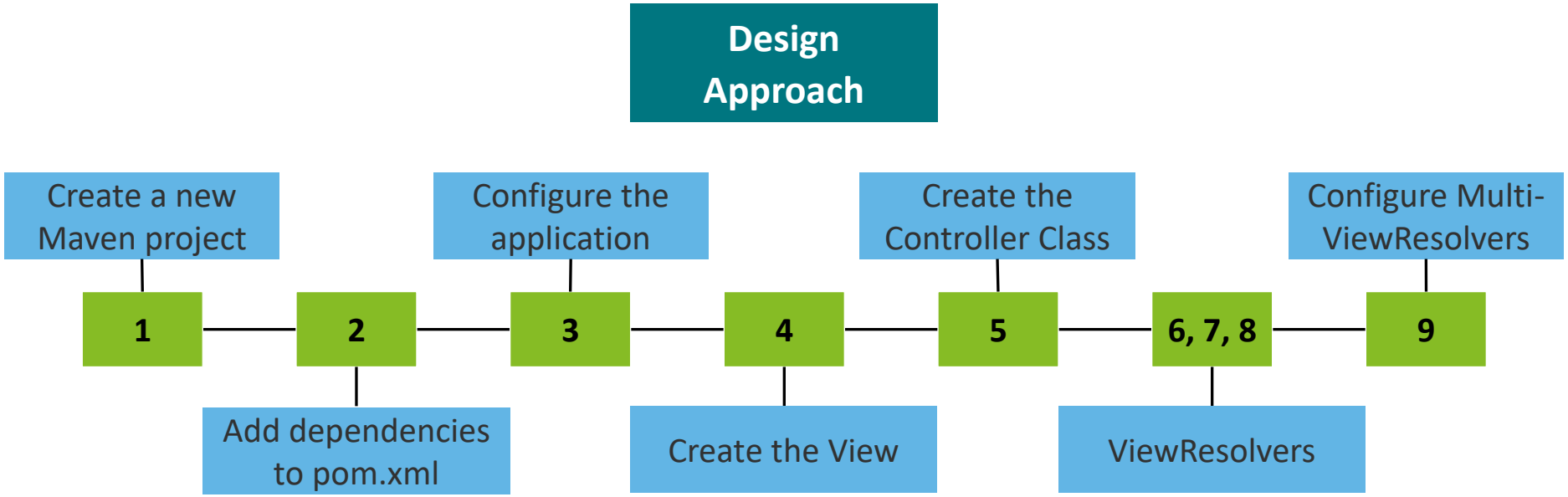
```
<bean id="viewResolver"
  class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
 <property name="basename" value="views" />
</bean>
```

Demo

</>

USE CASE

Create a Spring MVC application that shows implementation of ViewResolver

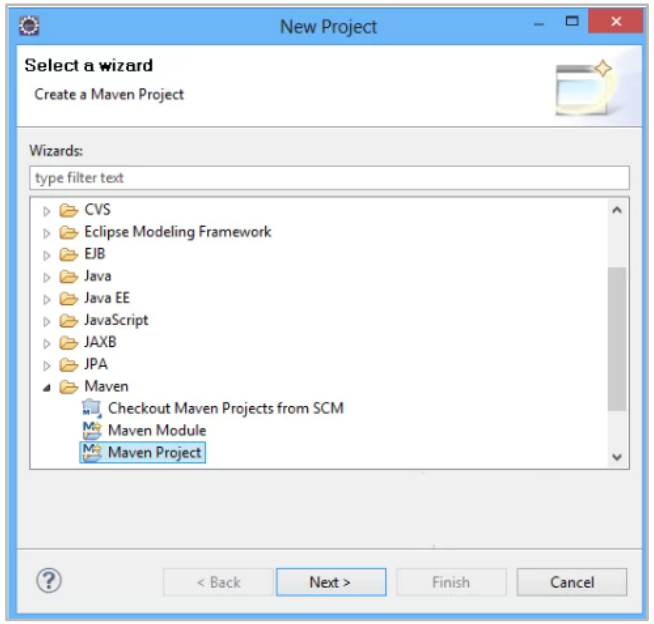


Demo (Cont.)

