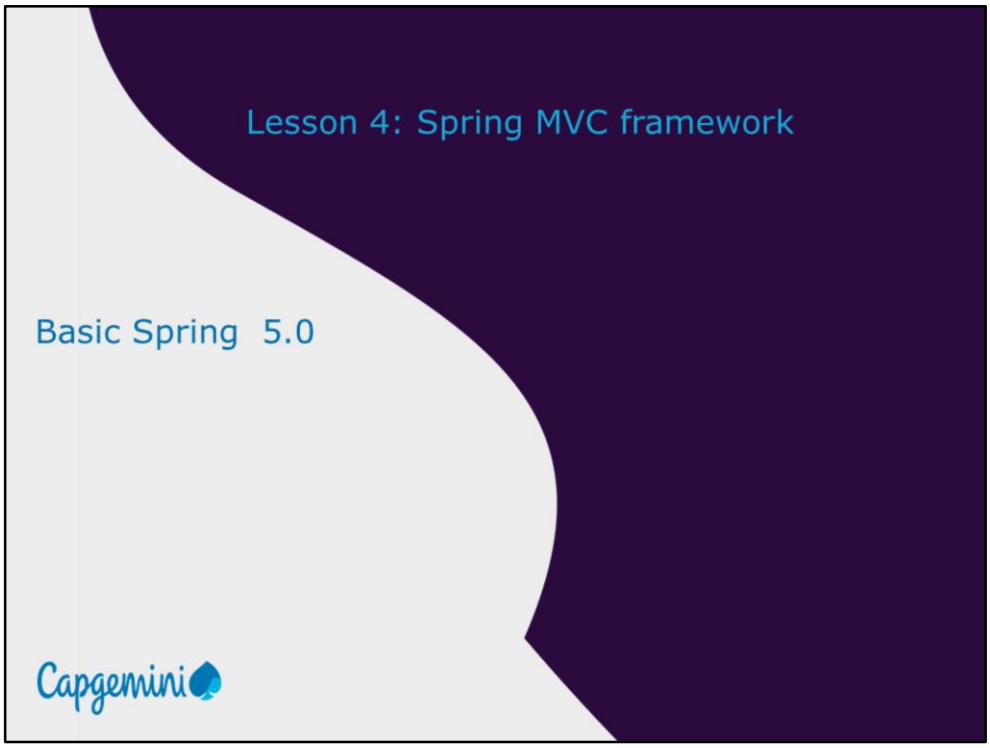
**Instructor Notes:**

Add instructor notes here.



**Instructor Notes:**

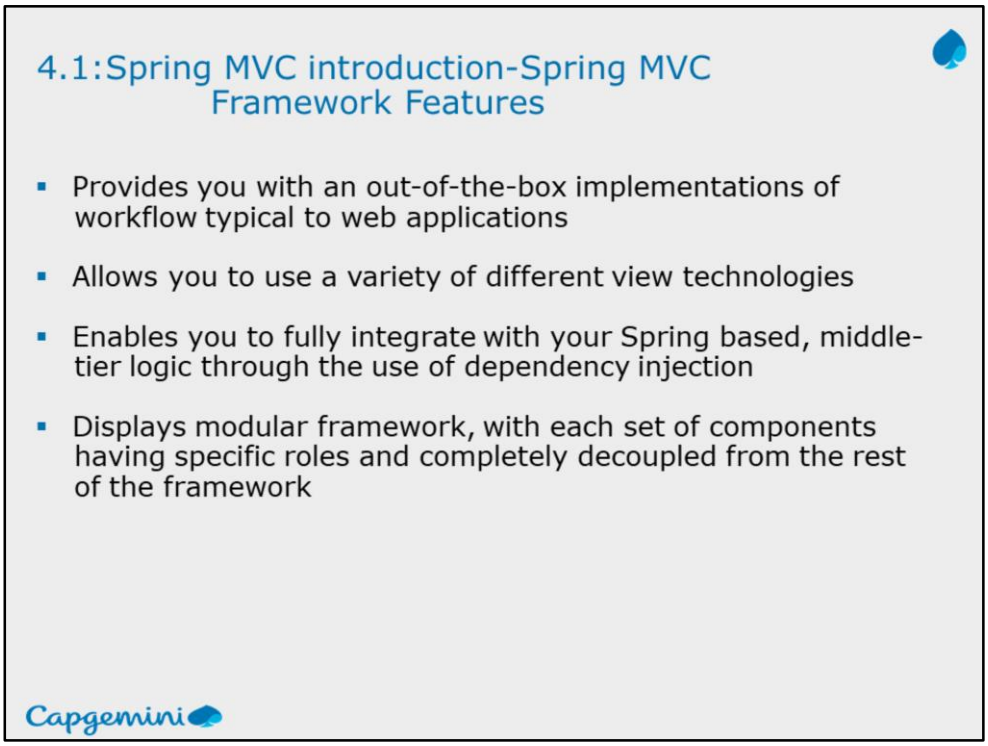
## Lesson Objectives

- Introduction to Spring MVC framework
- Learn how to develop web applications using Spring
- Understand the Spring MVC architecture and the request cycle of Spring web applications
- Understand components like handler mappings, ViewResolvers and controllers
- Use MVC Annotations like @Controller, @RequestMapping and @RequestParam
- Dispatcher Servlet Java Based Configuration
- MVC Annotations
- Java Based MVC Configuration

Capgemini

Web applications have become a very important part of any enterprise system. The key requirements for a web framework is to simplify development of the web tier as much as possible. Spring provides a web framework based on the MVC (Model view Controller) paradigm. Although it is similar in some ways to other popular MVC frameworks such as Struts and WebWork, Spring web MVC provides significant advantages over those frameworks.
Spring MVC helps in building flexible and loosely coupled web applications. The Model-view-controller design pattern helps in separating the business logic, presentation logic and navigation logic. Models are responsible for encapsulating the application data. The Views render response to the user with the help of the model object . Controllers are responsible for receiving the request from the user and calling the back-end services.

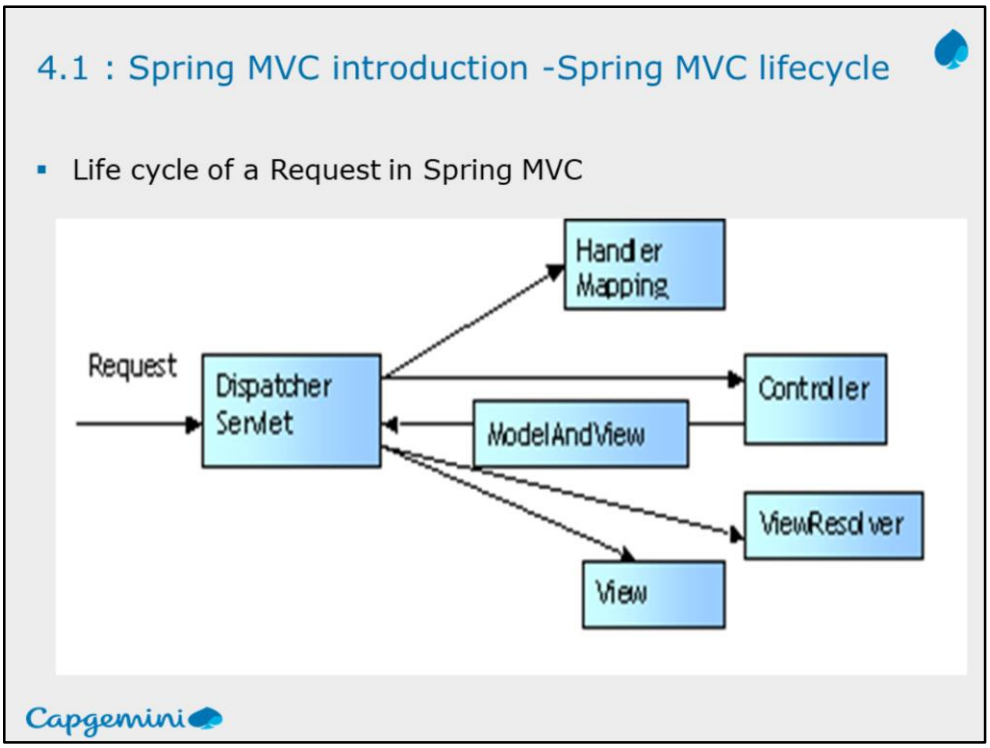
**Instructor Notes:**




The Spring MVC is a web framework built within the Spring Framework. There are a number of challenges while creating a web-based application - like state management, workflow and validation, which are addressed by Spring's web framework. This framework can be used to automatically populate your model objects from incoming request parameters while providing validation and error handling as well. The entire framework is modular, with each set of components having specific roles and completely decoupled from the rest of the framework. This allows you to develop the front end of your web application in a very pluggable manner.

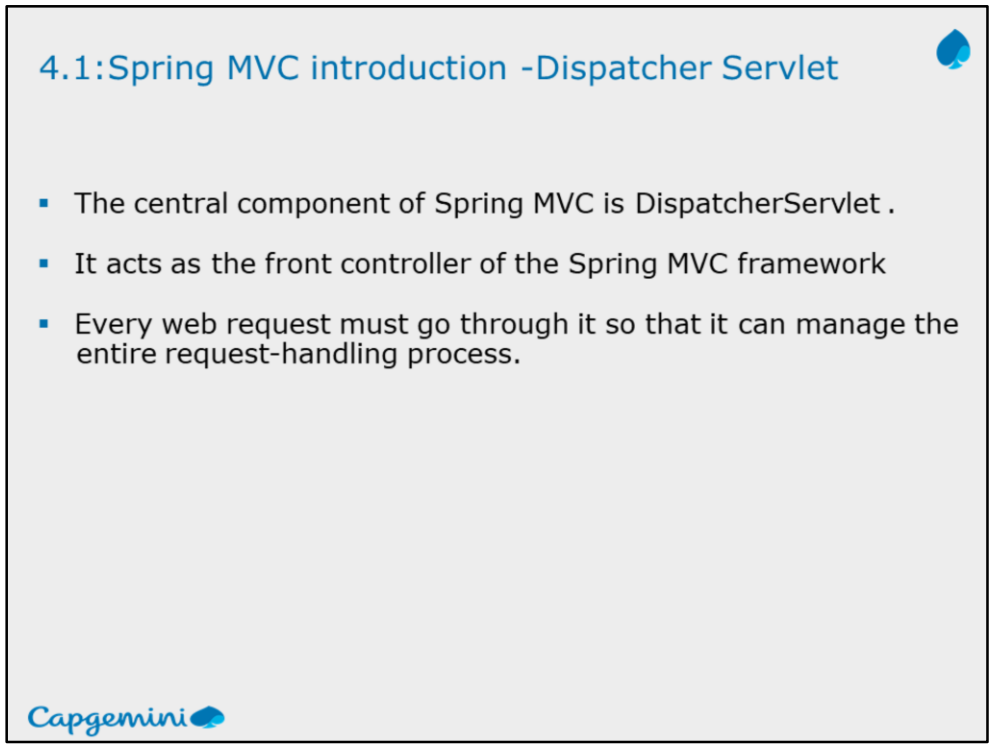
MVC provides out-of-the-box implementations of workflow typical to web applications. Its highly flexible, allowing you to use a variety of different view technologies. It also enables you to fully integrate with your Spring based, middle-tier logic through the use of dependency injection.
You can use Spring web MVC to make services that are created with other parts of Spring available to your users, by implementing web interfaces.

