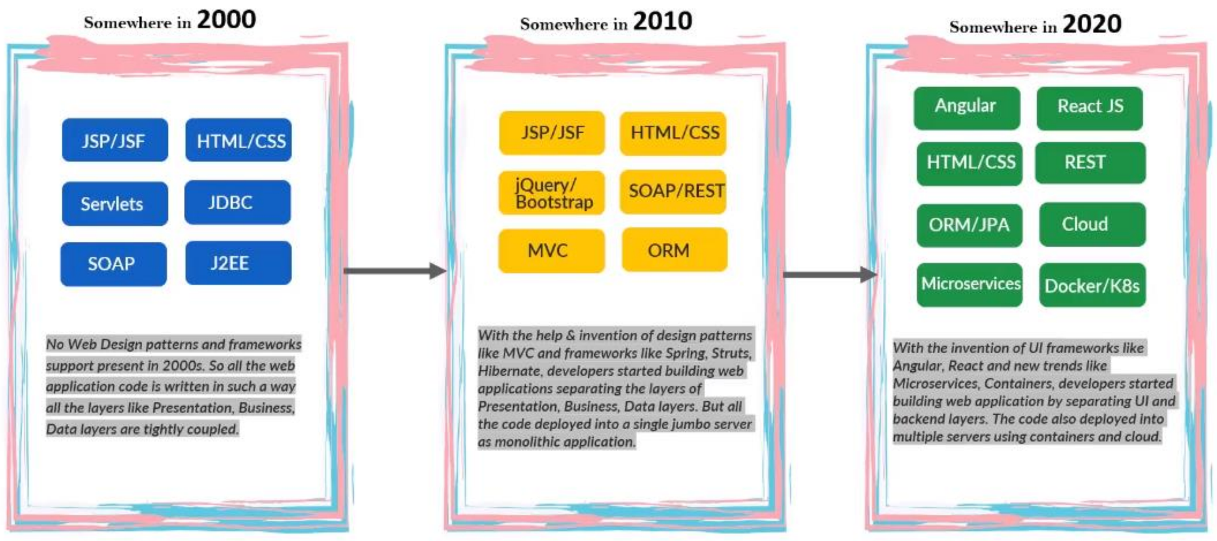
Spring MVC

1

Evolution of Web Applications inside Java Eco System



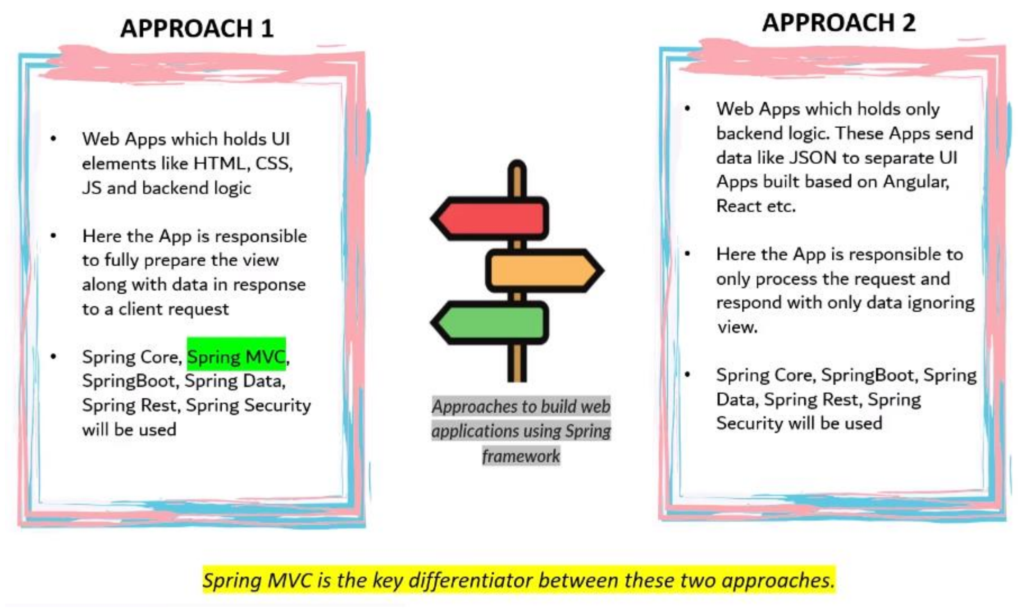
No Web Design patterns and frameworks support present in 2000s. So all the web application code is written in such a way all the layers like Presentation, Business, Data layers are tightly coupled.

With the help & invention of design patterns like MVC and frameworks like Spring, Struts, Hibernate, developers started building web applications separating the layers of Presentation, Business, Data layers. But all the code deployed into a single jumbo server as monolithic application.

With the invention of UI frameworks like Angular, React and new trends like Microservices, Containers, developers started building web application by separating UI and backend layers. The code also deployed into multiple servers using containers and cloud.

2

Web Applications using Spring



APPROACH 1

- Web Apps which holds UI elements like HTML, CSS, JS and backend logic
- Here the App is responsible to fully prepare the view along with data in response to a client request
- Spring Core, Spring MVC, SpringBoot, Spring Data, Spring Rest, Spring Security will be used

Approaches to build web applications using Spring framework

APPROACH 2

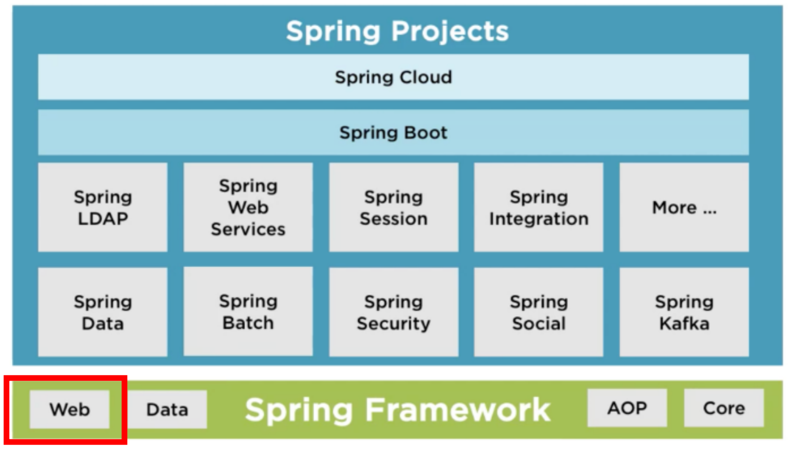
- Web Apps which holds only backend logic. These Apps send data like JSON to separate UI Apps built based on Angular, React etc.
- Here the App is responsible to only process the request and respond with only data ignoring view.
- Spring Core, SpringBoot, Spring Data, Spring Rest, Spring Security will be used

Spring MVC is the key differentiator between these two approaches.

