FPT Education
FPT UNIVERSITY

Java

Spring MVC

Objectives

- Spring MVC Basics
- Spring MVC Framework
- Controller/Model/View
- Spring Interceptor
- Spring Validator

The MVC pattern

- MVC pattern breaks an application into three parts:
  - Model: The domain object model / service layer
  - View: Template code / markup
  - Controller: Presentation logic / action classes

- MVC defines interaction between components to promote separation of concerns and loose coupling
  - Each file has one responsibility
  - Enables division of labor between programmers and designers
  - Facilitates unit testing
  - Easier to understand, change and debug

Advantages of MVC Pattern

Separation of application logic and web design through the *MVC pattern*

- Integration with template languages
- Some MVC frameworks provide built-in components

- Other advantages include
  - Form validation
  - Error handling
  - Request parameter type conversion
  - Internationalization
  - IDE integration

Exploring Spring's Web MVC Framework

- The Spring Web MVC framework is built on generic Servlet known as a DispatcherServlet class (Front Controller).
- The DispatcherServlet class sends the request to the handlers with configurable handler mappings, theme resolution, locale and view resolution along with file uploading.
- The handleRequest(request, response) method is given by the default handler called Controller interface.
- The application controller implementation classes of the controller interface are as follows:
  - AbstractController
  - AbstractCommandController
  - SimpleFormController.

Spring MVC Features

- Powerful configuration of both framework and application classes.
- Separation of roles.
- Flexibility in choosing subclasses.
- Model transfer flexibility.
- No need of duplication of code.
- Specific validation and binding.
- Specific local and theme resolution.
- Facility of JSP form tag library.