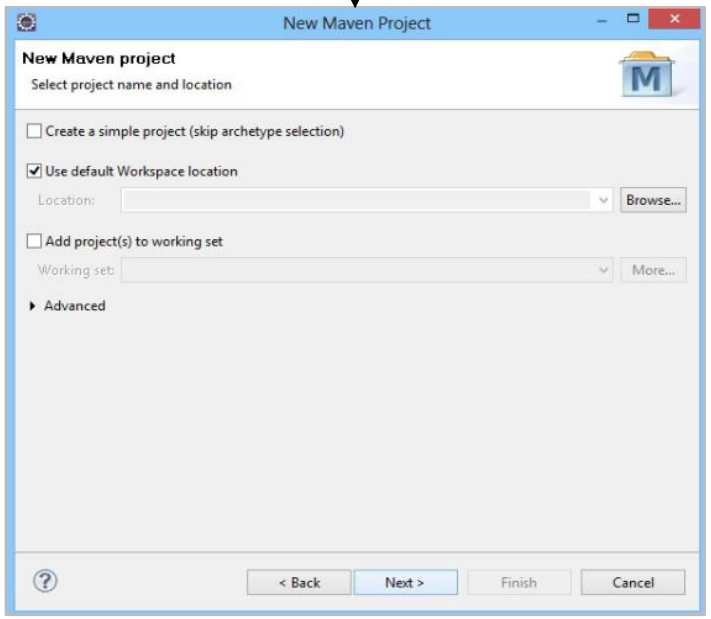
</ />

Create a New Maven Project

Go to File -> Project -> Maven -> Maven Project



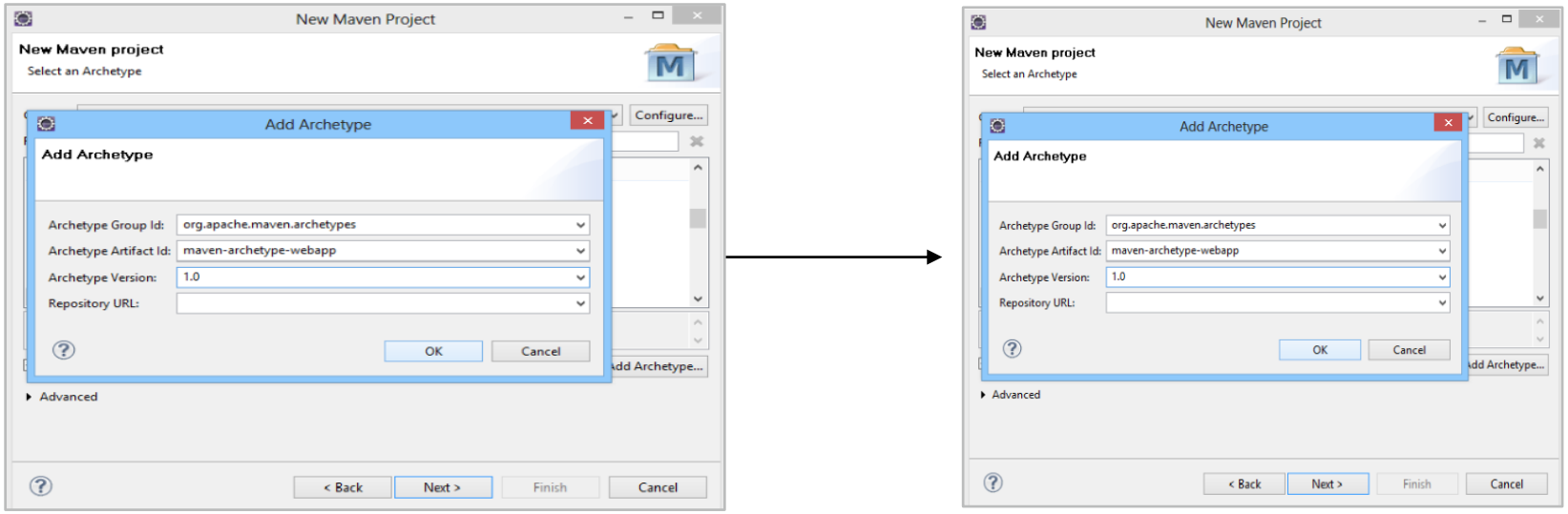
In the wizard's "Select project name and location" page, be sure that this option is unchecked: Create a simple project (skip archetype selection). Then, click "Next" to continue with default values.



Demo (Cont.)

</ />

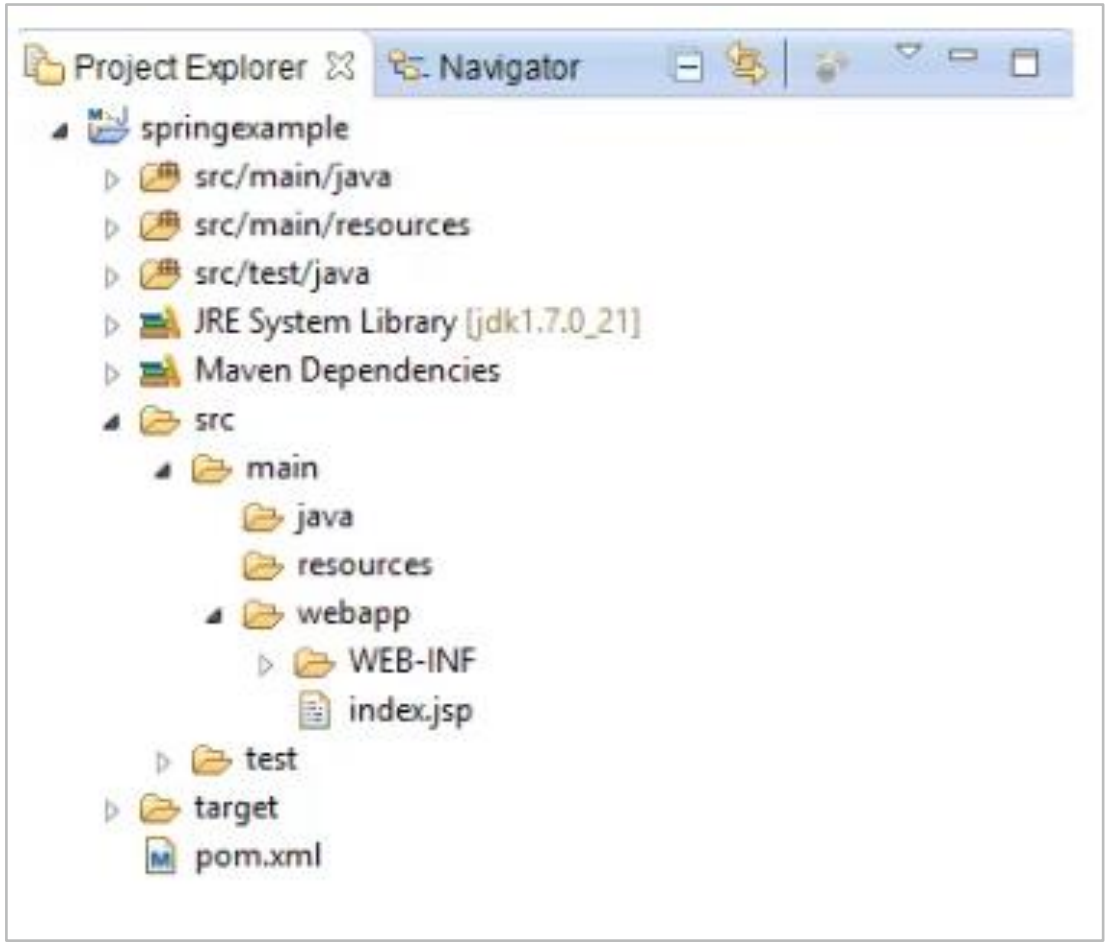
- Provide the name and main package of your project.
- Provide “Group Id” and “Artifact Id” values.
- Project package will be <Group Id value>.<Artifact Id value> and the project name <Artifact Id value>
- “Package” variable set to "war" so that a war file will be created to be deployed onto the server (tomcat).
- Click “Finish” to close the wizard and to create your project.