**Instructor Notes:**

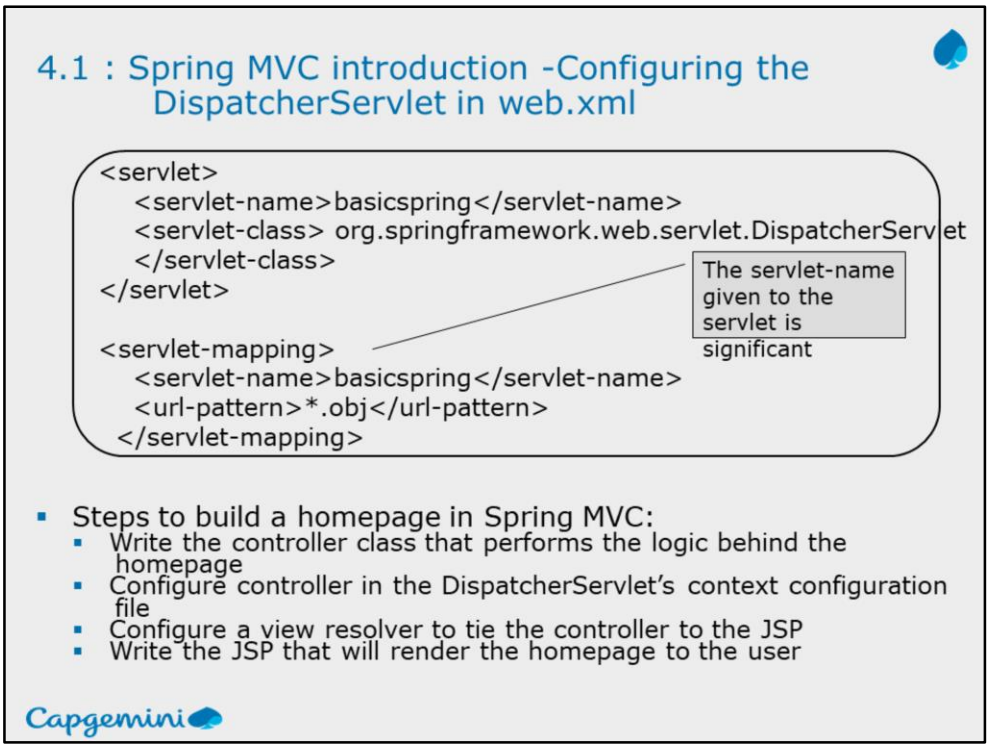



**Life cycle of a request in Spring MVC:** From the time that a request is received by Spring until the time that a response is returned to the client, many pieces of Spring MVC framework are involved.
The process starts when a client (typically a web browser) sends a request. It is first received by a DispatcherServlet. Like most Java-based MVC frameworks, Spring MVC uses a front-controller servlet (here DispatcherServlet) to intercept requests. This in turn delegates responsibility for a request to other components of an application for actual processing.
The Spring MVC uses a Controller component for handling the request. But a typical application may have several controllers. To determine which controller should handle the request, DispatcherServlet starts by querying one or more HandlerMappings. A HandlerMapping typically maps URL patterns to Controller objects.
Once the DispatcherServlet has a Controller object, it dispatches the request to the Controller which performs the business logic (a well-designed Controller object delegates responsibility of business logic to one or more service objects). Upon completion of business logic, the Controller returns a ModelAndView object to the DispatcherServlet. The ModelAndView object contains both the model data and logical name of view. The DispatcherServlet queries a ViewResolver with this logical name to help find the actual JSP.
Finally the DispatcherServlet dispatches the request to the View object, which is responsible for rendering a response back to the client.

**Instructor Notes:**



**Instructor Notes:**




**Configuring the DispatcherServlet**

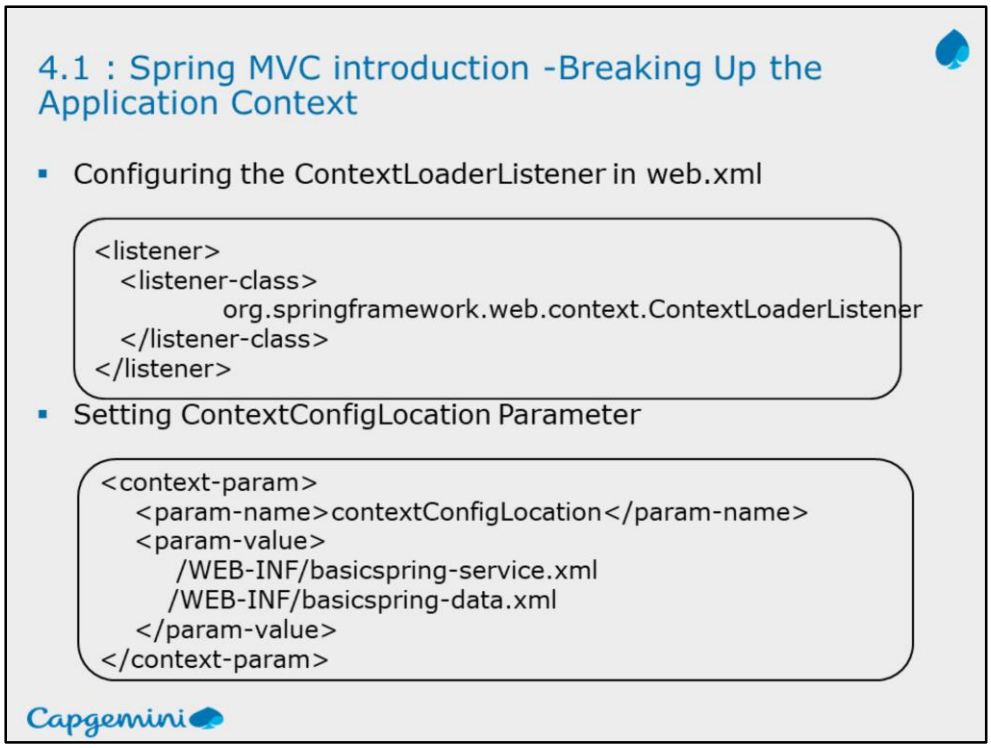
The dispatcher servlet is at the heart of the Spring MVC and functions as Spring MVC's front controller. Like any other servlet, it must be configured in web.xml file. Place the <servlet> declaration (in the first listing above) in the web.xml file. The servlet-name given to the servlet is significant. By default, when DispatcherServlet is loaded, it will load the Spring application context from an xml file whose name is based on the name of the servlet. In the above example, the servlet-name is basicspring and so the DispatcherServlet will try to load the application context from a file called basicspring-servlet.xml.

Next, you must indicate which URL's will be handled by DispatcherServlet. Add the following <servlet-mapping> tag (the second xml listing) to web.xml to let DispatcherServlet handle all url's that end in .obj. The URL pattern is arbitrary and could be any extension.

DispatcherServlet is now configured and ready to dispatch requests to the web layer of your application.

Pls. see the above steps that are followed while building a simpleMVC application. These steps are explained in detail in the coming slides.

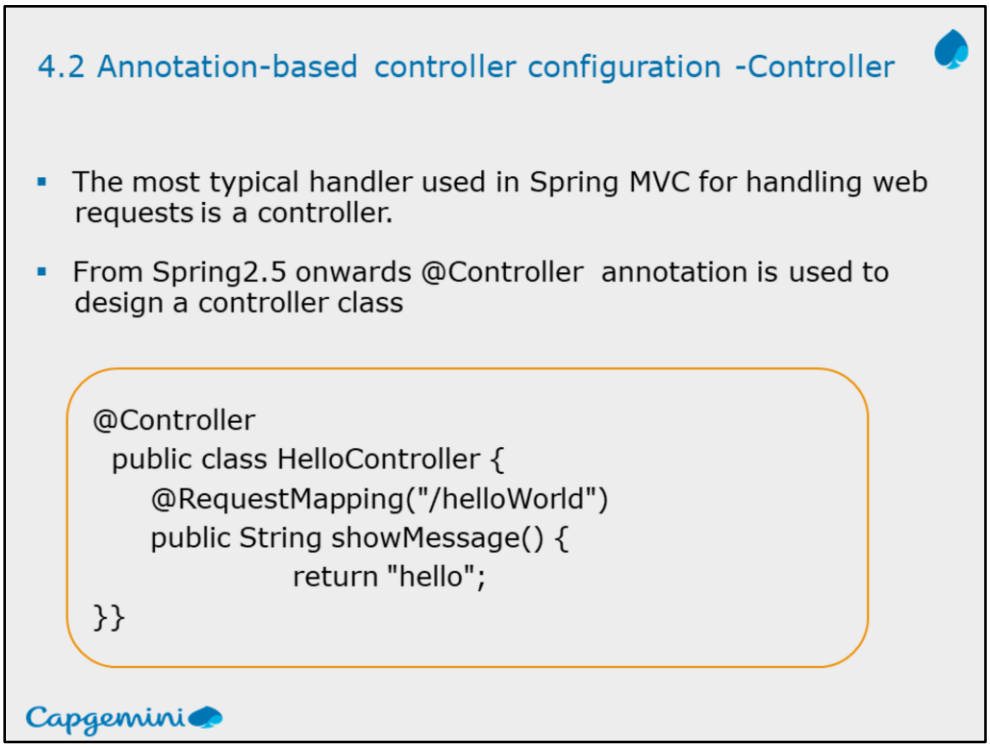
**Instructor Notes:**



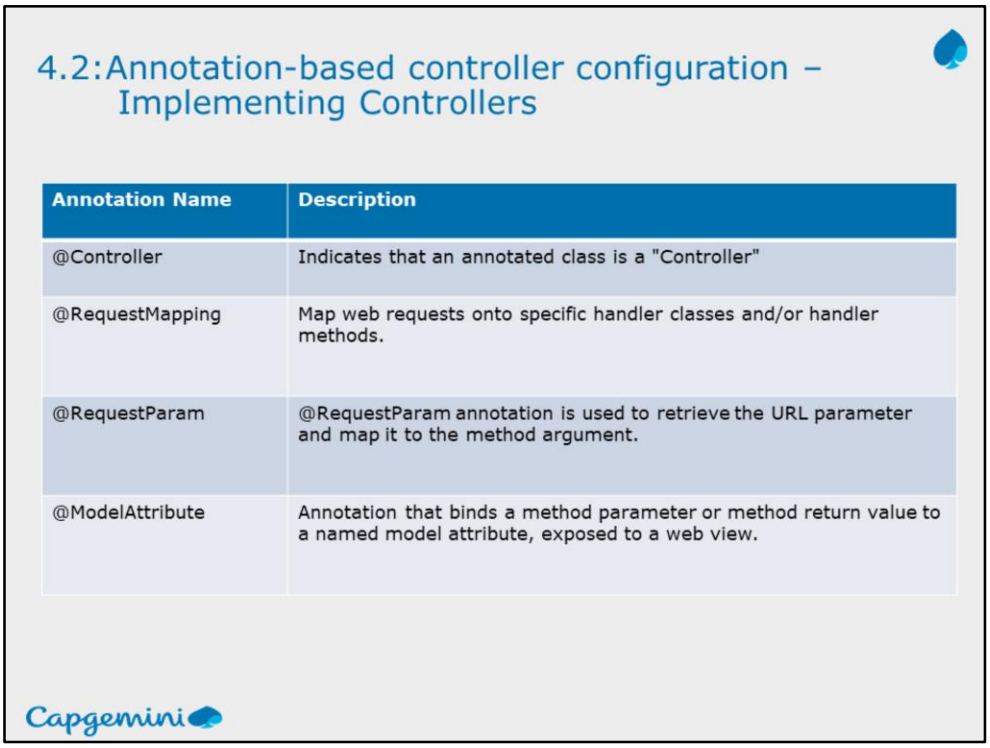

We have seen that DispatcherServlet will load the Spring application context from a single XML file whose name is <servlet-name>-servlet.xml. But you can also split the application context across multiple XML files. Ideally splitting it into logical pieces across application layers can make maintenance easier by keeping each of the Spring configuration files focused on a single layer of the application. To ensure that all the configuration files are loaded, you will need to configure a context loader in your web.xml file. A context loader loads context configuration files in addition to the one that DispatcherServlet loads. The most commonly used context loader is a servlet listener called ContextLoaderListener that is configured in web.xml as shown in the first listing above.