3

Spring MVC

- Spring MVC, being a part of the Spring Framework, leverages the core features of Spring while providing specialized support for building web applications based on the MVC architecture.
- It is widely used for developing scalable and maintainable web applications in Java.



4

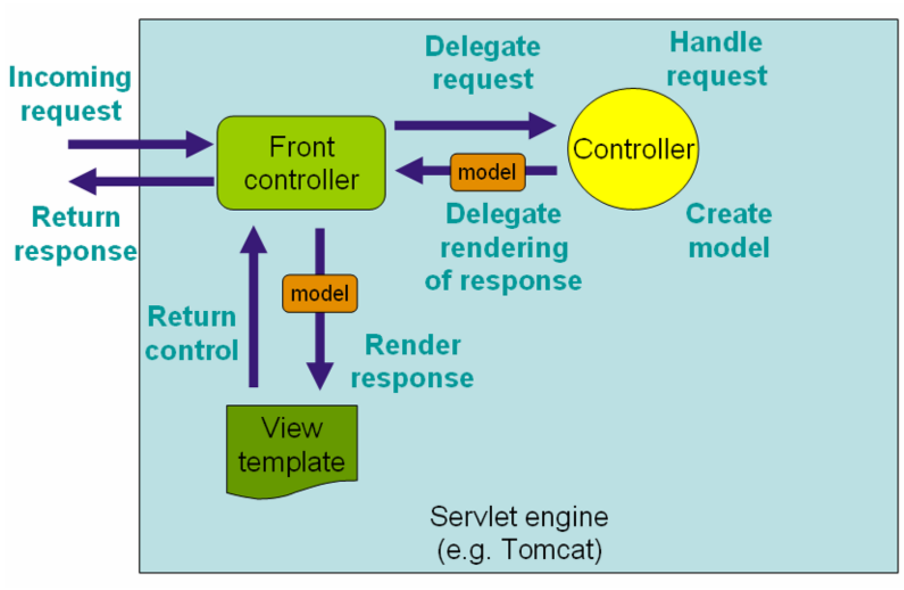
Dispatcher Servlet

- Spring MVC, like many other web frameworks, is designed around the front controller pattern where a central Servlet, the DispatcherServlet, provides a shared algorithm for request processing, while actual work is performed by configurable delegate components. This model is flexible and supports diverse workflows.
  - It is inherited from javax.servlet.http.HttpServlet
- The DispatcherServlet, as any Servlet, needs to be declared and mapped according to the Servlet specification by using Java configuration or in web.xml.
- A web application can define any number of `DispatcherServlet` instances. Each servlet will operate in its own namespace, loading its own application context with mappings, handlers.

Dispatcher Servlet

- The Dispatcher servlet is 'the thing' that decides which controller method the web request should be dispatched to.
  - Mapping a URL to the appropriate controller method is the primary duty of the Dispatcher servlet.
- This process of mapping a web request onto a specific controller method is called **request mapping**.
  - The Dispatcher servlet can do this with the help of the **`@RequestMapping`** **`(org.springframework.web.bind.annotation.RequestMapping)`** annotation.
- `DispatcherServlet` uses Spring configuration classes to discover the delegate components it needs for request mapping, view resolution, exception handling etc.

Dispatcher Servlet (Front Controller)



7

Java configuration to register and initialize the DispatcherServlet

```
public class MyWebApplicationInitializer implements WebApplicationInitializer {

  @Override
  public void onStartup(ServletContext servletContext) {

    // Load Spring web application configuration
    AnnotationConfigWebApplicationContext context = new AnnotationConfigWebApplicationContext();
    context.register(AppConfig.class);

    // Create and register the DispatcherServlet
    DispatcherServlet servlet = new DispatcherServlet(context);
    ServletRegistration.Dynamic registration = servletContext.addServlet("app", servlet);
    registration.setLoadOnStartup(1);
    registration.addMapping("/app/*");
  }
}
```

8

web.xml configuration to register and initialize the DispatcherServlet

```
<servlet>
     <servlet-name>app</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
         <param-name>contextConfigLocation</param-name>
        <param-value></param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
      <servlet-name>app</servlet-name>
      <url-pattern>/app/*</url-pattern>
</servlet-mapping>
```

9

View Resolvers

- A view resolver is a bean that helps the Dispatcher servlet to identify the views that have to be rendered as a response to a specific web request.
  - E.g. InternalResourceViewResolver
- It returns a URL which is used by the Dispatcher Servlet to get the view file
- Steps:
  - Dispatcher invoked the controller method -> dispatcher waits to get the logical view name -> it gives that view to the InternalResourceViewResolver -> gets the URL path of the actual view file -> the rendered view file is served to the client browser

10

Java configuration - InternalResourceViewResolver

```
@Configuration
public class AppConfig implements WebMvcConfigurer {

   // creating InternalResourceViewResolver bean
   @Bean
   public ViewResolver getInternalResourceViewResolver(){
      InternalResourceViewResolver viewResolver  = new InternalResourceViewResolver();

      // setting prefix and suffix to the path & extension
      viewResolver.setPrefix("/WEB-INF/jsp/");
      viewResolver.setSuffix(".jsp");
      return viewResolver;
   }
}
```

Prefix is the directory path from webapp ***/WEB-INF/jsp/*** (the directory where we will put the /jsp files) **and the suffix is the** ***.jsp*** **extension of the files.**

11

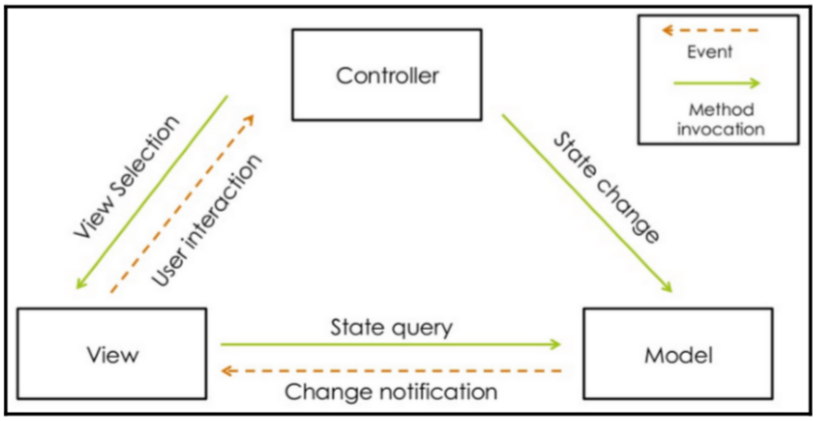
xml configuration - InternalResourceViewResolver

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

Prefix is the directory path from webapp ***/WEB-INF/jsp/*** (the directory where we will put the /jsp files) **and the suffix is the** ***.jsp*** **extension of the files.**