Demo (Cont.)

</ >

/src/main/java folder	Location to store source files for the dynamic content of the application
/src/test/java	Stores configurations files
/src/main/resources	Location for all source files for unit tests
/target	Location for the compiled and packaged deliverables
/src/main/resources/webapp/WEB-INF	Location for deployment descriptors for the Web application
pom.xml - project object model (POM)	Contains all project-related configurations.



Demo (Cont.)

</>

Add Spring-MVC dependencies

- Add the dependencies in Maven's pom.xml file by modifying it on the "Pom.xml" page of the POM editor.
- The dependency needed for MVC is the spring-web MVC package.

pom.xml

```
01  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLScher
02    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.
03    <modelVersion>4.0.0</modelVersion>
04    <groupId>com.javacodegeeks.snippets.enterprise</groupId>
05    <artifactId>springexample</artifactId>
06    <packaging>war</packaging>
07    <version>0.0.1-SNAPSHOT</version>
08    <name>springexample Maven Webapp</name>
09    <url>http://maven.apache.org</url>
10    <dependencies>
11
12          <dependency>
13              <groupId>org.springframework</groupId>
14              <artifactId>spring-webmvc</artifactId>
15              <version>${spring.version}</version>
16          </dependency>
17          <dependency>
18              <groupId>javax.servlet</groupId>
19              <artifactId>servlet-api</artifactId>
20              <version>2.5</version>
21          </dependency>
22
23    </dependencies>
24    <build>
25      <finalName>springexample</finalName>
26    </build>
27
28      <properties>
29          <spring.version>3.2.3.RELEASE</spring.version>
30      </properties>
31  </project>
```

Dependencies

Demo (Cont.)

</>

Configure the application

- The files that we must configure in the application are the
  - web.xml file
  - mvc-dispatcher-servlet.xml file
- The web.xml file:
  - Defines everything about the application that a server needs to know.
  - Stored in the/WEB-INF/ directory of the application.
- The <servlet> element declares the DispatcherServlet, and when it is initialized, the framework will try to load the application context from a file named [servlet-name]-servlet. xml is located in the /WEB-INF/ directory.
- The MVC-dispatcher-servlet.xml file is created, which will be explained below. The <servlet-mapping> element of the web.xml file specifies what URLs will be handled by the DispatcherServlet.

web.xml

```
01  <?xml version="1.0" encoding="UTF-8"?>
02  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xm
03
04      <servlet>
05          <servlet-name>mvc-dispatcher</servlet-name>
06          <servlet-class>
07              org.springframework.web.servlet.DispatcherServlet
08          </servlet-class>
09          <load-on-startup>1</load-on-startup>
10      </servlet>
11
12      <servlet-mapping>
13          <servlet-name>mvc-dispatcher</servlet-name>
14          <url-pattern>/</url-pattern>
15      </servlet-mapping>
16  </web-app>
```

Demo (Cont.)

< / >

- The mvc-dispatcher-servlet.xml file is saved in the WebContent/WEB-INF directory as well.
- All the beans are created in this file, for example Controllers will be placed and defined here.
- So, the HelloWorldController (the controller of our application) is defined here and will be shown in the next steps.
- The <context:component-scan> tag is used to allow the container to know where to look for the classes.

Demo (Cont.)

</>

Create the View

- The view is a simple jsp page, placed in /WEB-INF/ folder.
- It shows the value of the attribute that was set to the Controller.

helloWorld.jsp

```
<html>
<body>
    <h1>Spring 3.2.3 MVC view resolvers example</h1>

    <h3> ${msg}</h3>
</body>
</html>
```

Create the Controller

The HelloWorldController extends the AbstractController provided by Spring, and overrides the handleRequestInternal(HttpServletRequest request, HttpServletResponse response) method, where a org.springframework.web.servlet.ModelAndView is created by a handler and returned to be resolved by the DispatcherServlet.

HelloWorldController.java

```
package com.deloitte.test;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;
public class HelloWorldController extends AbstractController{
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        ModelAndView model = new ModelAndView("helloWorld");
        model.addObject("msg", "hello world!");
return model;
}
```

Demo (Cont.)

</>

InternalResourceViewResolver

- The jsp and html files are mapped by InternalResourceViewResolver in the WebContent/ WEB-INF/ folder.
- It enables us to define properties such as prefix or suffix to the view name to create the final view page URL.
- It is configured as shown here in mvc-dispatcher-servlet.xml.
- When the Controller returns the "helloworld" view, the InternalResourceViewResolver generates the URL of the view by using the prefix and suffix properties set and maps the "helloworld" view name to the correct "helloworld" view.

mvc-dispatcher-servlet.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springfr

    <context:component-scan base-package="com.deloitte.test"/>

   <bean class="com.deloitte.test.HelloWorldController"/>

 <bean

class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerM
apping" I>
  <bean

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

</beans>
```

Demo (Cont.)

</>

XmlViewResolver

- XmlViewResolver is an implementation of ViewResolver that accepts a configuration file written in XML, where the view implementation and the URL of the JSP file are set. Below is the configuration file, views.xml.
- The resolver is defined in MVC-dispatcher-servlet.xml. It provides a property to configure, which is the location property, and the path of the configuration file is set there.

views.xml