With ContextLoaderListener configured, we need to tell it the location of the Spring configuration file(s) to load. If not specified, context loader will look for a Spring configuration file at /WEB-INF/applicationContext.xml. We will specify one or more Spring configuration file(s) by setting the contextConfigLocation parameter in the servlet context as seen in the second listing above. The context loader will use contextConfigLocation to load the two context configuration files – one each for the service and data layer.

**Instructor Notes:**



**Instructor Notes:**




**@Controller:** Indicates that an annotated class is a "Controller"

**Example:**
@Controller
Public class LoginController{}

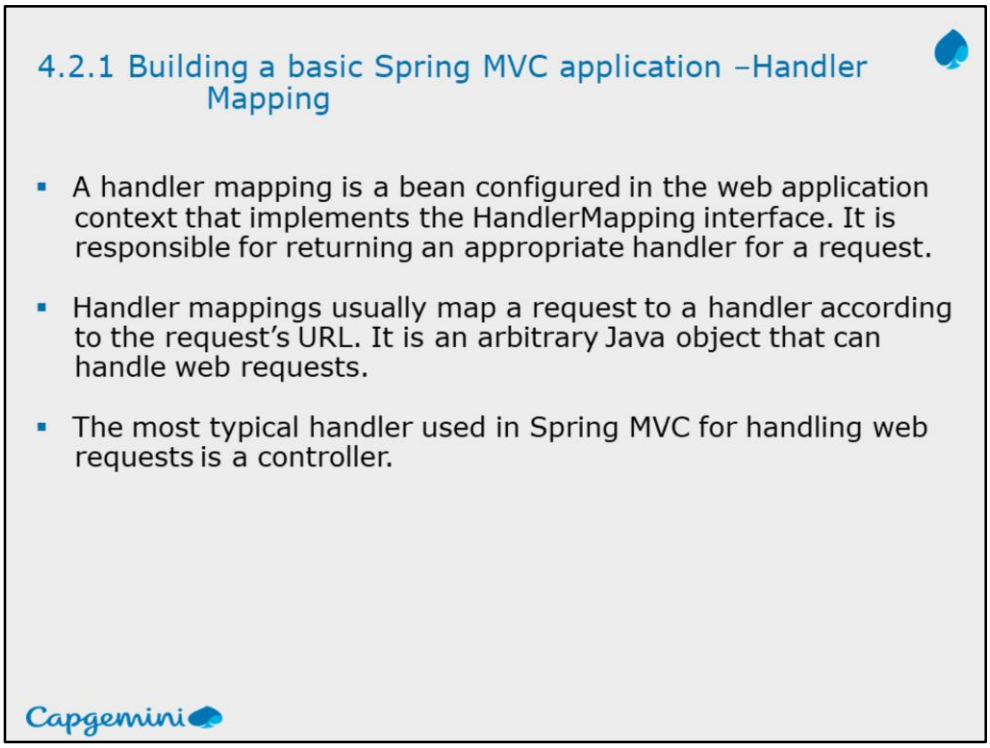
**@RequestMapping:** @RequestMapping can be applied to the controller class as well as methods. It maps web requests onto specific handler classes and/or handler methods.

**Example:**
@Controller
@RequestMapping(“loginController”)
Public class LoginController{
@RequestMapping(“loadForm”)
public String loadData(){}
}

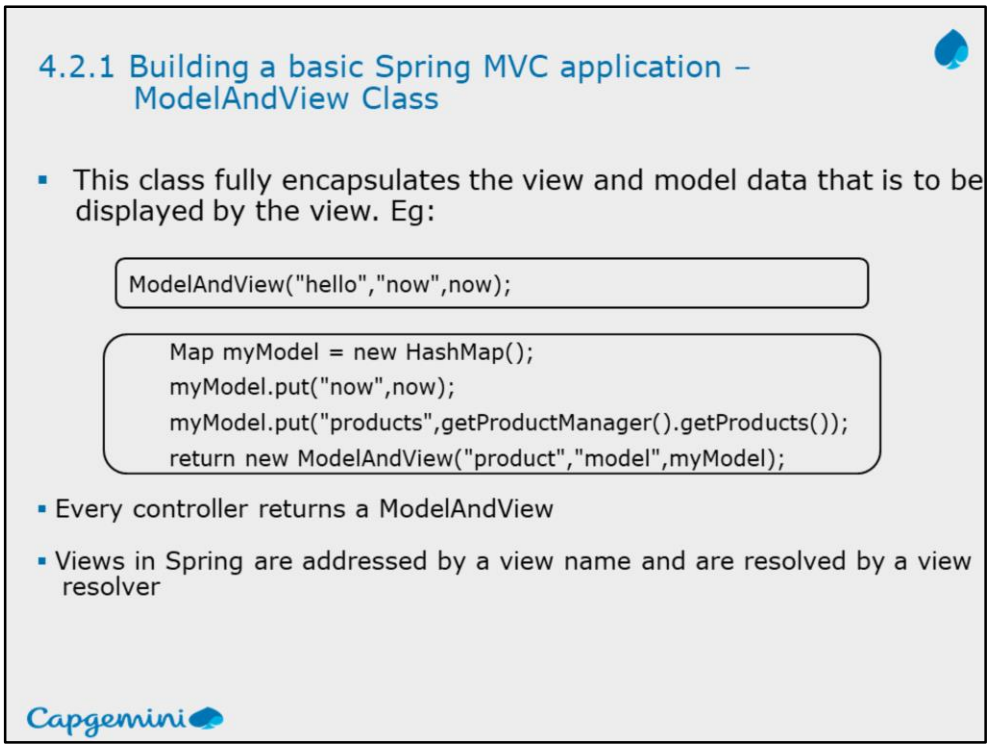
URL need to be used for making the request for the above mentioned controller is

http://localhost:8080/projectname/loginController/loadForm.obj

**Instructor Notes:**



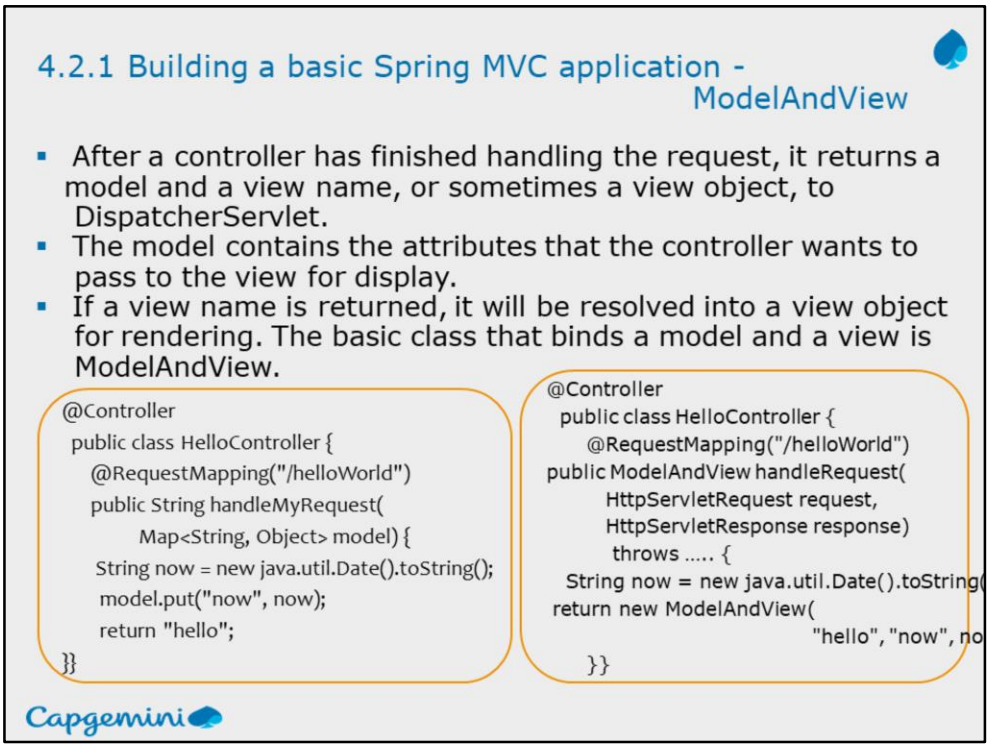
**Instructor Notes:**




The ModelAndView class allows you to specify a response to the client, resulting from actions performed in controller. This object holds both – the view the client will be presented with and the model used to render the view. Every controller must return a ModelAndView. See the two examples above.
The first parameter is the logical name of a view component that will be used to display the output from this controller. The next two parameters represent the model object that will be passed to the view and its value.
In the second example, the view component is product and the model is an object that is to be returned and its value is MyModel which is a HashMap object containing multiple values.
In the end, the dispatcher servlet needs a concrete view instance to render the view.

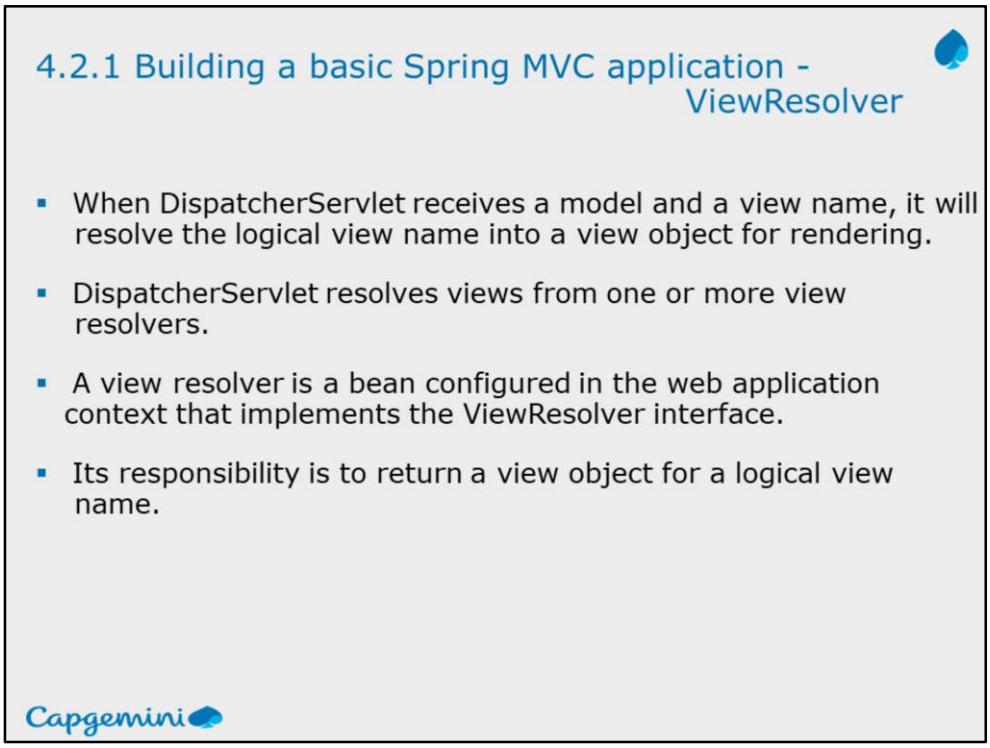
## Instructor Notes:

In previous versions of Spring, users were required to define HandlerMappings in the web application context to map incoming web requests to appropriate handlers. With the introduction of Spring 2.5, the DispatcherServlet enables the DefaultAnnotationHandlerMapping, which looks for @RequestMapping annotations on @Controllers.