12

Classic MVC pattern



13

MVC example

Without Spring

```
// User.java (Model)
public class User {
    private String username;
    private String email;

    // Getters and setters
}
```

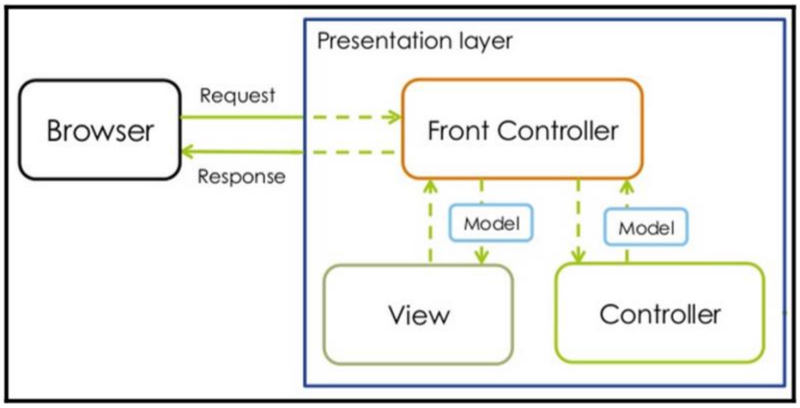
```
<!-- index.jsp (View) -->
<html>
<body>
   <h1>Welcome to our website!</h1>
   <form action="UserController" method="post">
      Username: <input type="text" name="username"><br>
      Email: <input type="text" name="email"><br>
      <input type="submit" value="Submit">
   </form>
</body>
</html>
```

```
// UserController.java (Controller)
public class UserController extends HttpServlet {
   protected void doPost(HttpServletRequest request,
                         HttpServletResponse response) {
     String username = request.getParameter("username");
     String email = request.getParameter("email");
     User user = new User();
     user.setUsername(username + "Example");
     user.setEmail(email);
     request.setAttribute("user", user);
         RequestDispatcher dispatcher =
         request.getRequestDispatcher("userDetails.jsp");
     dispatcher.forward(request, response);
   }
```

```
<!-- userDetails.jsp (View) -->
<html>
<body>
   <h1>User Details</h1>
   <p>Username: <%= request.getAttribute("user").getUsername() %></p>
   <p>Email: <%= request.getAttribute("user").getEmail() %></p>
</body>
</html>
```

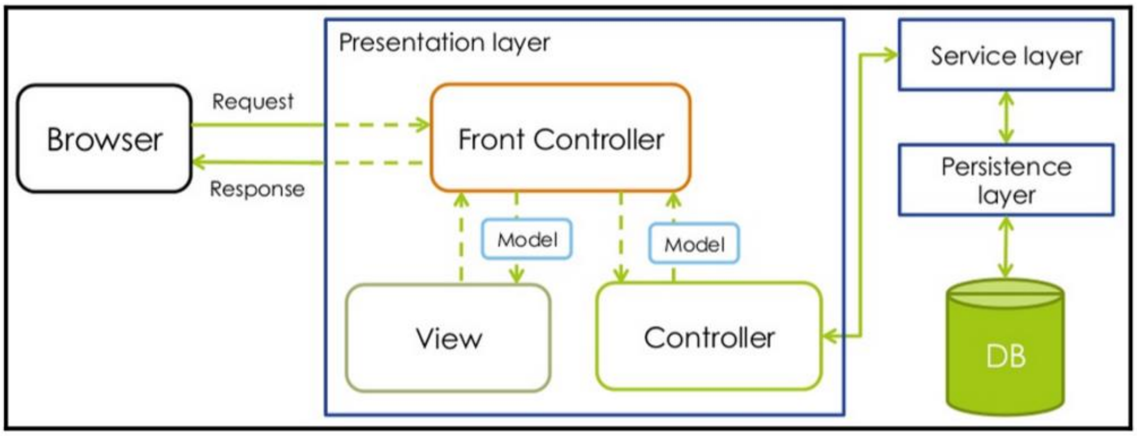
14

Spring MVC request flow



15

Layers of a Spring MVC application



16

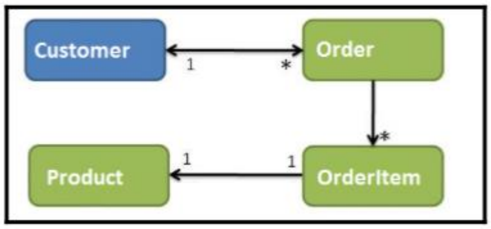
Application Architecture

- Source code should be organized into layers
  - improve reusability
  - loose coupling.
- A typical web application would normally have four layers:
  - presentation (Dispatcher servlet, controllers, view resolvers…),
  - domain,
  - services,
  - persistence.

17

Domain layer

- The Domain layer typically consists of a domain model.
- A domain model is a representation of the data storage types required by the business logic.
- It describes the various domain objects (entities), their attributes, roles, and relationships, plus the constraints that govern the problem domain.



18

Persistence layer

- Data retrieval from the database is not the duty of the controller:
  - The controller code will be large
  - The retrieval code can not be reused
- A Persistence layer usually contains repository objects to access domain objects
- A repository object:
  - sends queries to the data source for the data,
  - it maps the data from the data source to a domain object,
  - it persists the changes in the domain object to the data source.
- The @Repository (org.springframework.stereotype.Repository) annotation is an annotation that marks the specific class as a repository.
- A repository object is responsible for **CRUD** (Create, Read, Update, and Delete) operations on domain objects.

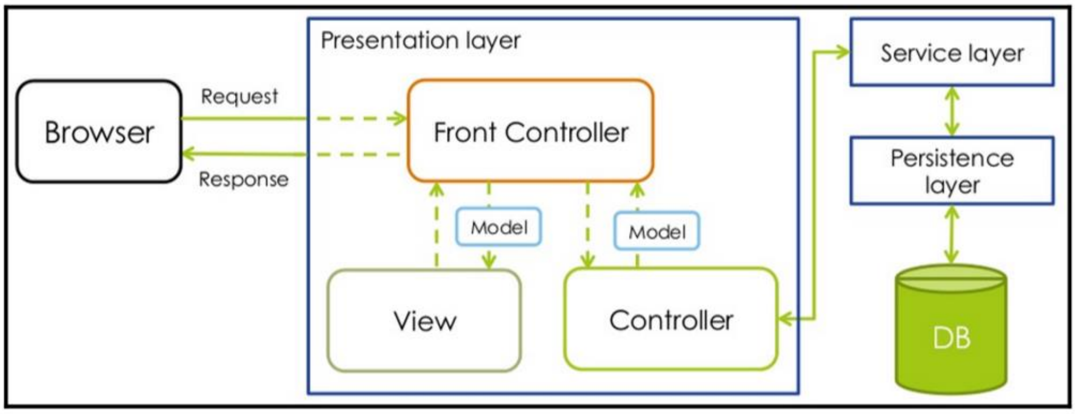
19

Service layer

- Where can we put the business operations code?
- The Service layer exposes business operations that could be composed of multiple CRUD operations
- E.g. you could have a business operation that would process a customer order, and in order to perform such a business operation, you would need to perform the following operations order:
  - First, ensure that all the products in the requested order are available in your store.
  - Second, ensure there are sufficient quantities of those products in your store.
  - Finally, update the product inventory by reducing the available count for each product ordered.
- Usually all CRUD operations for a single service call should be inside a **transaction**.