```
01  <beans xmlns="http://www.springframework.org/schema/beans"
02      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03      xsi:schemaLocation="http://www.springframework.org/schema/beans
04  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
05
06      <bean id="helloWorld"
07          class="org.springframework.web.servlet.view.JstlView">
08          <property name="url" value="/WEB-INF/helloWorld.jsp" />
09      </bean>
10
11  </beans>
```

XmlViewResolver (Cont.)

mvc-dispatcher-servlet.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springfr
     <context:component-scan base-package="com.deloitte.test"/>
    <bean class="com.deloitte.test.HelloWorldController"/>
   <bean
class="org.springframework.web.servlet.mvc.support.ControllerClassNam
eHandlerMapping“ />
  <bean class="org.springframework.web.servlet.view.XmlViewResolver"›
      <property name="location"›
          <value>/WEB-INF/views.xml</value>
      </property>
  </bean>
</beans>
```

- Now, when the Controller returns the “HelloWorld" view, the XmlViewResolver will use the views.xml file to get the view class and the view URL that will be mapped to the name “HelloWorld.”

Demo (Cont.)

</>

ResourceBundleViewResolver

- The ResourceBundleViewResolver uses bean definitions in a ResourceBundle, which is specified by the bundle basename.
- The bundle is usually defined in a properties file in the classpath.
- Below is the views.properties file: views.properties

```
– helloworld.(class)=org.springframework.web.servlet.view.JstlView
– helloworld.url=/WEB-INF/helloworld.jsp
```

- The ResourceBundleViewResolver is defined in MVC-dispatcher-servlet.xml, and in its definition, the basename property is set to view.properties file.
- So, in this case, when the Controller returns the “HelloWorld" view, the ResourceBundleViewResolver will use the views.properties file to get info about the view class and URL of the view that will be mapped to the name “HelloWorld.”