Spring 2.5 introduced an annotation-based programming model for MVC controllers that uses annotations such as @Controller, @RequestMapping, @RequestParam, @ModelAttribute etc. Controllers implemented in this style do not have to extend specific base classes or implement specific interfaces.
See the listing above for a simple example.
**@Controller** annotation indicates that this class is a controller class. This is a specialization of the @Component annotation. Thus <context: component-scan> will pick up and register @Controller-annotated classes as beans, just as if they were annotated with @Component.
**@RequestMapping** annotation serves two purposes. First, it identifies handleRequest() as a request-handling method. Second, it specifies that this method should handle requests whose path is /helloWorld. See the second listing above. Notice that handleMyRequest(), a user-defined method, replaces the handleRequest() method. @RequestMapping annotation identifies it as a request-handling method.
The handleMyRequest() takes a Map as a parameter, which represents the model—the data that's passed between the controller and a view. But the signature of a request-handling method can have anything as an argument. Notice that handleMyRequest() returns a String value which is the logical name of the view that should render the results. ViewResolver uses this to resolve actual view.
Note : @RequestMapping annotation can be used at class level too to map URLs onto an entire class. The @RequestMapping at class level defines the root URL path that the controller will handle. Method-level @RequestMappings narrow the scope of class-level @RequestMapping.

**Instructor Notes:**



**Instructor Notes:**

## 4.2.1 Building a basic Spring MVC application - Resolving Views: The ViewResolver

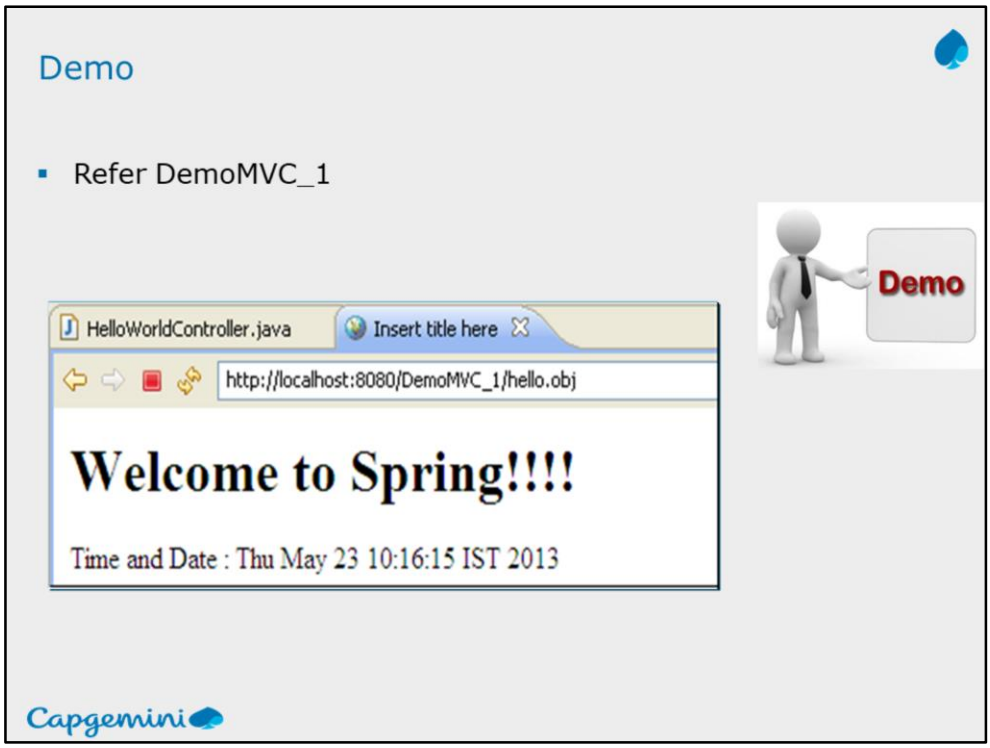
| View resolver | How it works |
|---|---|
| InternalResourceViewResolver | Resolves logical view names into View objects that are rendered using template file resources |
| BeanNameViewResolver | Looks up implementations of the View interface as beans in the Spring context, assuming that the bean name is the logical view name |
| ResourceBundleViewResolver | Uses a resource bundle that maps logical view names to implementations of the View interface |
| XmlViewResolver | Resolves View beans from an XML file that is defined separately from the application context definition files |

Capgemini

So far, we have seen how model objects are passed to the view through the ModelAndView object. In Spring MVC, a view is a bean that renders results to the user. The view most likely is a JSP. But you could also use other view technologies like Velocity and FreeMarker templates or even views that produce PDF and MS-Excel documents.
View resolvers resolve the view name given by the ModelAndView object to a View bean. Spring provides a number of useful view resolvers, some of which are shown in the table above. See Spring docs for more.
InternalResourceViewResolver resolves a logical view name by affixing a prefix and a suffix to the view name returned by the ModelAndView object. It then loads a View object with the path of the resultant JSP. By default, the view object is an InternalResourceView, which simply dispatches the request to the JSP to perform the actual rendering. But, if the JSP uses JSTL tags, then you may replace InternalResourceView with JstlView as seen in the code demos earlier.

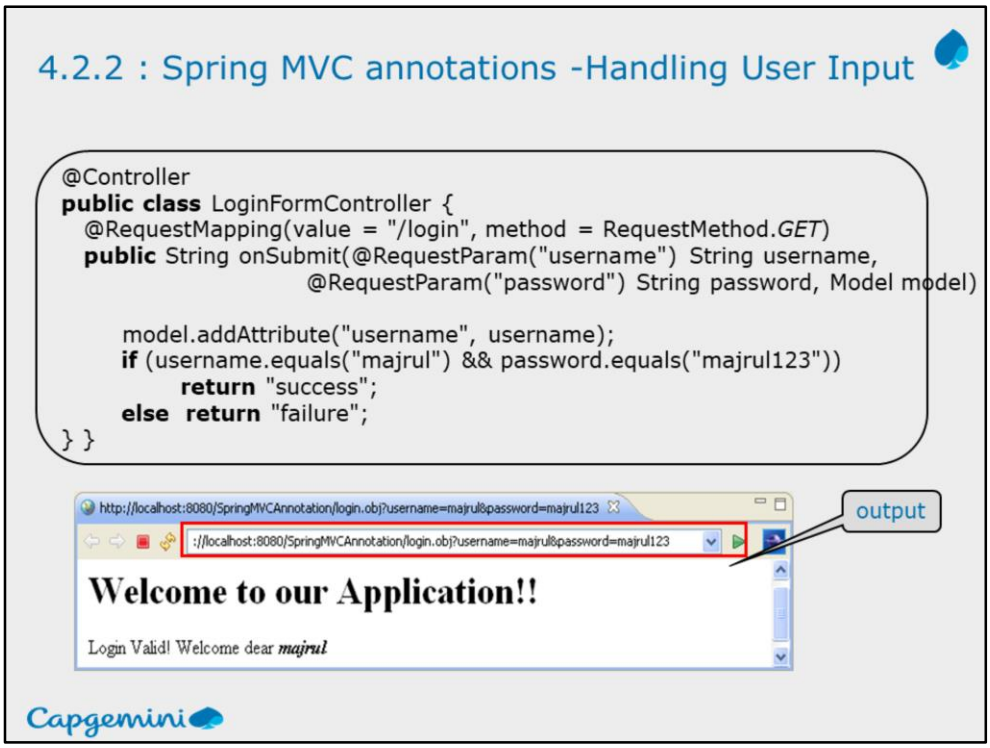
```
<bean id="viewResolver" class=
"org.springframework.web.servlet.view.InternalResourceViewResolver ">
  <property name="viewClass">
     <value>org.springframework.web.servlet.view.JstlView </value>
   </property>
  <property name="prefix"><value>/</value></property>
  <property name="suffix"><value>.jsp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

**Instructor Notes:**




Please refer to the DemoMVC_1 web project.

**Instructor Notes:**




We have seen simple Controllers so far. But Controllers may need to perform business logic using information passed in as URL parameters or as submitted form data.
See the code listing above for an example. Here, LoginFormController is mapped to /login at the method level. This means that onSubmit() handles requests for /login. “method” attribute is set to GET indicating that this method will only handle HTTP GET requests for /login. The onSubmit() method takes two String parameters (username and password) and a Model object as parameters. The username parameter is annotated with @RequestParam(“username") to indicate that it should be given the value of the username query parameter in the request. Same goes for password too. onSubmit() will use these parameters to authenticate user.
Now, see the model parameter. In earlier examples we passed in a Map<String, Object> to represent the model. Here we’re using a new Model parameter. Model provides a few convenient methods for populating the model, such as addAttribute(). The addAttribute() method is similar to Map’s put() method, except that it finds out key part of the map by itself.
Notice how the values set in addAttribute() can be accessed in the jsp page:

```
<!-- success.jsp →
<body>
<h1>Welcome to our Application!!</h1>
  Login Valid! Welcome dear <i><b>${username} </b></i>
</body>
```

**Instructor Notes:**

Demo 7->2

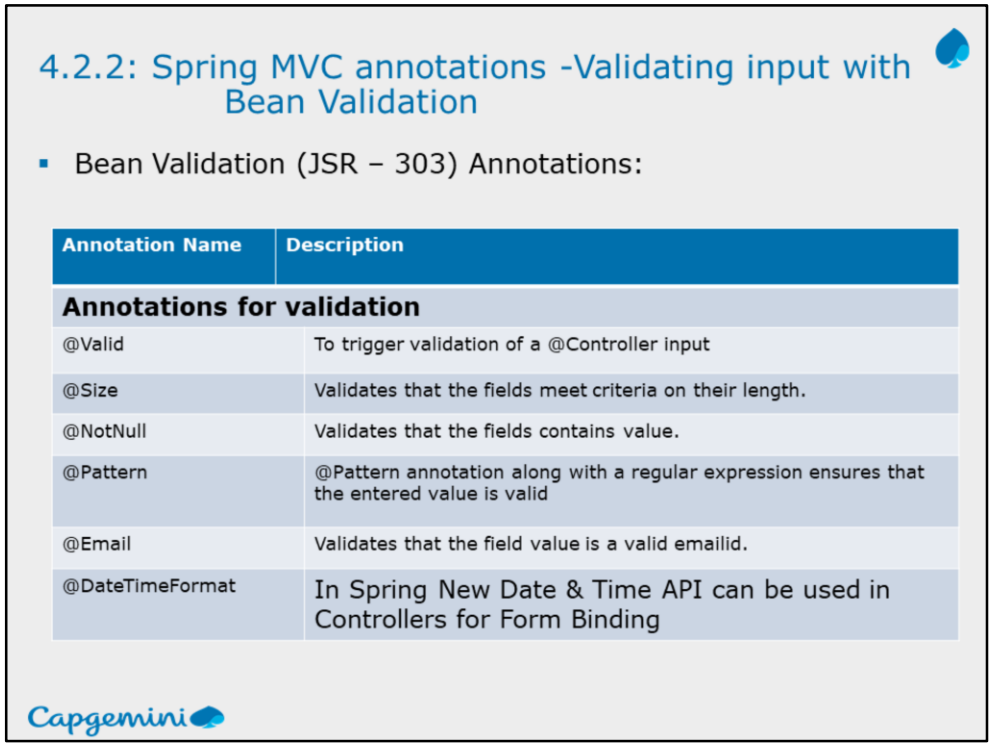
## Demo

- Refer DemoMVC_2 application


Capgemini

Please refer to the DemoMVC_2web project.
Refer to HelloWorldController.java.
Invoke the application as :
http://localhost:8080/DemoMVC_2/hi/hello.obj?name=CapGemini

**Instructor Notes:**




Before an object can be processed further, it is essential to ensure that all the data in the object is valid and complete. Faulty form input must be rejected. For example, username must not contain spaces, password must be minimum 6 characters long, email must be correct etc.

The @Valid annotation (part of the JavaBean validation specification) tells Spring that the User object should be validated as it's bound to the form input. If anything goes wrong while validating the User object, the validation error will be carried to the processForm() method via the BindingResult that's passed in on the second parameter. If the BindingResult's hasErrors() method returns true, then that means that validation failed.

How do we declare validation rules?

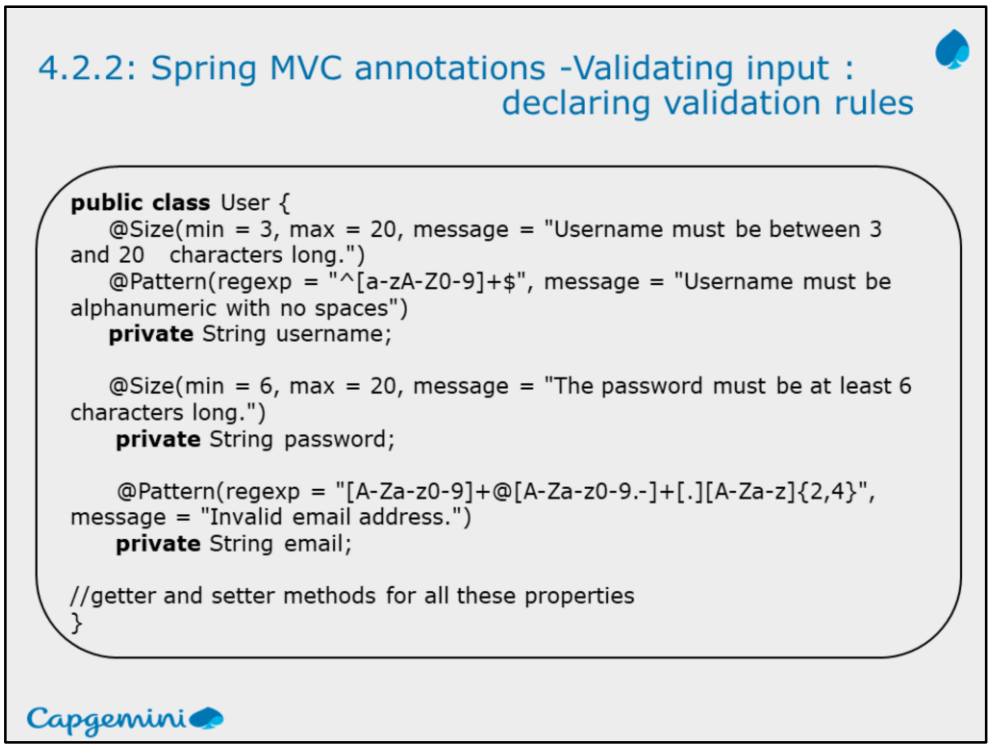
JSR-303 defines some annotations that can be placed on properties to specify validation rules. The code above shows the properties of the User class that are annotated with validation annotations.

@Size annotation validates that the fields meet criteria on their length.

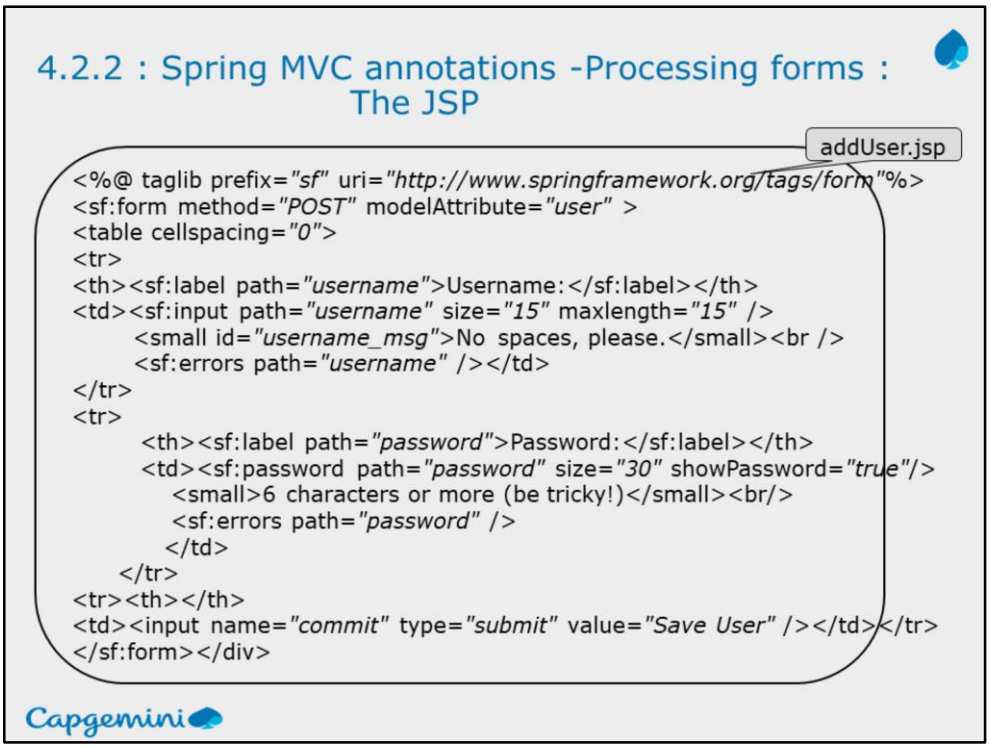
@Pattern annotation along with a regular expression ensures that the value given to the email property fits the format of an email address and that the username is only made up of alphanumeric characters with no spaces.

Notice how we've set the message attribute with the message to be displayed in the form when validation fails. With these annotations, when a user submits a registration form to AddUserFormController's processForm() method, the values in the User object's fields will be validated. If any of those rules are violated, then the handler method will send the user back to the form.

**Instructor Notes:**



**Instructor Notes:**



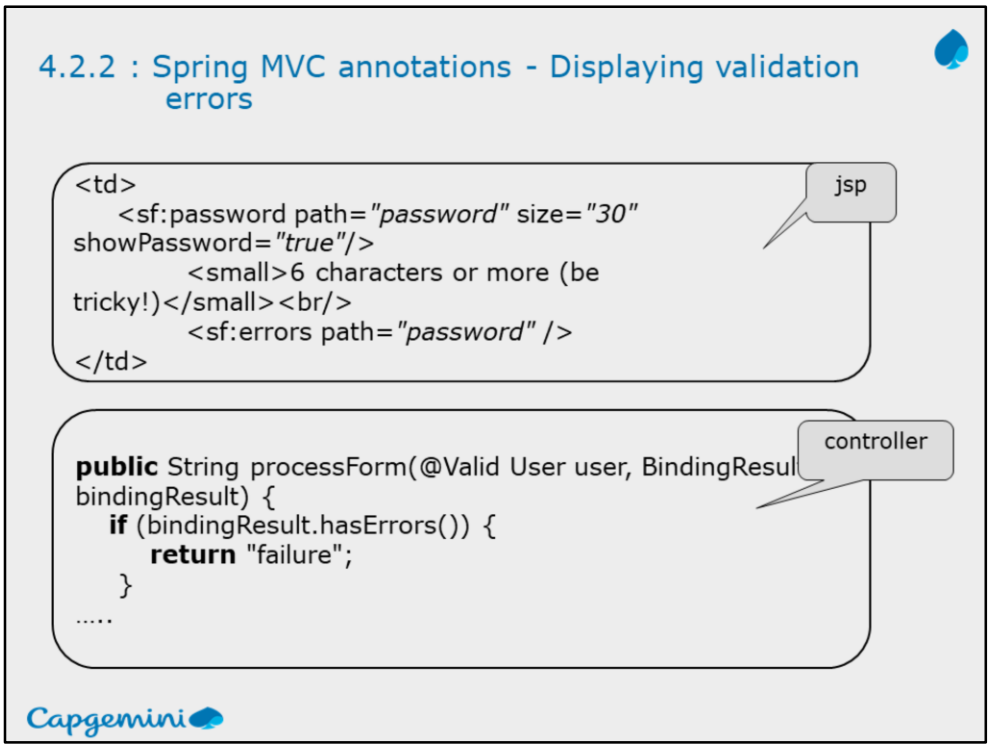

The addUser.jsp page uses Spring's form binding library. The <sf:form> tag binds the User object (identified by the modelAttribute attribute - see above) that showForm() placed into the model to the various fields in the form. The <sf:input> and <sf:password> tags have a path attribute that references the property of the User object that the form is bound to. When the form is submitted, whatever values these fields contain will be placed into a User object and submitted to the server for processing.

Note that the <sf:form> specifies that it'll be submitted as an HTTP POST request. We thus now need another handler method that accepts POST requests. See the processForm() method in the code listing in the previous page which will process form submissions.

When the addUser.jsp form is submitted, the fields in the request will be bound to the User object (passed as an argument to processForm(). From there, some logic can be employed to persist user object into database.

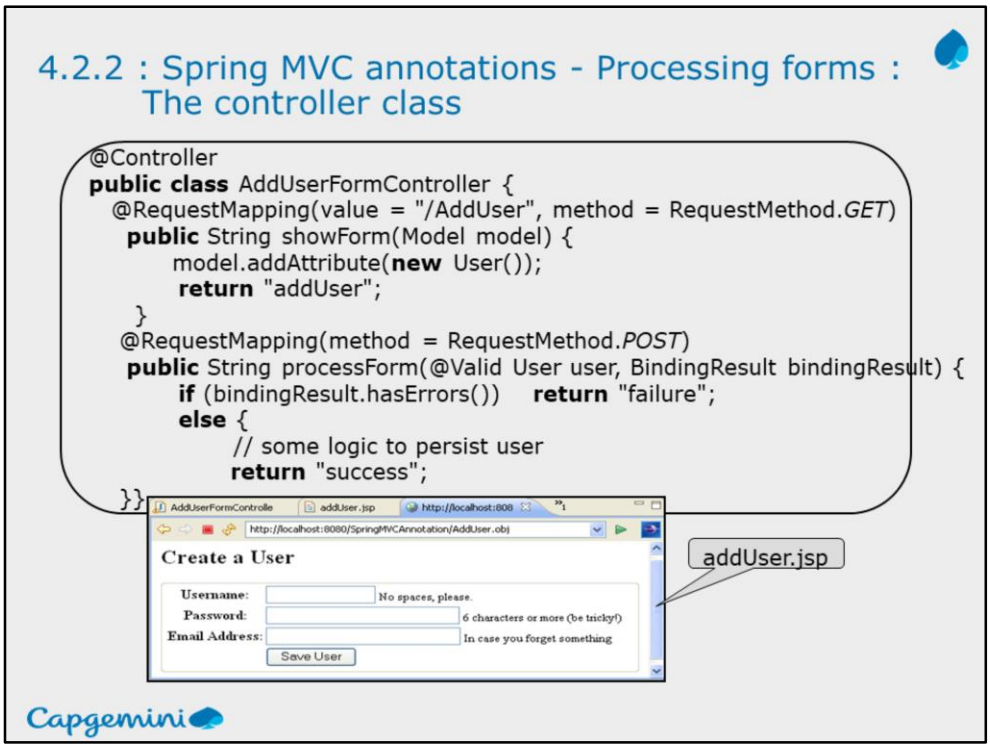
Notice that User parameter is annotated with @Valid. This indicates that the User should pass validation before being persisted. We shall cover validation next.

**Instructor Notes:**



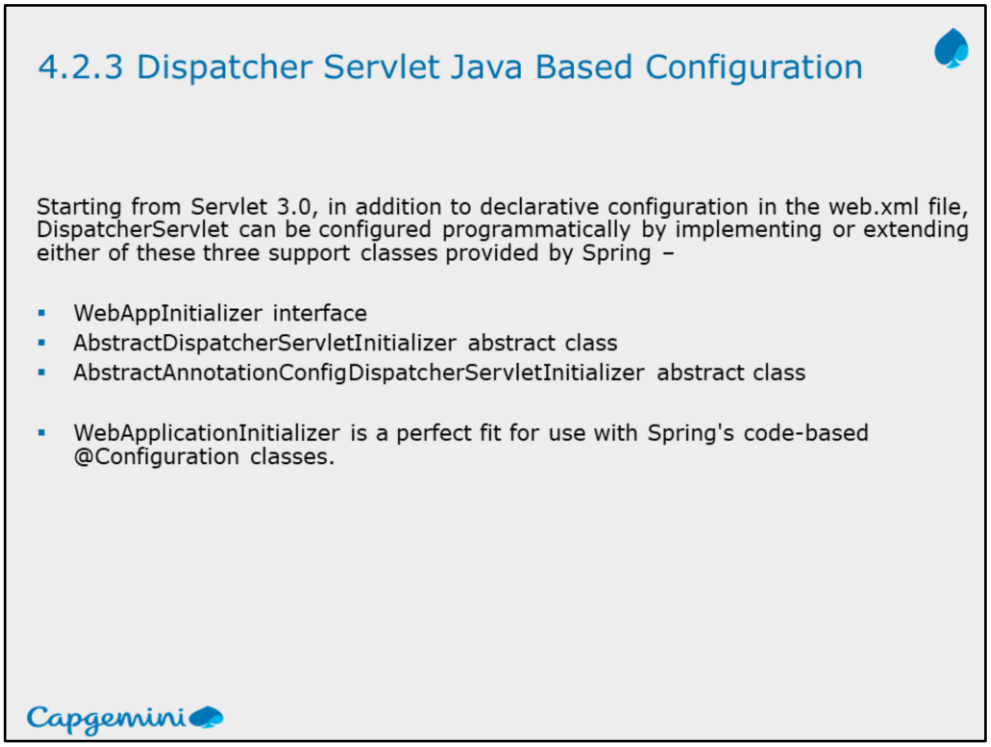

We now need to tell jsp to display the validation messages.
We have passed the BindingResult parameter to processForm(). This knows whether the form had any validation errors via its hasErrors() method. But the actual error messages are also in there, associated with the fields that failed validation. To display those errors to the users, use Spring's form binding JSP tag library to display the errors. ie, the <sf:errors> tag can render field validation errors. See code listing above to see how this is done.
The <sf:errors> tag's path attribute specifies the form field for which errors should be displayed. For example, the above listing displays errors (if any) for the field whose name is password. If there are multiple errors for a single field, they'll all be displayed, separated by an HTML <br/> tag.

**Instructor Notes:**

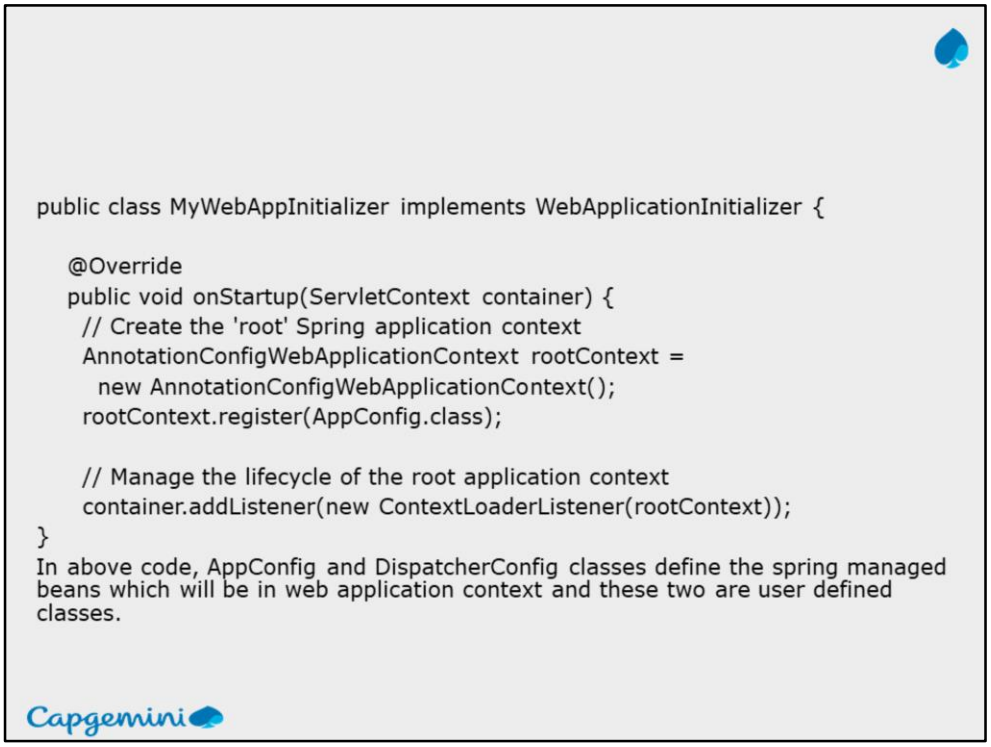



Working with forms in a web application involves two operations: displaying the form and processing the form submission.
Our example allows to register new User. For this we have defined two handler methods to AddUserFormController to handle each of the operations.
We first need to display the form in the browser before it can be submitted.
The first method (showForm()) displays the registration form. See the figure above.
Once the form is displayed, it'll need a User object to bind to the form fields.
The showForm() method creates a User object and places it in the model.

**Instructor Notes:**



**Instructor Notes:**

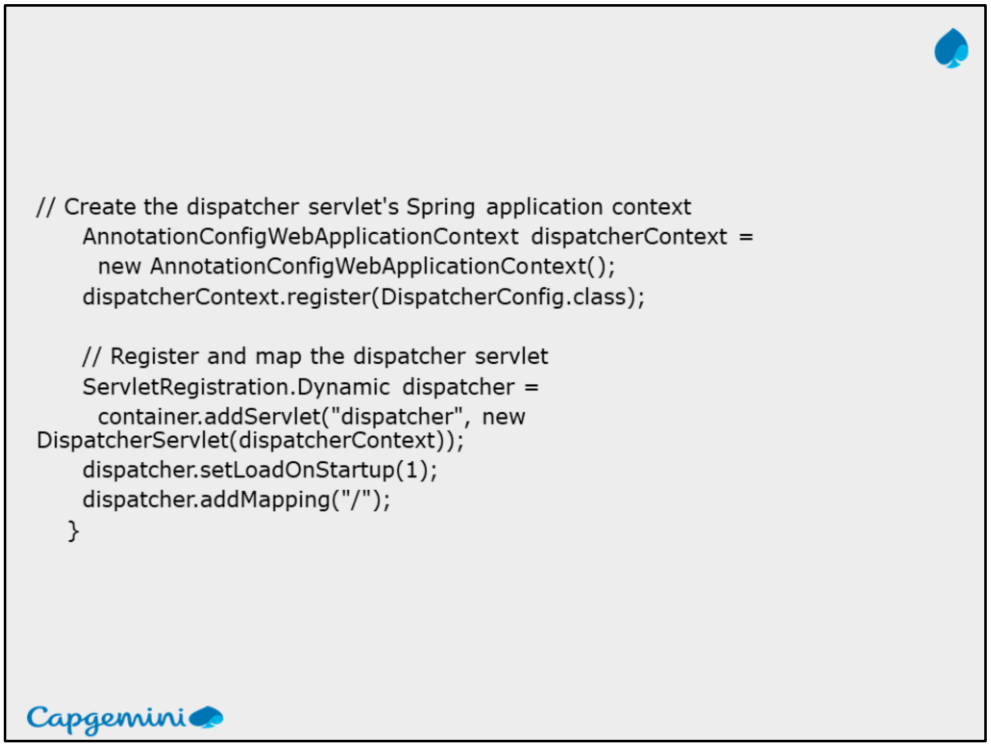