20

Layers of a Spring MVC application



21

- Controllers

22

Role of the controllers

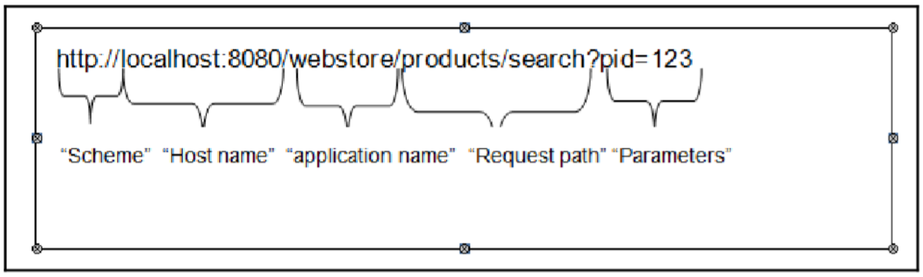
- In Spring MVC, the Controller's methods are the final destination point that a web request can reach.
- After being invoked, the Controller's method starts to process the web request by interacting with the Service layer to complete whatever work needs to be done.
- After the Service layer object completes the processing, the Controller is responsible for updating and building up the model object and chooses a View for the user to see next as a response.
- The model parameter can be updated and the View name can be returned with the help of the **ModelAndView** .

Defining a Controller

- Any regular Java classes can be transformed into a Controller by simply annotating them with the @Controller (org.springframework.stereotype.Controller) annotation.
  - This annotation tells Spring that this is a controller and will be used to handle requests.
- No need to extend from HttpServlet
- A Controller class is made up of request-mapped methods (also in short called handler methods). These handler methods are annotated with the @RequestMapping (org.springframework.web.bind.annotation.RequestMapping) annotation.

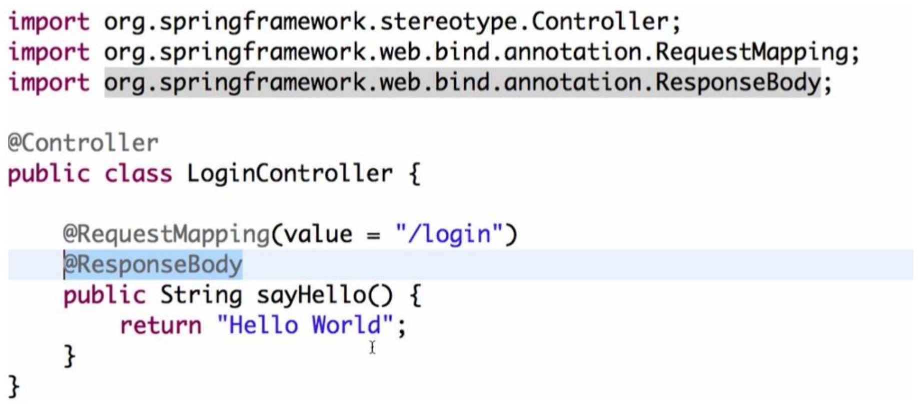
The logical parts of a typical Spring MVC application URL

- In a Spring MVC application, a URL can be logically divided into five parts
  - Scheme
  - Host name
  - Application name
  - Request path
  - Parameters
- The @RequestMapping annotation only matches against the URL request path; it will omit the scheme, host name, application name, and parameters.



25

Controller - Example



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

@Controller
public class LoginController {

    @RequestMapping(value = "/login")
    @ResponseBody
    public String sayHello() {
        return "Hello World";
    }
}
```

http://localhost:8080/app/login => will be mapped in this controller.

```
DispatcherServlet servlet = new DispatcherServlet(context);
ServletRegistration.Dynamic registration = servletContext.addServlet("app", servlet);
registration.setLoadOnStartup(1);
registration.addMapping("/app/*");
```

Java configuration to register and initialize the DispatcherServlet

26

Using URI template patterns (@PathVariable)

- E.g.
- http://localhost:8080/webstore/products/{category}
- Category will be converted to a variable called ‘path variable’

```
@RequestMapping("/products/{category}")
public String getProductsByCategory(Model model,
@PathVariable("category") String productCategory) {
…
  }
```

```
http://localhost:8080/webstore/products/Tablet
```

27

Request parameters (@RequestParam)

- E.g.

http://localhost:8080/webstore/market/product?id=P1234

```
 @RequestMapping("/product")
public String getProductById(@RequestParam("id") String productId,
Model model) {
```

Spring MVC will try to read a GET request parameter with the name id from the URL and will assign that to the getProductById method's parameter productId.

28

Using matrix variables

- E.g.

http://localhost:8080/webstore/market/products/filter/params;brands=Google,Dell;categories=Tablet,Laptop