mvc-dispatcher-servlet.xml

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

<context:component-scan base-package="com.deloitte.test"/>

 <bean class="com.deloitte.test.HelloWorldController"/>

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

<bean class="org.springframework.web.servlet.view.
ResourceBundleViewResolver"›

<property name="basename" value="views" />
</bean>

</beans>
```

Demo (Cont.)

</>

Multiple View Resolvers together

- To set multiple Resolvers in the same configuration file, you can set the order property in all definitions so that the order they are used will be defined, as shown beside.
- Note that lowest priority is given to InternalResourceViewResolver as it can map any request to the correct view. If it is set before other resolvers, none of them will be used.

mvc-dtspatcher-servlet.xml

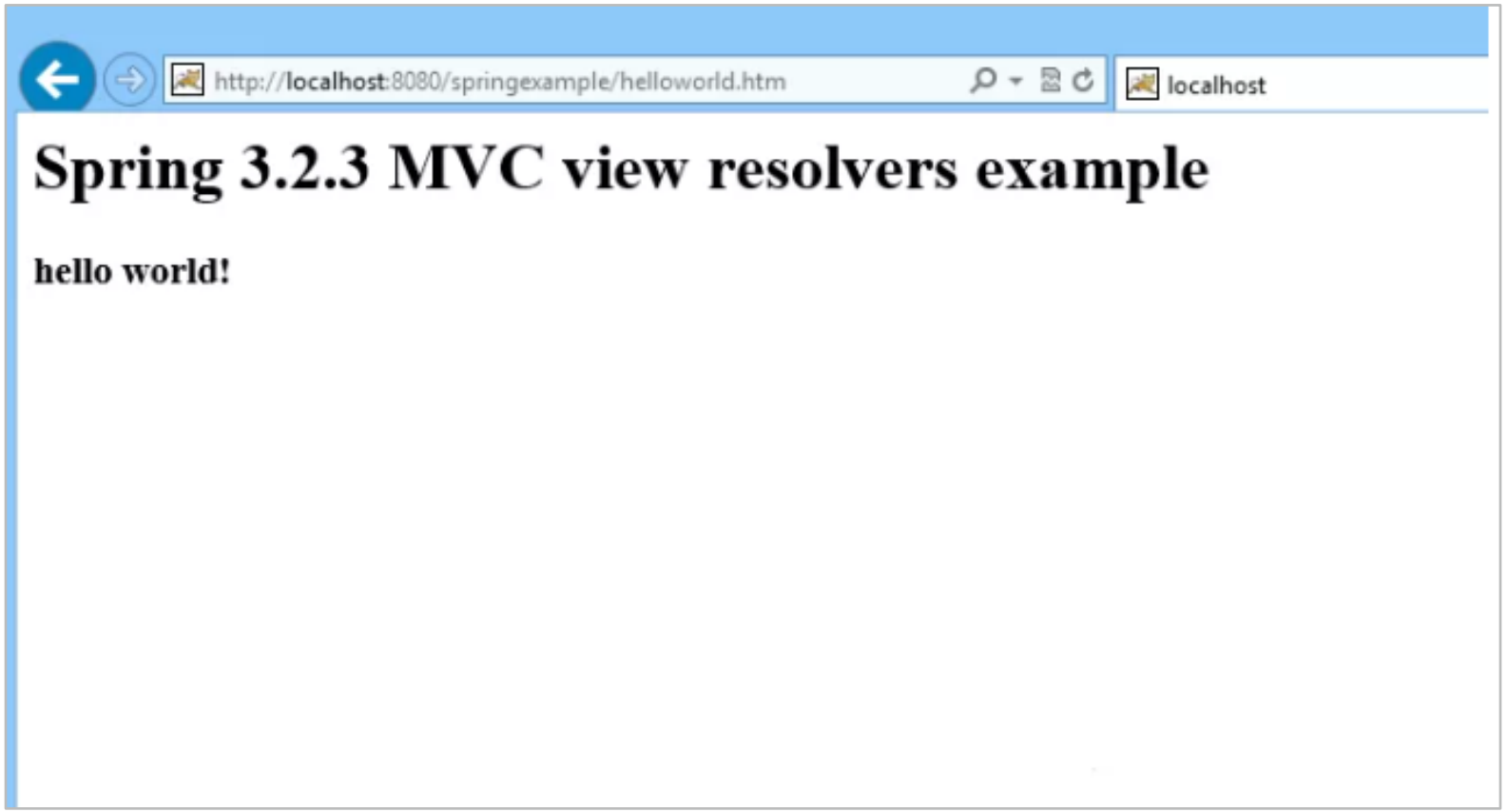
```
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:context="http://www.springfr
     <context:component-scan base-package="com. deloitte.test"/>
   <bean class="com. deloitte.test.HelloWorldController"/>
 <bean
     class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping" />
<bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver"›
    <property name="prefix"›
        <value>/WEB-INF/</value>
    </property>
    <property name="suffix"›
        <value>.jsp</value>
    </property>
    <property name="order" value="2" />
 </bean>
 <bean class="org.springframework.web.servlet.view.XmlViewResolver"›
    <property name="location"›
        <value>/WEB-INF/views.xml</value>
    </property>