```
public class MyWebAppInitializer implements WebApplicationInitializer {

   @Override
   public void onStartup(ServletContext container) {
     // Create the 'root' Spring application context
     AnnotationConfigWebApplicationContext rootContext =
       new AnnotationConfigWebApplicationContext();
     rootContext.register(AppConfig.class);

     // Manage the lifecycle of the root application context
     container.addListener(new ContextLoaderListener(rootContext));
}
```

In above code, AppConfig and DispatcherConfig classes define the spring managed beans which will be in web application context and these two are user defined classes.

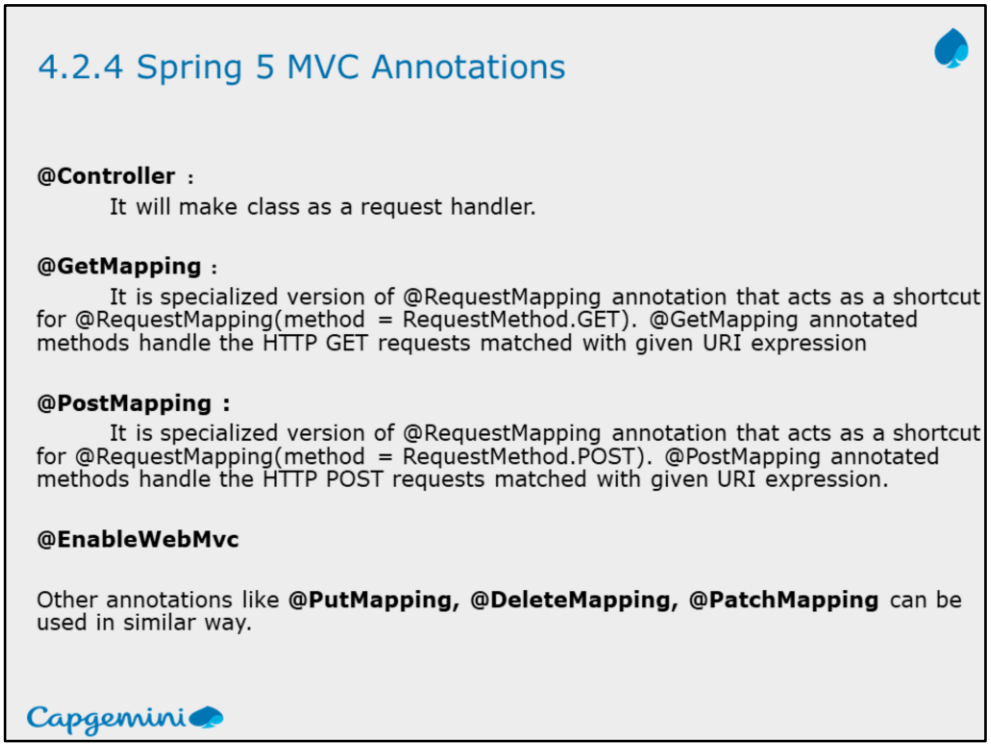
Capgemini

**Instructor Notes:**



```
// Create the dispatcher servlet's Spring application context
     AnnotationConfigWebApplicationContext dispatcherContext =
       new AnnotationConfigWebApplicationContext();
     dispatcherContext.register(DispatcherConfig.class);

     // Register and map the dispatcher servlet
     ServletRegistration.Dynamic dispatcher =
       container.addServlet("dispatcher", new
DispatcherServlet(dispatcherContext));
     dispatcher.setLoadOnStartup(1);
     dispatcher.addMapping("/");
   }
```

**Instructor Notes:**




```
package com.cg.controller;

@Controller
public class HomeController
{
   @GetMapping("/")
   public String homeInit(Model model) {
      return "home";
   }
   @GetMapping("/home")
public String homeInit(Model model) {
   return "home";
}

@GetMapping("/members/{id}")
public String getMembers(Model model) {
   return "member";
}
@PostMapping(path = "/members", consumes = "application/json", produces
="application/json")
public void addMember(@RequestBody Member member) {
   //code
}