```
@RequestMapping("/products/filter/{params}")
public String
getProductsByFilter(@MatrixVariable(pathVar="params")
Map<String,List<String>> filterParams, Model model) {
…
}
```

- Spring MVC will read all the matrix variables found in the URL after the {params} URI template and put them into the method parameter filterParams map.

29

Spring annotations for HTTP requests

- **@RequestMapping** is a generic annotation that can be used to map any HTTP request method (GET, POST, PUT, DELETE, etc.)
- Spring provides more specific annotations for common HTTP methods and scenarios.
  - **@GetMapping**:
    - Maps HTTP GET requests to handler methods.
    - Equivalent to **@RequestMapping(method = RequestMethod.GET)**.
  - **@PostMapping**:
    - Maps HTTP POST requests to handler methods.
    - Equivalent to **@RequestMapping(method = RequestMethod.POST)**.
  - **@PutMapping**:
    - Maps HTTP PUT requests to handler methods.
    - Equivalent to **@RequestMapping(method = RequestMethod.PUT)**.
  - **@DeleteMapping**:
    - Maps HTTP DELETE requests to handler methods.
    - Equivalent to **@RequestMapping(method = RequestMethod.DELETE)**.
  - **@PatchMapping**:
    - Maps HTTP PATCH requests to handler methods.
    - Equivalent to **@RequestMapping(method = RequestMethod.PATCH)**.

30

- Working with View Resolver

31

Spring MVC View

- Spring MVC maintains a high level of decoupling between the View and the Controller.
  - The Controller knows **nothing** about the View except the View name.
- It is the responsibility of the view resolver to map the correct View for the given View name.
- According to Spring MVC, a View is identifiable as an implementation of the `org.springframework.web.servlet.View` interface.

32

InternalResourceView/ InternalResourceViewResolver

- InternalResourceView (org.springframework.web.servlet.view.InternalResourceView) is a View for rendering a response as a JSP page.
- Spring comes with quite a few view resolvers to resolve various types of Views.
  - We have seen how to configure InternalResourceViewResolver as our view resolver to resolve JSP Views
  - InternalResourceViewResolver resolves a particular logical View name into a View
    - the Controller's method didn't return any actual View, it just returns the View name.
    - It is the role of InternalResourceViewResolver to form the correct URL path for the actual InternalResourceView.

33

Spring **Model** and **ModelMap**

- In Spring MVC, both Model and ModelMap are used to pass data from **controllers to views**, but they have slight differences in usage.
  - Model:
    - Model is an **interface** provided by Spring MVC that represents a data holder, allowing controllers to pass data to the view.
    - Typically, you add attributes to the Model object using its addAttribute() method.
    - The Model interface does not provide additional methods beyond adding attributes.
  - ModelMap:
    - ModelMap is a **class** provided by Spring MVC that extends LinkedHashMap and implements the Model interface.
    - Like Model, you can add attributes to the ModelMap object using its addAttribute() method.
    - In addition to addAttribute(), ModelMap provides additional methods inherited from LinkedHashMap for manipulating the attributes.
- When the DispatcherServlet receives a request, it identifies the appropriate controller handler method to handle the request based on the request mapping configured in the application. Before invoking the handler method, the DispatcherServlet creates a new instance of the Model (or ModelMap) object and passes it as a parameter to the handler method.
- After the controller handler method, finishes the execution, The rendered view, along with any model attributes, is sent back to the client's web browser, which displays the content to the user.

34

Spring Model and ModelMap - Example

```
CONTROLLER
@GetMapping("/example")
public String exampleHandler(Model model) {
   model.addAttribute("message", "Hello, world!");
   return "exampleView";
}
```

```
CONTROLLER
@GetMapping("/example")
public String exampleHandler(ModelMap model) {
   model.addAttribute("message", "Hello, world!");
   return "exampleView";
}
```

```
VIEW (exampleView.jsp)
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Example View</title>
</head>
<body>
   <h1>Example View</h1>
   <p>${message}</p>
</body>
</html>
```

35

ModelAndView

- ModelAndView is another class provided by Spring MVC for passing data from controllers to views, and it combines both the model data and view name into a single object.
  - It allows you to specify the view name and add model attributes in a single step.

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

@Controller
public class ExampleController {

   @GetMapping("/example")
   public ModelAndView exampleHandler() {
      ModelAndView modelAndView = new ModelAndView("exampleView");
      modelAndView.addObject("message", "Hello, world!");
      return modelAndView;
   }
}
```

```
VIEW (exampleView.jsp)
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Example View</title>
</head>
<body>
   <h1>Example View</h1>
   <p>${message}</p>
</body>
</html>
```

36

RedirectView

- URL redirection or forwarding is the technique of moving visitors to a different web page than the one they requested.
  - Most of the time, this technique is used after submitting a web form to avoid resubmission of the same form due to pressing the browser's back button or refresh.
- Spring MVC has a special View object that handles redirection and forwarding.
  - To use a RedirectView (org.springframework.web.servlet.view.RedirectView) with our Controller, we simply need to return the target URL string with the redirection prefix from the Controller.
- There are two redirection prefixes available in Spring MVC:
  - redirect
  - forward

37

Redirect vs. Forward

```
@RequestMapping("/welcome/greeting")
public String greeting() {
  return "welcome";
}
@RequestMapping("/")
public String welcome(Model model) {
  model.addAttribute("greeting", "Welcome to Web Store!");
  model.addAttribute("tagline", "The one and only amazing web store");
  return "welcome";
}
```

- **Exercise:**
  - **Case 1**: alter the return statement to:

    `return "forward:/welcome/greeting";`
  - **Case 2**: alter the return statement to:

    `return "redirect:/welcome/greeting";`

http://localhost:8080/webstore/

What happens?

Forward: the forwarded request is still the active original request -> *whatever value we put in the model at the start of the request would still be available*

Redirect: Spring will consider this request as a new request -> *whatever value we put in the model at the start of the original request will be gone*

38

Exercise

- Consider the following customer Controller:

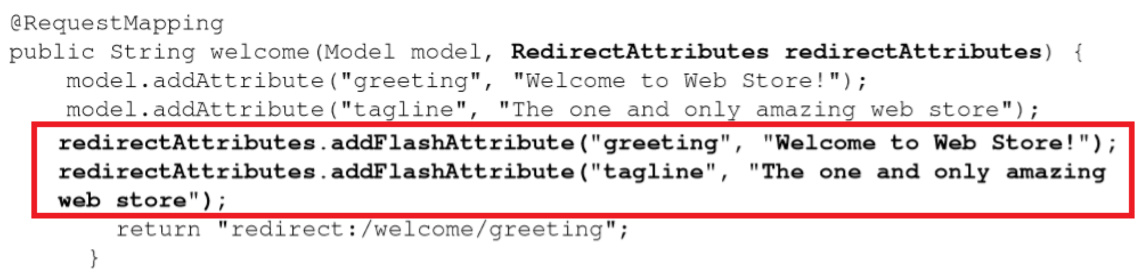
```
@Controller("/customers")
  public class CustomerController {
     @RequestMapping("/list")
    public String list(Model model) {
       return "customers";
    }
   @RequestMapping("/process")
   public String process(Model model) {
     // return
   } }
```

- If I want to redirect the list method from the process method, how should you form the return statement with the process method?

```
return "redirect:/customers/list"
```

Flash Attribute

- Flash attributes provide a way for us to store information that is intended to be used in another request.
- Flash attributes are saved temporarily in a session to be available for an **immediate** request after redirection.



```
@RequestMapping
public String welcome(Model model, RedirectAttributes redirectAttributes) {
    model.addAttribute("greeting", "Welcome to Web Store!");
    model.addAttribute("tagline", "The one and only amazing web store");
   redirectAttributes.addFlashAttribute("greeting", "Welcome to Web Store!");
   redirectAttributes.addFlashAttribute("tagline", "The one and only amazing
   web store");
        return "redirect:/welcome/greeting";
     }
```

Serving static resources

- Most of the time the View files contain dynamic content
- If we want to serve static content (e.g. images), it is not needed to pass the request through the controllers
- The directory to search for static content is specified in the WebApplicationContextConfig.java