    <property name="order" value="1" />
 </bean>
 <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver"›
     <property name="basename" value="views" />
     <property name="order" value="0" />
 </bean>
</beans>
```

Demo (Cont.)

</ />

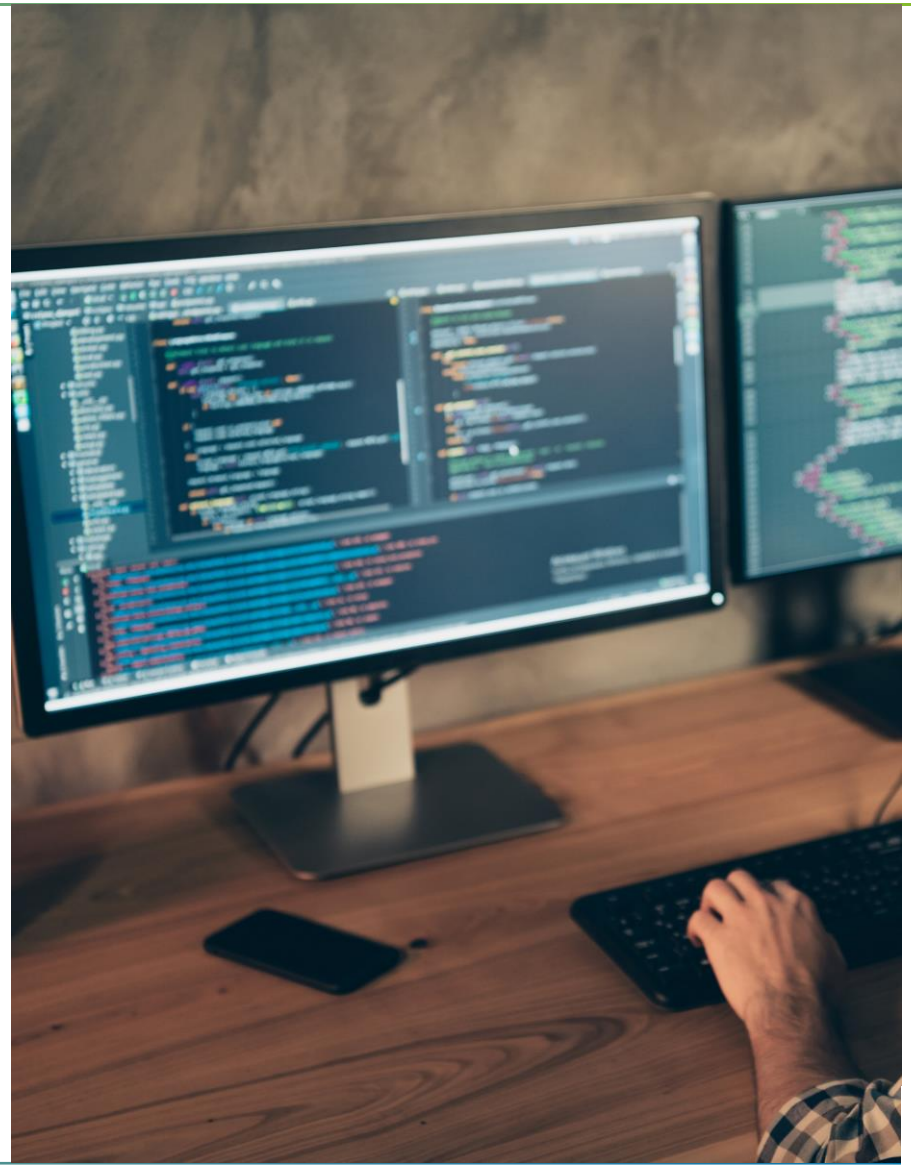
Output

You can execute your application using a tomcat server. The result will look like this:



</ / >

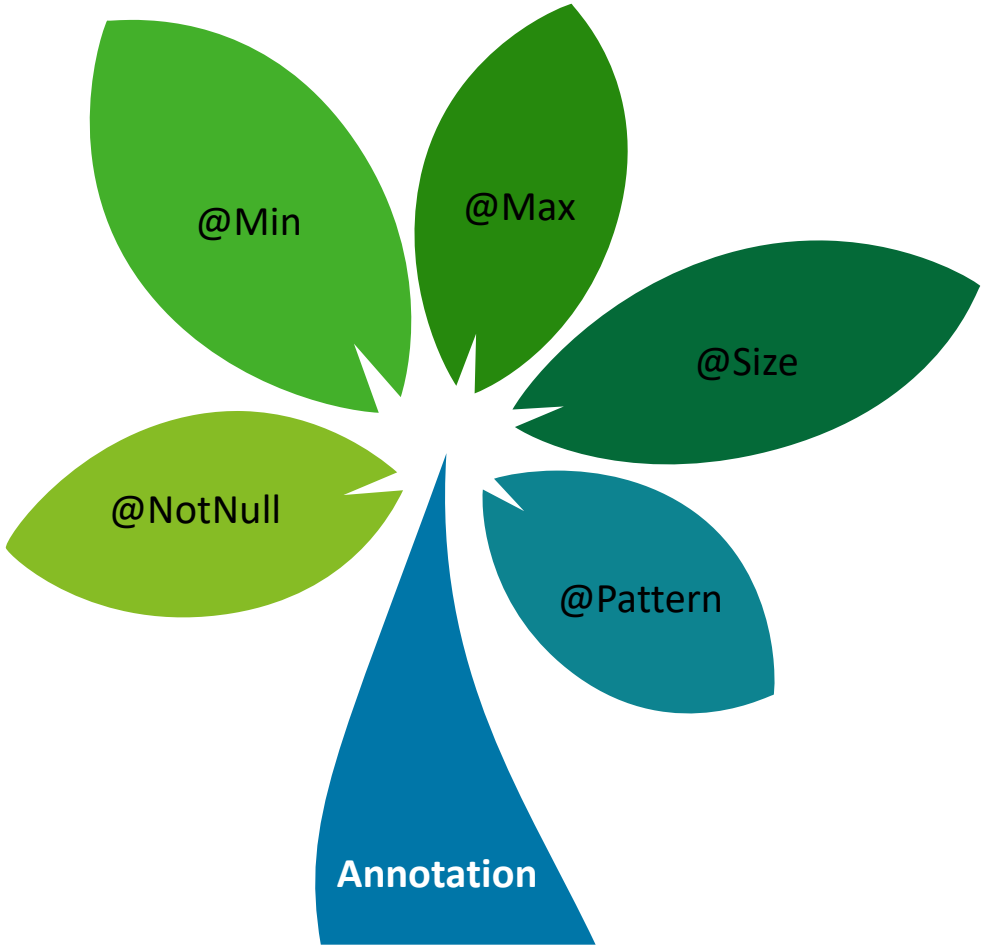
Spring Validation



Spring MVC Validation

Validation Annotations

- Below are some of the frequently used validation annotations.



@NotNull

It determines that the value cannot be null.

@Min

It determines that the number must be equal to or greater than the specified value.

@Max

It determines that the number must be equal to or less than the specified value.

@Size

It determines that the size must be equal to the specified value.

@Pattern

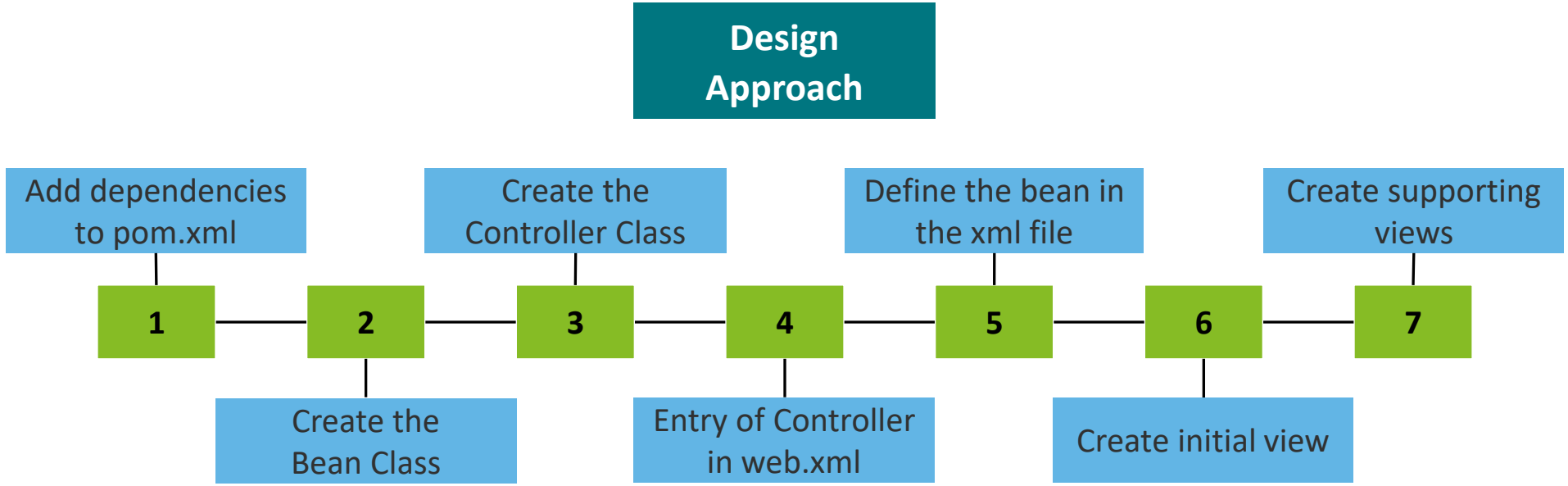
It determines that the sequence follows the specified regular expression.

Demo

< / >

USE CASE

Create a Spring MVC application with a Login form and implement Spring MVC validation for the form fields.



Demo (Cont.)

</>

Add dependencies to pom.xml

Highlighted the dependencies to be included in the below code snippet.

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-jasper</artifactId>
    <version>9.0.12</version>
</dependency>
    <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
  <!-- https://mvnrepository.com/artifact/org.hibernate.validator/hibernate-validator -->
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.13.Final</version>
</dependency>
```