}
```

**Instructor Notes:**

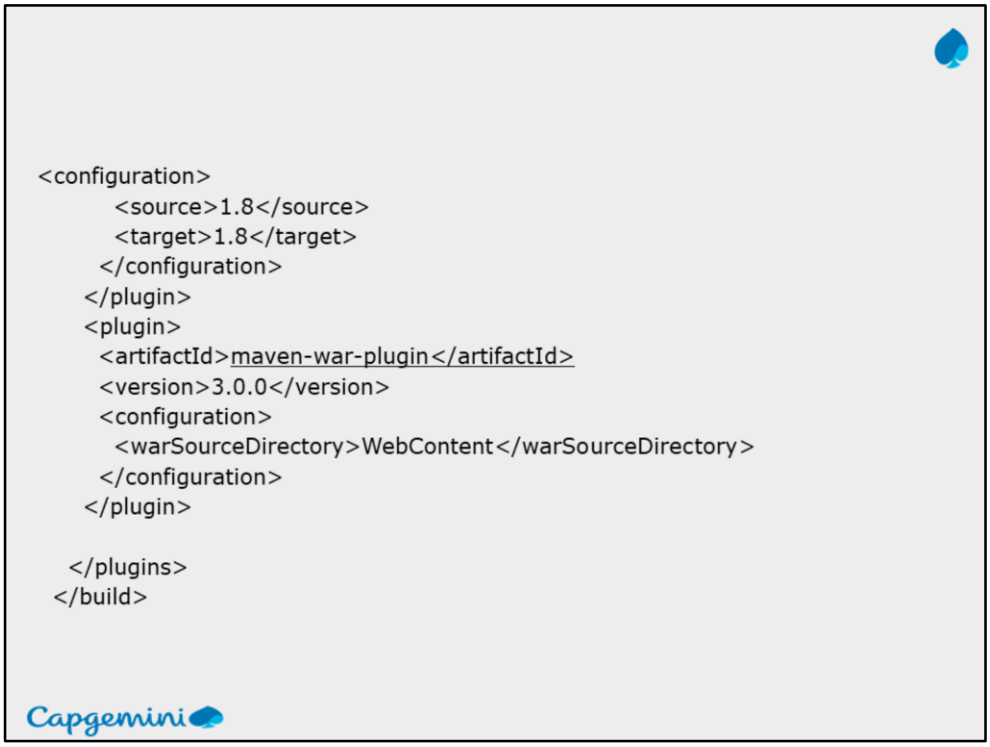
## Java Configuration

**STEP 1: Create POM.xml**

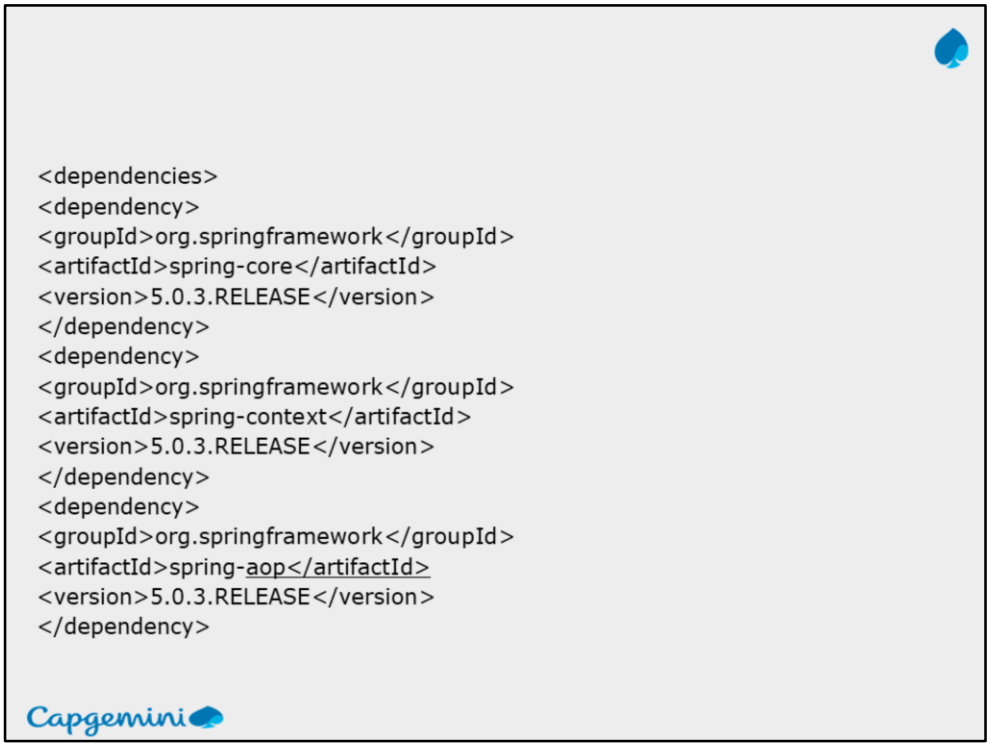
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>MVCJavaBasedExample</groupId>
  <artifactId>MVCJavaBasedExample</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>
  <build>
    <sourceDirectory>src</sourceDirectory>
    <plugins>
        <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.7.0</version>
```

