Introduction to MVC (Model-View-Controller)

The Model-View-Controller (MVC) design pattern is a foundational concept in web application development. It separates an application into three interconnected components:

1. **Model**: Represents the application's data and business logic.
2. **View**: Displays the data (from the model) to the user and sends user commands to the controller.
3. **Controller**: Handles user input, manipulates the model, and updates the view.

This separation of concerns makes applications easier to manage, test, and scale.

Understanding MVC1 and MVC2 Architectures

MVC1 Architecture:

- **Description**: In the early days of web development, MVC1 was a common pattern where the JSP (JavaServer Pages) handled both the presentation (View) and some of the business logic (Controller). The Model was directly embedded within JSP, leading to a mix of business logic and presentation code.
- **Characteristics**:
  - **Tight coupling**: Business logic and presentation are tightly coupled.
  - **Less maintainable**: As the application grows, maintaining the code becomes challenging due to the intertwined logic.
- **Flow**:
  1. User interacts with the JSP.
  2. JSP processes the request and interacts with the Model.
  3. The JSP returns the response to the user.

MVC2 Architecture:

- **Description**: MVC2 is a more refined version of the MVC pattern, which enforces a strict separation of concerns. The Controller, usually a servlet, handles all user requests, processes them using the Model, and then delegates the response to the appropriate View (JSP).
- **Characteristics**:
    - **Loose coupling**: Business logic, presentation, and control flow are separated.
    - **More maintainable**: Easier to manage and scale as the application logic is cleanly separated.
- **Flow**:
    1. User interacts with the Controller (Servlet).
    2. The Controller processes the request and updates the Model.
    3. The Controller forwards the response to a View (JSP).
    4. The View renders the data and sends it back to the user.

Front Controller Design Pattern

Description: The Front Controller Design Pattern is a single entry point for handling all requests. In the context of MVC2, this is usually implemented by a Servlet, which acts as the Controller.

Advantages:

- **Centralized request handling**: All requests are handled in a unified way.
- **Improved security**: Central control of request processing allows for better security mechanisms.
- **Reusability**: Common processing logic can be reused for multiple requests.

Flow:

1. All incoming requests are intercepted by the Front Controller (e.g., DispatcherServlet in Spring MVC).

2. The Front Controller determines the appropriate handler (Controller) for the request.
3. The handler processes the request and returns the Model and View.
4. The Front Controller renders the View with the provided data and sends the response back to the user.

Spring MVC Basics

Spring MVC is a web framework within the Spring Framework that provides a Model-View-Controller architecture for developing web applications.

Core Concepts:

- **DispatcherServlet**: Acts as the Front Controller in Spring MVC. It intercepts requests, dispatches them to appropriate handlers, and manages the flow of the application.
- **Controller**: Handles user requests and returns a Model and View.
- **Model**: Encapsulates the data the application works with.
- **View**: Responsible for rendering the UI based on the Model data.

Flow:

1. A user sends a request to the application.
2. The request is intercepted by the DispatcherServlet.
3. The DispatcherServlet consults the handler mappings to find the appropriate Controller.
4. The Controller processes the request, interacts with the Model, and returns a View name.
5. The DispatcherServlet forwards the response to the appropriate View resolver.
6. The View renders the data and sends it back to the user.

Configuration and the DispatcherServlet

DispatcherServlet Configuration:

- **web.xml**:

```
<web-app id = "WebApp_ID" version = "2.4"
   xmlns = "http://java.sun.com/xml/ns/javaee"
   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>dispatcher</servlet-name>
      <servlet-
class>org.springframework.web.servlet.DispatcherSer
vlet</servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
      <servlet-name>dispatcher</servlet-name>
      <url-pattern>/</url-pattern>
   </servlet-mapping>

</web-app>
```

- **Spring Configuration (dispatcher-servlet.xml)**:

```
<?xml version="1.0" encoding="UTF-8"?>

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

```
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xsi:schemaLocation="http://www.springframework.or
g/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/sprin
g-context-3.0.xsd">

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


  <bean

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



  </bean>

</beans>
```

Key Points:

- **DispatcherServlet** is the core component in Spring MVC that controls the request flow.
- Configuration can be done either through XML or Java annotations (e.g., @Configuration, **@EnableWebMvc**).
- Spring Boot simplifies this process with auto-configuration.

@Controller and @RequestMapping (Handlers)

@Controller: Indicates that a particular class serves the role of a Controller in Spring MVC.

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {
    @RequestMapping("/hello")
    public String sayHello() {
        return "hello"; // returns the view name "hello.jsp"
    }
}
```

@RequestMapping: Maps web requests to specific handler methods.