Create the Bean Class

An Employee class with two properties for capturing Username and Password

```
package com.example;
import javax.validation.constraints.Size;

public class Employee {

    private String name;
    @Size(min=1,message="required")
    private String pass;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getPass() {
        return pass;
    }
    public void setPass(String pass) {
        this.pass = pass;
    }
}
```

Demo (Cont.)

</>

Create the Controller Class

In controller class:

- The @Valid annotation applies validation rules to the provided object.
- The BindingResult interface contains the result of validation.

```
package com.deloitte;
import javax.validation.Valid;
import org.springframemork.stereotype.Controller;
import org.springframemork.ui.Model;
import org.springframemork.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframemork.web.bind.annotation.Requestmapping;
@Controller
public class EmployeeController {
   @RequestMapping("/hello")
   public String display(Model m)
   {
       m.addAttribute("emp", new Employee());
       return "viewpage";
   }
   @RequestMapping("/helloagain")
   public String submitForm( @Valid @ModelAttribute("emp") Employee e, BindingResult br)
   {
}
       if(br.hasErrors())
       {
            return "viewpage";
       }
       else
       {
       return "final"; }
```

Entry of Controller in web.xml

- Once the DispatcherServlet is initialized, the framework will attempt to load the application context from a file named [servlet-name]-servlet.xml
- The <servlet-mapping> element of the web.xml file specifies what URLs will be handled by the DispatcherServlet.

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://java.sun.com/xml/ns/javaee"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         id="WebApp_ID" version="3.0">
  <display-name>SpringMVC</display-name>
   <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

Demo (Cont.)

</>

Define the bean in the xml file

spring-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.xsd
http://tant.springframework.org/schema/context
http://tane.springframework.org/schema/context/spring-
context.xsd
http://www.springfrasework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc.xsd'>
<!-- Provide support for component scanning ..>
<context:component-scan base-package="com.deloitte" />
<!--Provide support for conversion, formatting and validation
-->
<mvc:annotation-driven/>
<!-- Define Spring MVC view resolver -->
<bean id=“viewResolver“
class="org.springframework.web.servlet.view.InternalResource
ViewResolver"›
<property name="prefix value="/WEB-INF/jsp/'></property>
<property name."suffix" value=".jsp”></property>
</bean>
</beans>
```

Create initial view

index.jsp

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<body>
<a href="hello">Click here...</a>
</body>
</html>
```

Create supporting views

viewpage.jsp

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<head>
<style>
.error{color:red}
</style>
</head>
<body>
<form:form action="helloagain" modelAttribute="emp">
Username: <form:input path="name"/> <br><br>
Password(*): <form:password path="pass"/>
<form:errors path="pass" cssClass="error"/><br><br>
<input type="submit" value="submit">
</form:form>
</body>
</html>
```

final.jsp