Capgemini

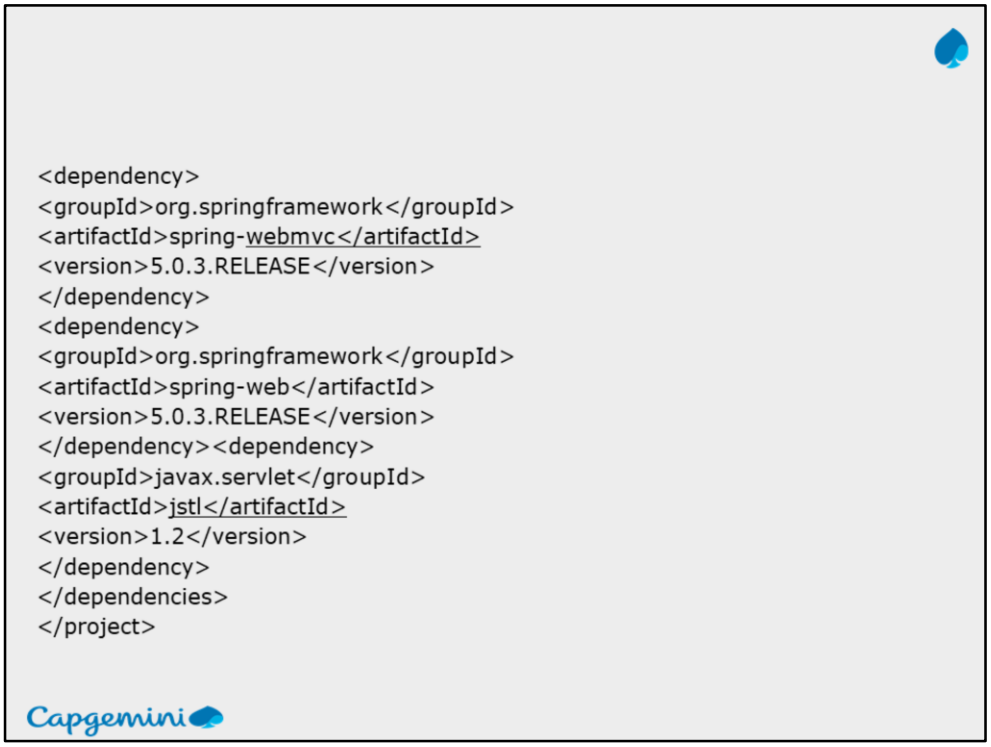
**Instructor Notes:**



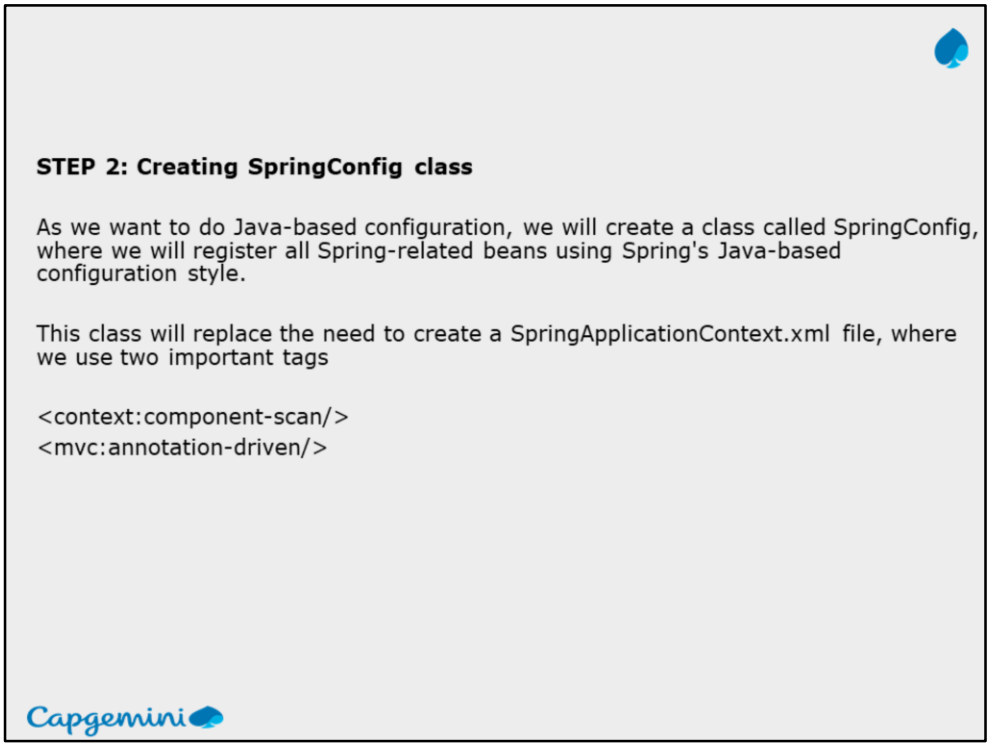
**Instructor Notes:**



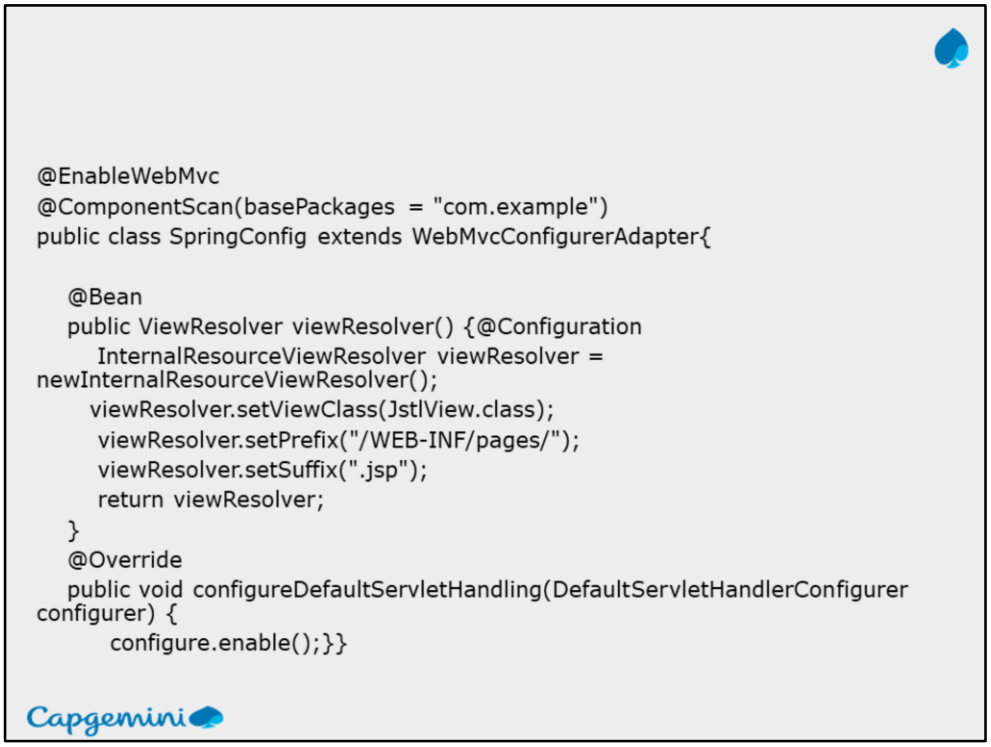
**Instructor Notes:**



**Instructor Notes:**

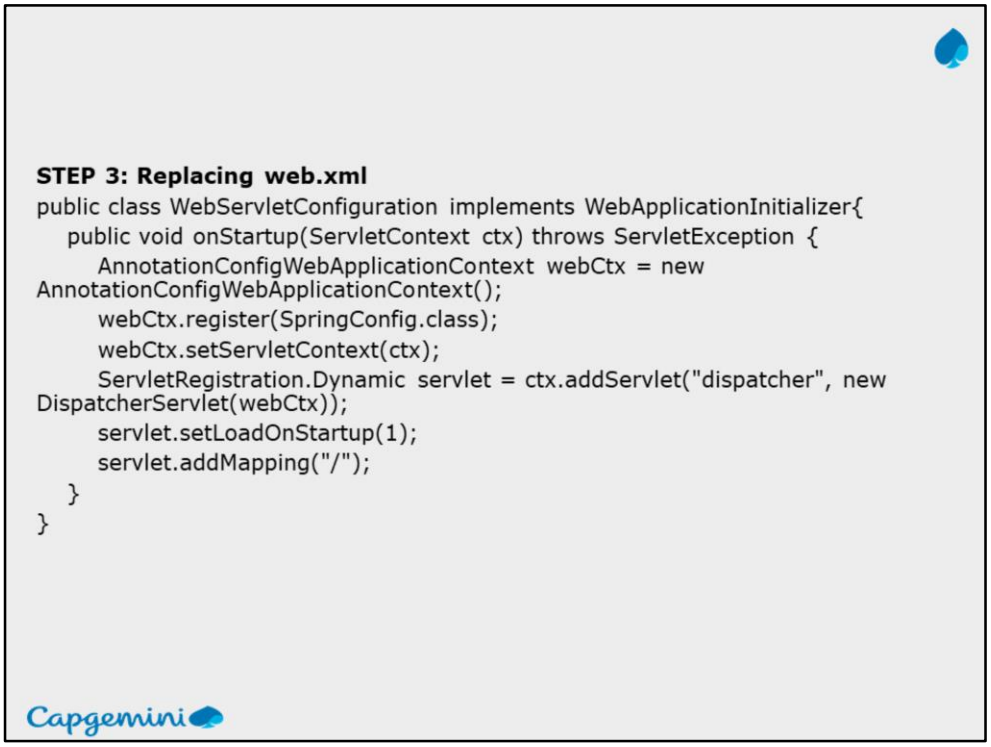


**Instructor Notes:**




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

**Instructor Notes:**

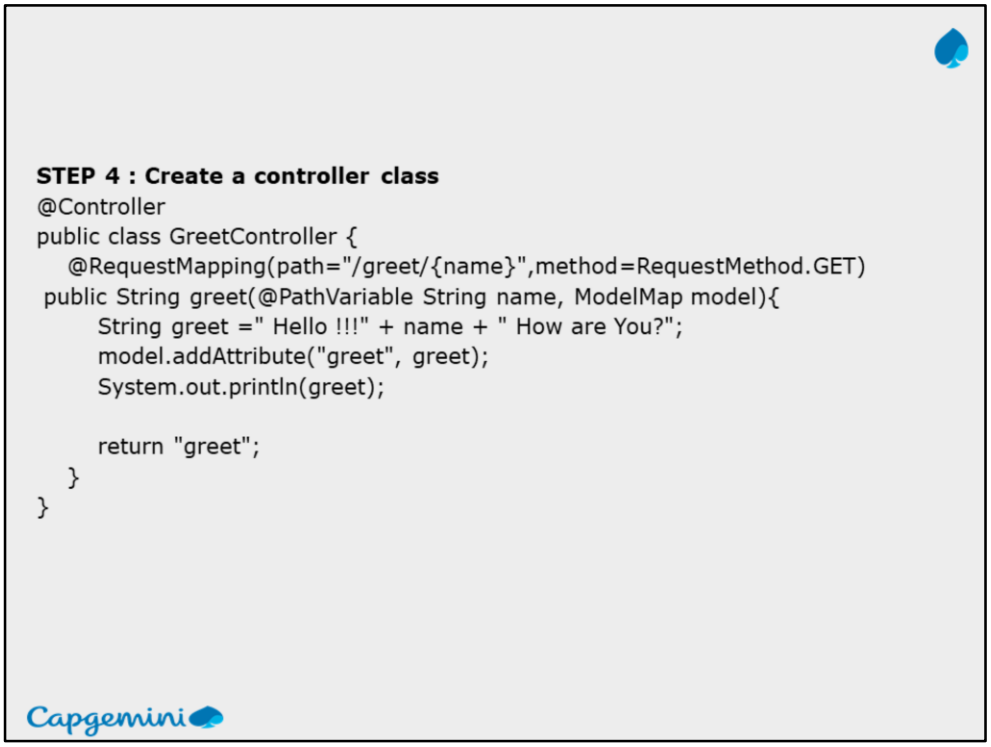


**STEP 3: Replacing web.xml**

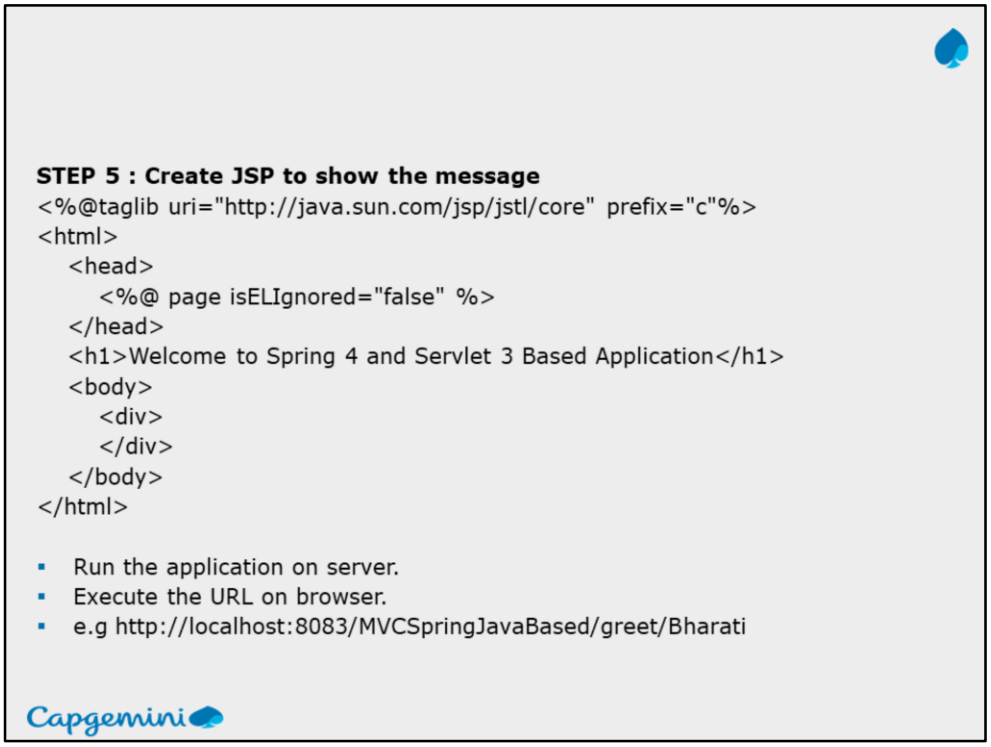
```
public class WebServletConfiguration implements WebApplicationInitializer{
   public void onStartup(ServletContext ctx) throws ServletException {
      AnnotationConfigWebApplicationContext webCtx = new
AnnotationConfigWebApplicationContext();
      webCtx.register(SpringConfig.class);
      webCtx.setServletContext(ctx);
      ServletRegistration.Dynamic servlet = ctx.addServlet("dispatcher", new
DispatcherServlet(webCtx));
      servlet.setLoadOnStartup(1);
      servlet.addMapping("/");
   }
}
```

Capgemini

**Instructor Notes:**



**Instructor Notes:**



**STEP 5 : Create JSP to show the message**

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<html>
   <head>
      <%@ page isELIgnored="false" %>
   </head>
   <h1>Welcome to Spring 4 and Servlet 3 Based Application</h1>
   <body>
      <div>
      </div>
   </body>
</html>
```

- Run the application on server.
- Execute the URL on browser.
- e.g http://localhost:8083/MVCSpringJavaBased/greet/Bharati

Capgemini

**Instructor Notes:**

Demo 7->2

## Demo

- Refer MVCJavaBasedExample application


Capgemini

**Instructor Notes:**

Additional notes for instructor

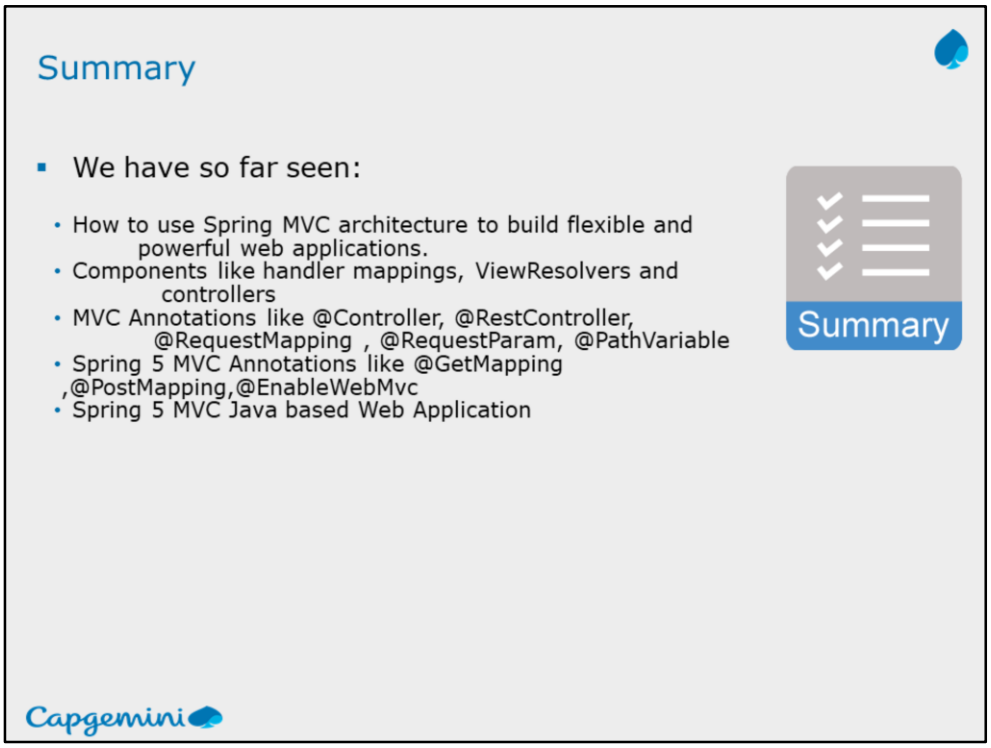
## Demo

- Refer the following Demos:
  - DemoMVC_3
  - DemoMVC_4
  - DemoMVC_5
  - DemoMVC_6
  - DemoMVC_Complete
  - MVCJavaBasedExample


Capgemini

**Instructor Notes:**

Additional notes for instructor



**Instructor Notes:**

Ans-1 : 1. Pls. explain, if necessary, to participants using the older technique's config file. Example is available therein.

Ans-2 : 1

## Review Questions

- Question 1: If multiple handler mappings have been declared in an application, select the property that indicates which handler mapping has precedence?
  - Option 1: Order
  - Option 2: Sequence
  - Option 3: Index
  - Option 4: An application cant have multiple handler mappings

- Question 2: To figure out which controller should handle the request, DispatcherServlet queries _____
  - Option 1: HandlerMappings
  - Option 2: ModelAndView
  - Option 3: ViewResolver
  - Option 4: HomeController

Capgemini