- **Class Level**:

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping("/home")
public class HomeController {
    @RequestMapping("/welcome")
    public String welcome() {
        return "welcome";
    }
}
```

Here, the URL would be **/home/welcome**.

Method Level:

```
package com.mphasis.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/home")
public class HomeController {
   @RequestMapping(value = "/submit", method =RequestMethod.POST)
   public String handleSubmit() {
       return "result";
   }
}
```

Specifies the URL pattern and HTTP method (GET, POST, etc.).

Additional Attributes:

- params: Specifies request parameters that must be present.
- headers: Specifies required request headers.

@RequestParam and Parameter Binding

@RequestParam: Binds a web request parameter to a method argument in the controller.

```
package com.mphasis.controller;

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

@Controller
@RequestMapping("/home")
public class HomeController {
     @RequestMapping("/greet")
     public String greetUser(@RequestParam("name") String
name, Model model) {
         model.addAttribute("message", "Hello " + name);
         return "greet";
     }
```

- }

Attributes:

  - value or name: Name of the request parameter.

  - required: Indicates if the parameter is mandatory (true by default).
  - defaultValue: Specifies a default value if the parameter is not present.

View Resolvers

Purpose: A View Resolver maps view names returned by controllers to actual view files.

Types:

- **InternalResourceViewResolver**: Maps a view name to a JSP file under a specific directory.

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

@Configuration
@EnableWebMvc
@ComponentScan(basePackages="com.mphasis")
public class MyWebMVCConfig {

   @Bean
   public ViewResolver getViewResolver() {

         InternalResourceViewResolver viewResolver = new
InternalResourceViewResolver();
         viewResolver.setPrefix("/WEB-INF/jsp/");
         viewResolver.setSuffix(".jsp");
         return viewResolver;

   }

}
```

Here, the view name hello would resolve to /WEB-INF/views/hello.jsp.

Controller Details - @RequestParam, @PathVariable

@RequestParam: Binds query parameters, form data, or parts of a multi-part request.

- **Example**:

```
package com.mphasis.controller;

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

@Controller
@RequestMapping("/home")
public class HomeController {
	@RequestMapping("/search")
	public String search(@RequestParam("query") String query, Model
	model) {
		model.addAttribute("result", searchService.search(query));
		return "searchResults";
	}

}
```

- **@PathVariable**: Binds a URI template variable to a method parameter.

- **Example**:

```
package com.mphasis.controller;

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

@Controller
@RequestMapping("/home")
public class HomeController {
	@RequestMapping("/user/{id}")
	public String getUser(@PathVariable("id") String userId, Model
	model) {
		model.addAttribute("user",
		userService.findUserById(userId));
		return "userProfile";
	}

}
```

- If the URL is /user/123, the method parameter userId would be 123.

Model Data and @ModelAttribute

Model: In Spring MVC, the Model interface is used to pass data from the controller to the view.

- **Adding Data**:

```
package com.mphasis.controller;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import
org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HomeController {
    @RequestMapping("/home")
    public String home(Model model) {
        model.addAttribute("message", "Welcome to Spring
MVC");
        return "home";
    }

}
```

- **@ModelAttribute**: Used to bind a method parameter or method return value to a named model attribute.

On Method:

```
package com.mphasis.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;

@Controller
public class HomeController {
     @ModelAttribute("user")
     public User populateUser() {
          return new User();
     }

}
```

- This method runs before every request mapping in the controller, adding a User object to the model.
- **On Method Parameter**:

```
@Controller
public class HomeController {
  @RequestMapping("/submitForm")
  public String submitForm(@ModelAttribute("user")
User user) {
      // user object is populated with form data
      return "result";
  }


}
```

Use Cases:

- Binding form data to a model.
- Pre-populating forms with default data.
- Ensuring certain data is always available in the model.