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/img/**")
            .addResourceLocations("/resources/images/");
}
```

Mapping value

Directory of the static content
src/main/webapp/resources/images/

`http://localhost:8080/webstore/img/P1234.png`

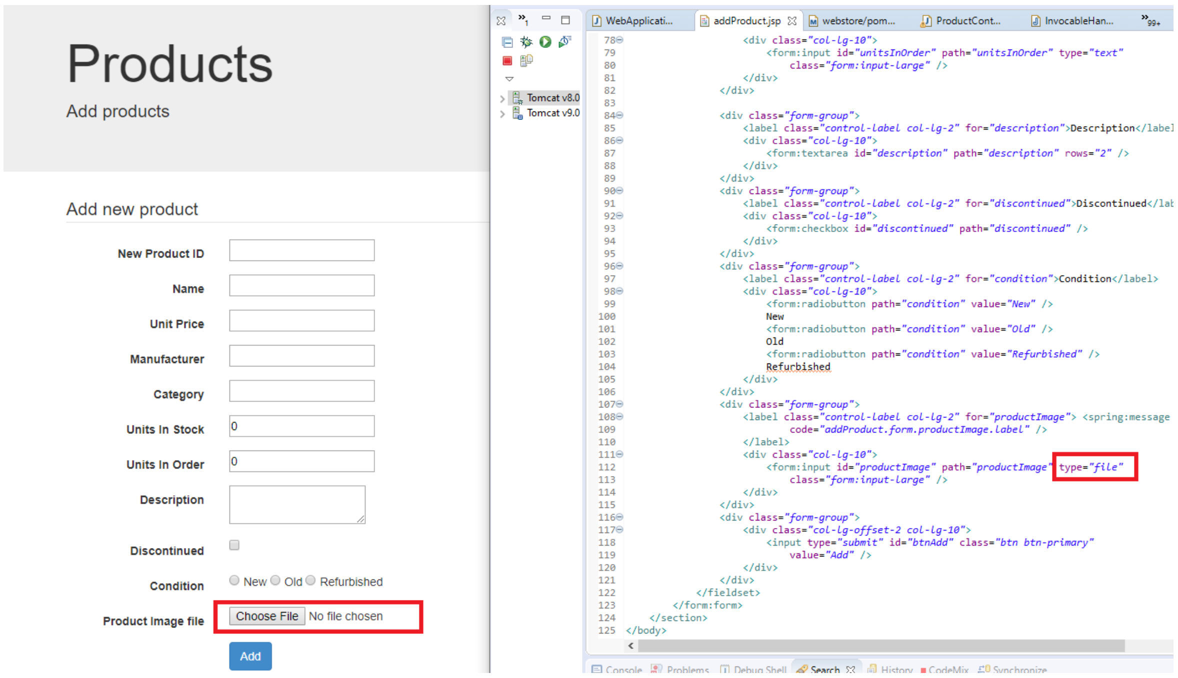
41

Multipart Requests

- A multipart request is a type of HTTP request to send files and data to the server.
- Spring's CommonsMultipartResolver (org.springframework.web.multipart.commons.CommonsMultipartResolver) class is the thing that determines whether the given request contains multipart content and parses the given HTTP request into multipart files and parameters.
- Create this bean in WebApplicationContextConfig.java

```
@Bean
public CommonsMultipartResolver multipartResolver() {
  CommonsMultipartResolver resolver=new CommonsMultipartResolver();
  resolver.setDefaultEncoding("utf-8");
  resolver.setMaxUploadSize(10240000);
  return resolver;
}
```

42



Products

Add products

Add new product

New Product ID	
Name	
Unit Price	
Manufacturer	
Category	
Units In Stock	0
Units In Order	0
Description	
Discontinued	☐
Condition	○ New ○ Old ○ Refurbished
Product Image file	Choose File No file chosen

Add

```
                <div class="col-lg-10">
                    <form:input id="unitsInOrder" path="unitsInOrder" type="text"
                        class="form:input-large" />
                </div>
            </div>

            <div class="form-group">
                <label class="control-label col-lg-2" for="description">Description</label
                <div class="col-lg-10">
                    <form:textarea id="description" path="description" rows="2" />
                </div>
            </div>
            <div class="form-group">
                <label class="control-label col-lg-2" for="discontinued">Discontinued</lat
                <div class="col-lg-10">
                    <form:checkbox id="discontinued" path="discontinued" />
                </div>
            </div>
            <div class="form-group">
                <label class="control-label col-lg-2" for="condition">Condition</label>
                <div class="col-lg-10">
                    <form:radiobutton path="condition" value="New" />
                    New
                    <form:radiobutton path="condition" value="Old" />
                    Old
                    <form:radiobutton path="condition" value="Refurbished" />
                    Refurbished
                </div>
            </div>
            <div class="form-group">
                <label class="control-label col-lg-2" for="productImage"> <spring:message
                        code="addProduct.form.productImage.label" />
                </label>
                <div class="col-lg-10">
                    <form:input id="productImage" path="productImage" type="file"
                        class="form:input-large" />
                </div>
            </div>
            <div class="form-group">
                <div class="col-lg-offset-2 col-lg-10">
                    <input type="submit" id="btnAdd" class="btn btn-primary"
                        value="Add" />
                </div>
            </div>
        </fieldset>
    </form:form>
  </section>
</body>
```

43

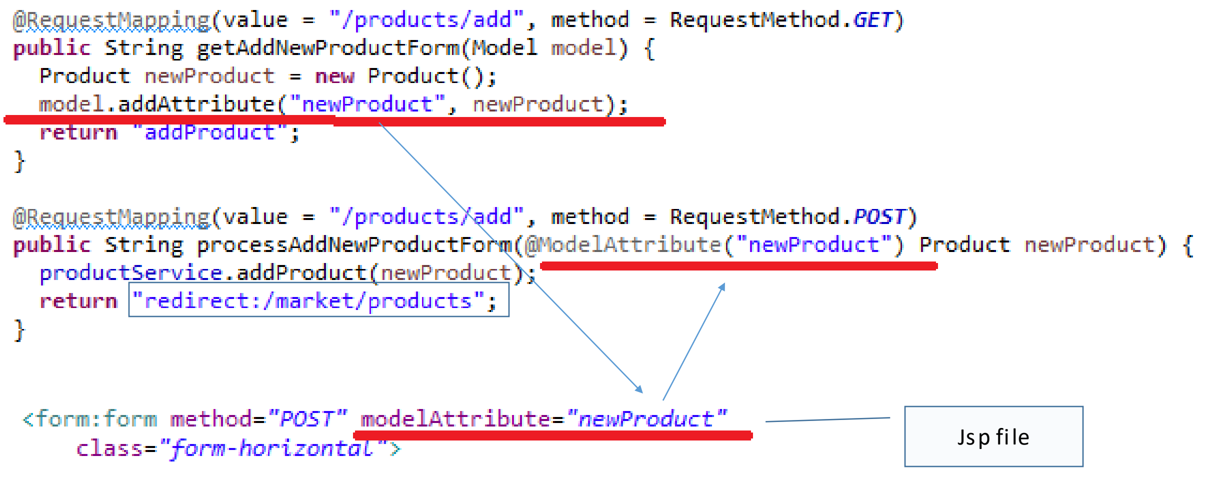
- Forms

44

Serving and Processing Forms

- How to store the data in database from the View?
  - How do we retrieve that data from the View in the Controller?
- In Spring MVC, the process of putting an HTML form element's values into model data is called **form binding.**
- The form backing bean is used to store form data.
- We can even use our domain objects as form beans;
  - This works well when there's a close match between the fields in the form and the properties in our domain object.
- Another approach is creating separate classes for form beans, which is sometimes called Data Transfer Objects (DTO).
- **@ModelAttribute** annotation is used to bind a method parameter or method return value to a model attribute.

@ModelAttribute – Example (1)