```
<html>
<body>
Username: ${emp.name} <br><br>
Password: ${emp.pass}
</body>
</html>
```

Server-Side Validation

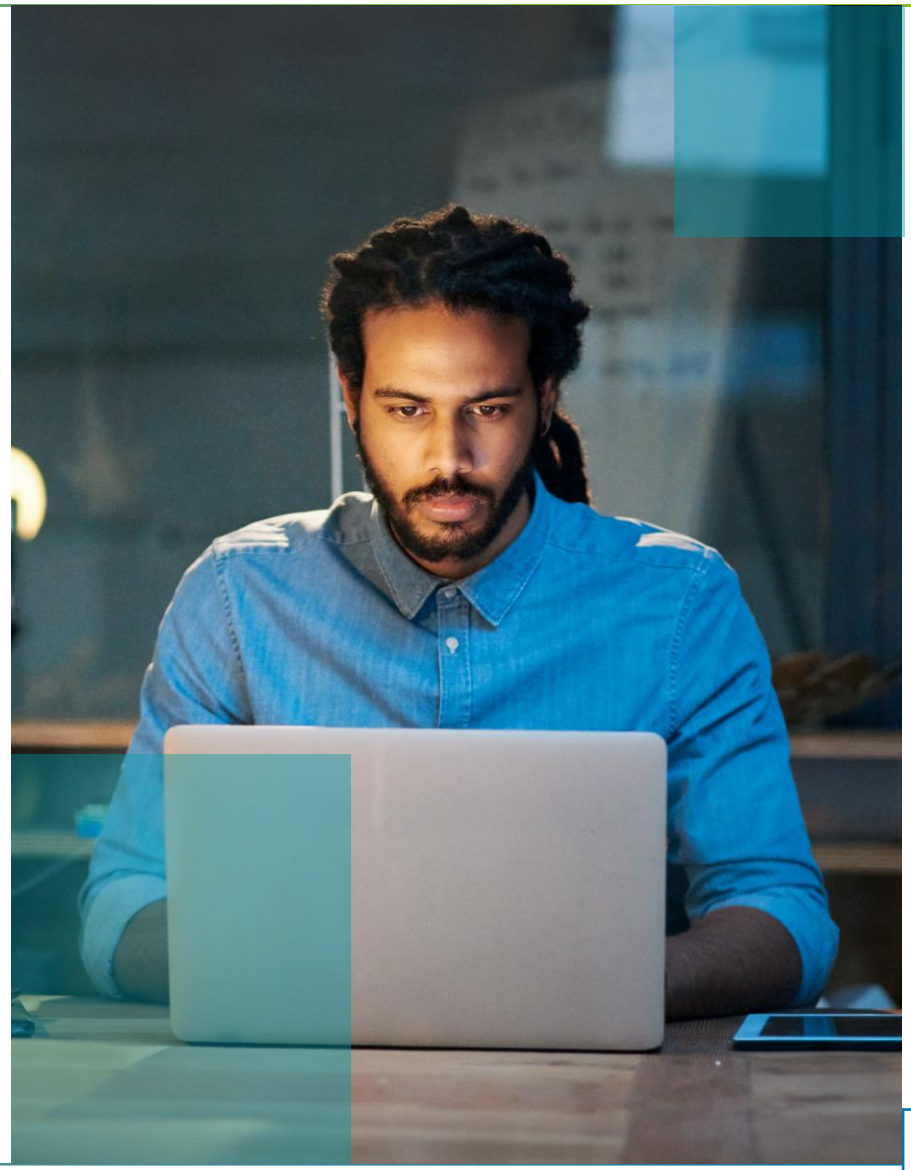
</ />

- When we accept user inputs in any web application, it becomes necessary to validate them. We can validate the user input on the client side using JavaScript. Still, it’s also necessary to validate them on the server side to make sure we are processing valid data in case the user has javascript disabled.
- Spring MVC Framework supports JSR-303(Java Specification Request) specs by default, and all we need is to add JSR-303 and its implementation dependencies in the Spring MVC application.
- Spring also provides @Valid annotation and BindingResult class through which we can get the errors raised by Validator implementation in the controller request handler method.
- We can create our custom validator implementations in two ways—First is to create an annotation that confirms the JSR-303 specs and implement its Validator class.
- The Second approach is to implement the org.spring framework. validation. Validator interface and add set it as a validator in the Controller class using @InitBinder annotation.

Bean Validation API

</ />

- The Spring MVC Validation is used to restrict the input provided by the user.
- To validate the user's input, the Spring 4 or higher version supports and uses Bean Validation API. It can validate both server-side as well as client-side applications.
- The Bean Validation API is a Java specification that applies constraints on object models via annotations.
- Here, we can validate a length, number, regular expression, etc. Apart from that, we can also provide custom validations.
- As Bean Validation API is just a specification, it requires an implementation.
- So, for that, it uses Hibernate Validator. The Hibernate Validator is a fully compliant JSR-303/309 implementation that allows to express and validate application constraints.



Summary

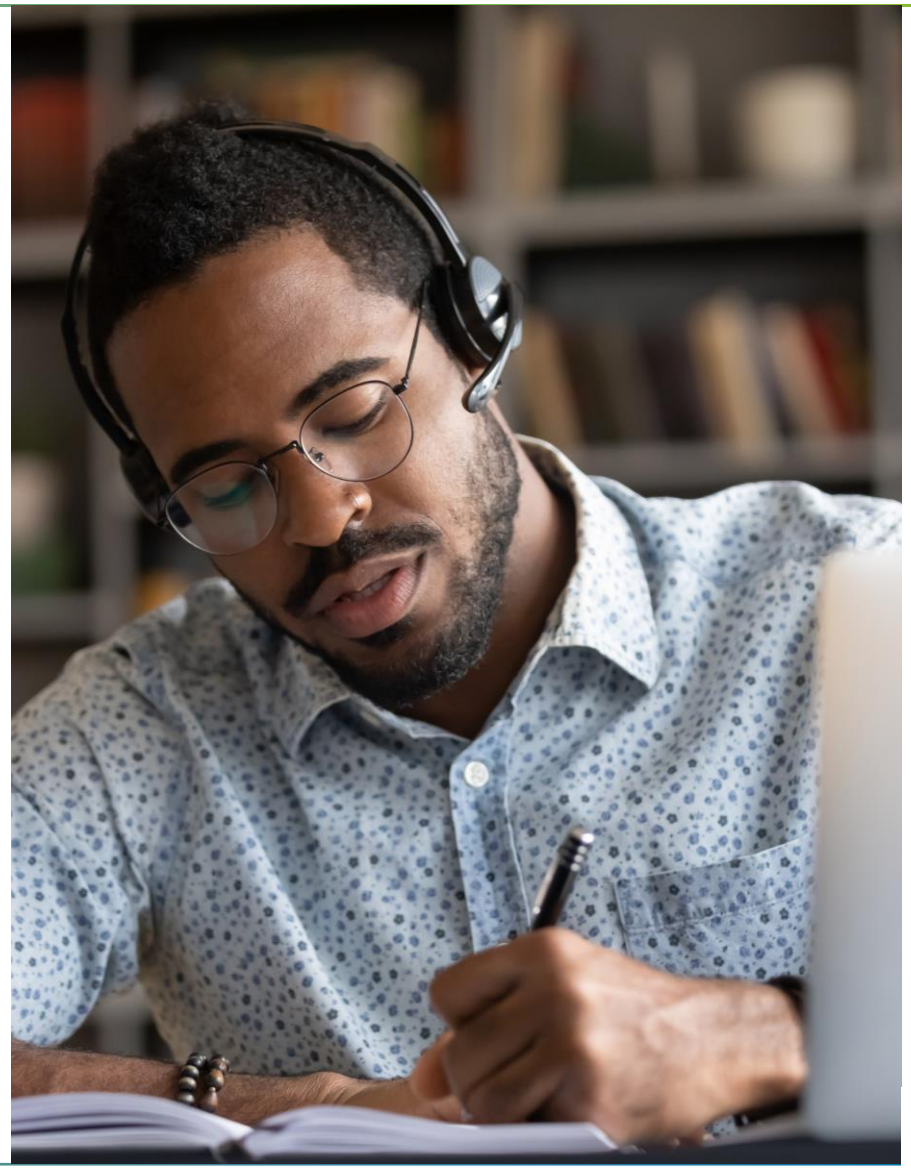
< / >

Here are the key learning points of the module.

- Spring MVC is a Java framework for building web applications and follows the Model-View-Controller Design Pattern.
- Stereotype annotations are markers for any class that fulfills a role within an application and helps remove the Spring XML configuration required for @Component, @Controller, @Service, and @Repository.
- Handler Adapter facilitates the handling of HTTP requests in a flexible way. The processing occurs by passing the requests to the relevant component with the help of handler mappings.
- With the use of view resolvers, models can be rendered in a browser without being restricted to the use of one particular view technology, such as JSP, Velocity, XML, etc.

Thank You

Hands-On Labs



Hands-On Activity 1

< / >

Activity Details	
Problem Statement	Create a Spring MVC Application which prints the name of the user which is hardcoded in the application on click of Login button on the home page.
Sample Input	Hard code the name, James in the Controller class and display the output, Welcome James !

Deloitte.