```
@RequestMapping(value = "/products/add", method = RequestMethod.GET)
public String getAddNewProductForm(Model model) {
   Product newProduct = new Product();
   model.addAttribute("newProduct", newProduct);
   return "addProduct";
}

@RequestMapping(value = "/products/add", method = RequestMethod.POST)
public String processAddNewProductForm(@ModelAttribute("newProduct") Product newProduct) {
   productService.addProduct(newProduct);
   return "redirect:/market/products";
}
```

```
<form:form method="POST" modelAttribute="newProduct"
     class="form-horizontal">
```

@ModelAttribute – Example (2)

JSP Form (exampleForm.jsp):

```
<form:form action="/submitForm" method="post"
modelAttribute="exampleModel">
      <label for="name">Name:</label>
      <input type="text" id="name" name="name">
      <br>
      <label for="email">Email:</label>
      <input type="email" id="email" name="email">
      <br>
      <input type="submit" value="Submit">
  </form>
```

```
@Controller
public class ExampleController {

   @PostMapping("/submitForm")
   public String submitForm(@ModelAttribute("exampleModel") ExampleModel exampleModel)
{
      // This method handles form submission and processes the model attributes
      // In a real application, you would typically perform some business logic here
      String name = exampleModel.getName();
      String email = exampleModel.getEmail();

     return "confirmationPage";
   }
}
```

47

Data Binding

- In order to do the binding, Spring MVC internally uses a special binding object called WebDataBinder
  - It is responsible for data binding between web request parameters and Java objects.
- WebDataBinder extracts the data out of the HttpServletRequest object and converts it to a proper data format, loads it into a form backing bean, and validates it.
- To customize the behavior of data binding, we can initialize and configure the WebDataBinder object in our Controller.
- We can explicitly tell Spring which fields are allowed for form binding.
- The @InitBinder annotation designates a Controller method as a hook method to do some custom configuration regarding data binding on WebDataBinder

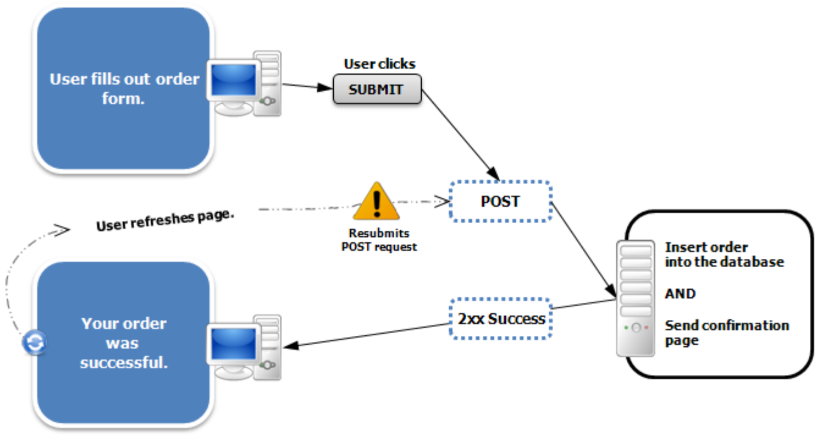
48

Customizing Data Binding

```
@InitBinder
public void initialiseBinder(WebDataBinder binder) {
   binder.setAllowedFields("productId",
            "name",
            "unitPrice",
            "description",
            "manufacturer",
            "category",
            "unitsInStock",
            "condition");
}
```

49

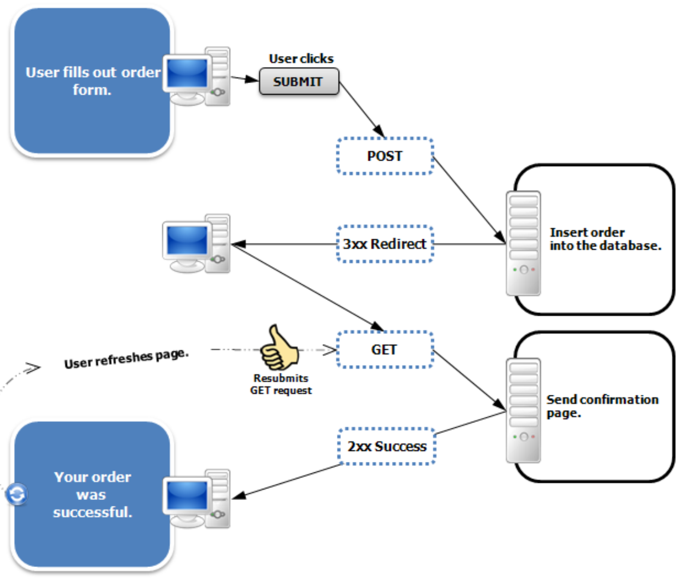
Double post problem when submitting forms



- https://en.wikipedia.org/wiki/Post/Redirect/Get

50

Double post problem when submitting forms - Solved



- https://en.wikipedia.org/wiki/Post/Redirect/Get

51

Bean Scopes

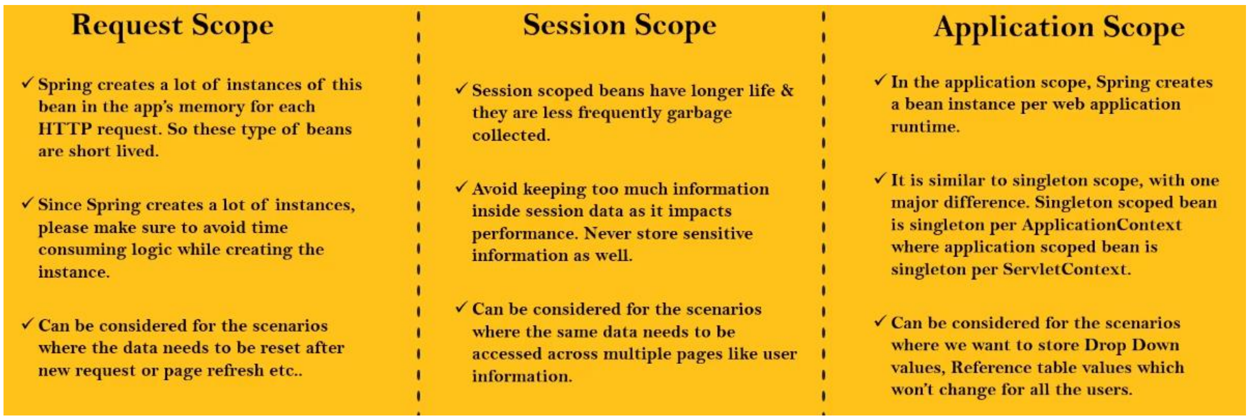
52

Bean Scopes inside Spring

- Singleton
- Prototype
- Scopes used only inside web applications:
  - Request
    - Spring creates an instance of the bean class for every HTTP request. The instance exists only for that specific HTTP request.
  - Session
    - Spring creates an instance and keeps the instance in the server's memory for the full HTTP session. Spring links the instance in the context with the client's session.
  - Application
    - The instance is unique in the app's context, and it is available while the app is running.

53

Key points of Spring Web Scopes



Request Scope

- ✓ Spring creates a lot of instances of this bean in the app's memory for each HTTP request. So these type of beans are short lived.
- ✓ Since Spring creates a lot of instances, please make sure to avoid time consuming logic while creating the instance.
- ✓ Can be considered for the scenarios where the data needs to be reset after new request or page refresh etc..

Session Scope

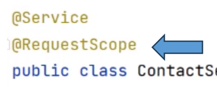
- ✓ Session scoped beans have longer life & they are less frequently garbage collected.
- ✓ Avoid keeping too much information inside session data as it impacts performance. Never store sensitive information as well.
- ✓ Can be considered for the scenarios where the same data needs to be accessed across multiple pages like user information.

Application Scope

- ✓ In the application scope, Spring creates a bean instance per web application runtime.
- ✓ It is similar to singleton scope, with one major difference. Singleton scoped bean is singleton per ApplicationContext where application scoped bean is singleton per ServletContext.
- ✓ Can be considered for the scenarios where we want to store Drop Down values, Reference table values which won't change for all the users.

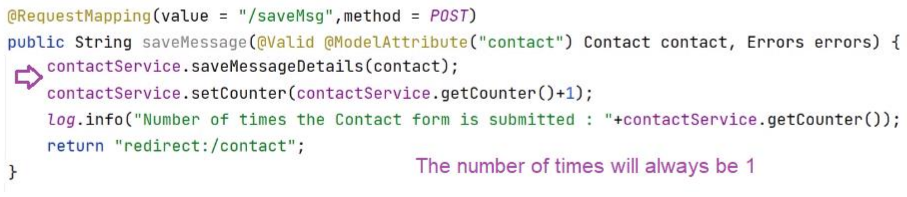
54

Spring Web Scopes - Example



```
@Service
@RequestScope
public class ContactService {
```

This bean will be created for every request.

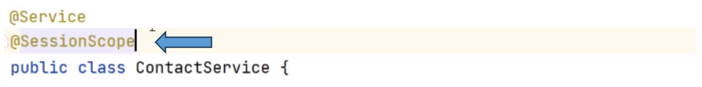


```
@RequestMapping(value = "/saveMsg",method = POST)
public String saveMessage(@Valid @ModelAttribute("contact") Contact contact, Errors errors) {
    contactService.saveMessageDetails(contact);
    contactService.setCounter(contactService.getCounter()+1);
    log.info("Number of times the Contact form is submitted : "+contactService.getCounter());
    return "redirect:/contact";
}
```

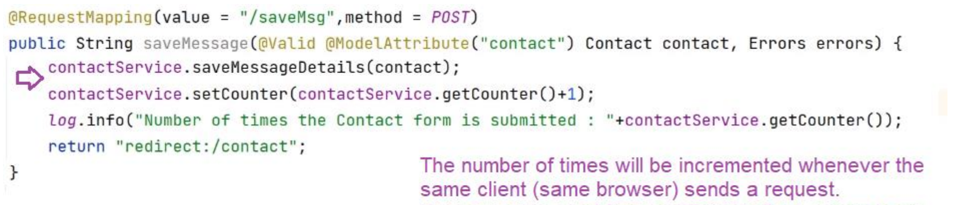
The number of times will always be 1

55

Spring Web Scopes - Example



```
@Service
@SessionScope
public class ContactService {
```

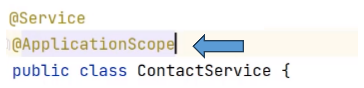


```
@RequestMapping(value = "/saveMsg",method = POST)
public String saveMessage(@Valid @ModelAttribute("contact") Contact contact, Errors errors) {
    contactService.saveMessageDetails(contact);
    contactService.setCounter(contactService.getCounter()+1);
    log.info("Number of times the Contact form is submitted : "+contactService.getCounter());
    return "redirect:/contact";
}
```

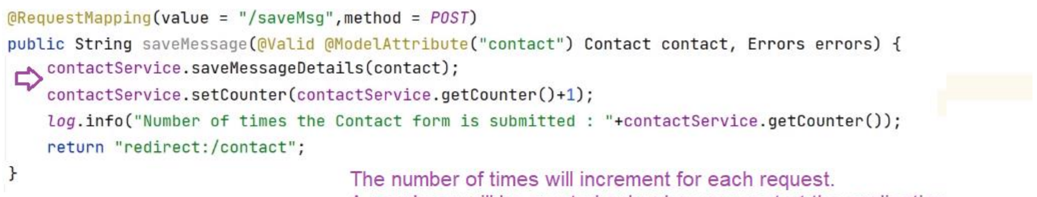
The number of times will be incremented whenever the same client (same browser) sends a request.
If a request is send from a different client, a new bean will be created.

56

Spring Web Scopes - Example



```
@Service
@ApplicationScope
public class ContactService {
```



```
@RequestMapping(value = "/saveMsg",method = POST)
public String saveMessage(@Valid @ModelAttribute("contact") Contact contact, Errors errors) {
    contactService.saveMessageDetails(contact);
    contactService.setCounter(contactService.getCounter()+1);
    log.info("Number of times the Contact form is submitted : "+contactService.getCounter());
    return "redirect:/contact";
}
```

The number of times will increment for each request.
A new bean will be created only when you restart the application.

57