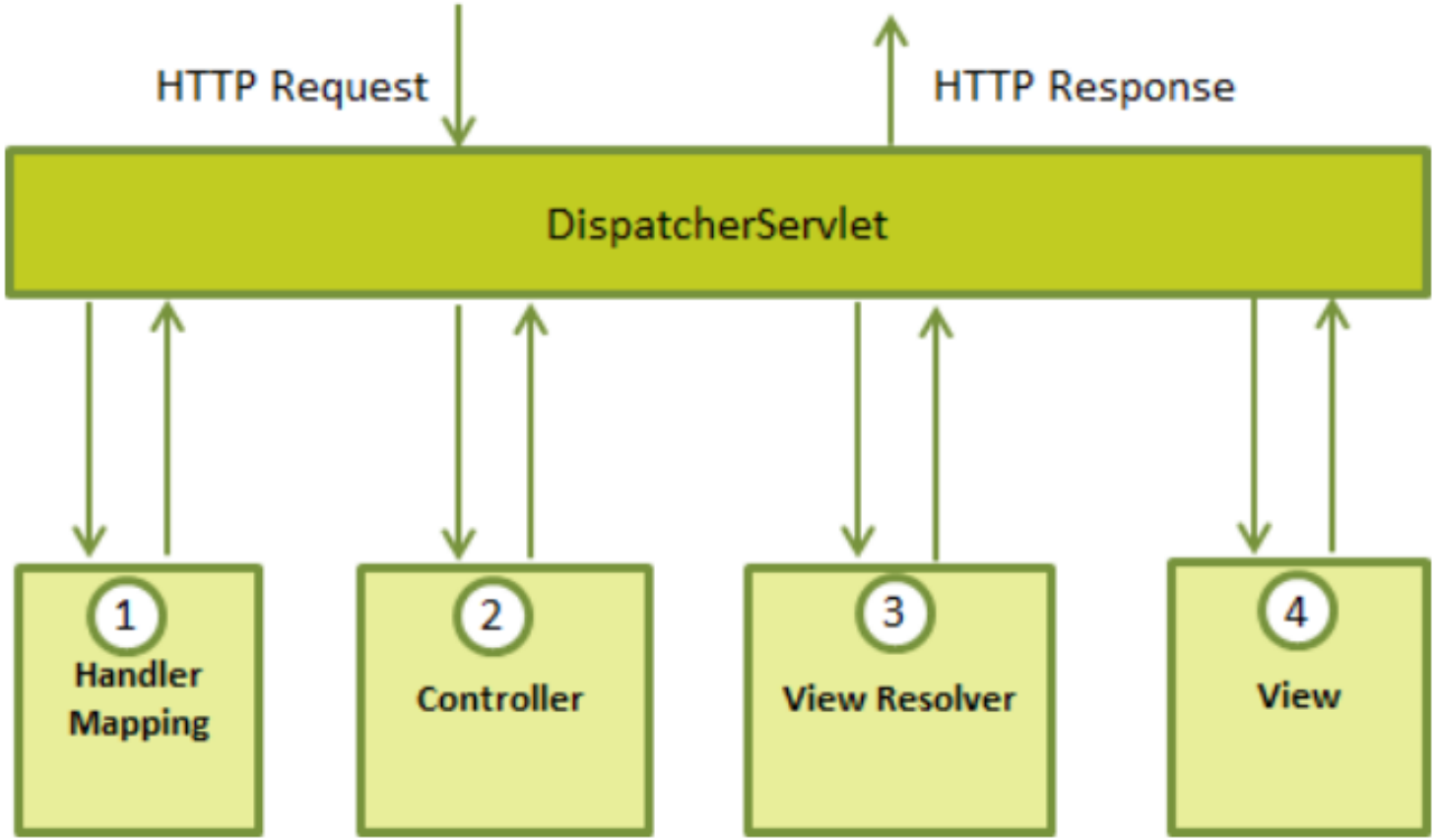
Spring Web MVC

- Spring Web MVC is the original web framework built on the Servlet API and has been included in the Spring Framework from the very beginning.
- The formal name, "Spring Web MVC," comes from the name of its source module (spring-webmvc), but it is more commonly known as "Spring MVC".

Spring Web MVC

- DispatcherServlet
- Filters
- Annotated Controllers
- Functional Endpoints
- URI Links
- Asynchronous Requests
- CORS
- Error Responses
- Web Security
- HTTP Caching
- View Technologies
- MVC Config
- HTTP/2

DispatcherServlet



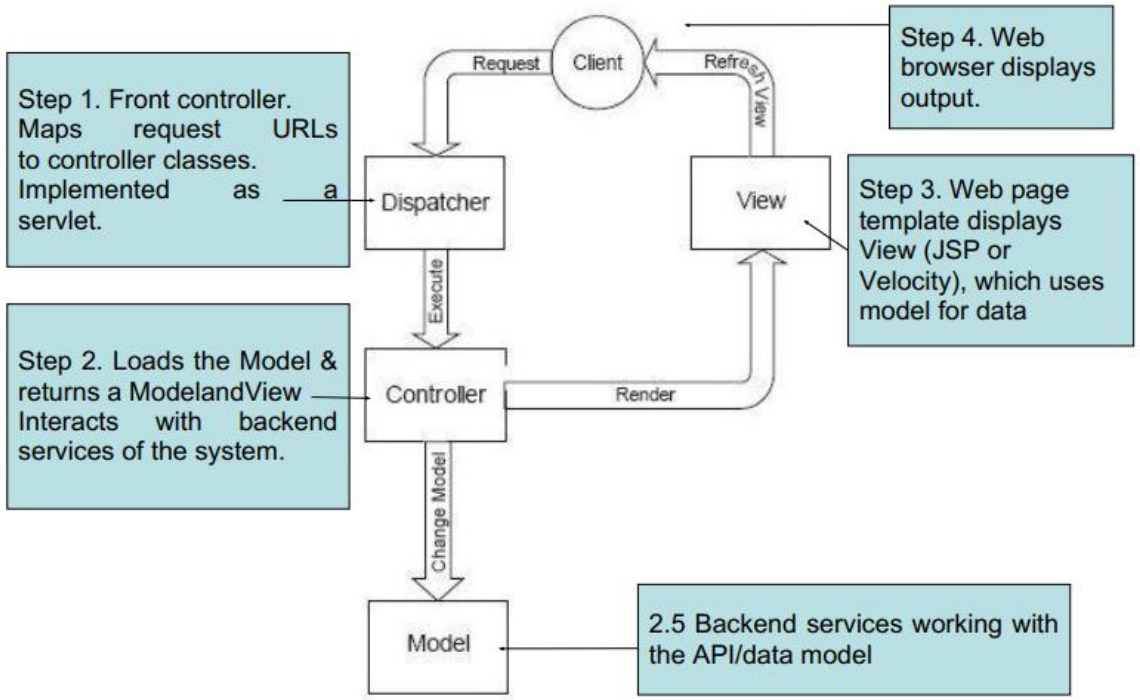
DispatcherServlet

Following is the sequence of events corresponding to an incoming HTTP request to *DispatcherServlet*

- After receiving an HTTP request, *DispatcherServlet* consults the *HandlerMapping* to call the appropriate *Controller*.
- The *Controller* takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the *DispatcherServlet*.
- The *DispatcherServlet* will take help from *ViewResolver* to pickup the defined view for the request.
- Once view is finalized, The *DispatcherServlet* passes the model data to the view which is finally rendered on the browser.

DispatcherServlet

- HandlerMapping, Controller, and ViewResolver are parts of *WebApplicationContext* which is an extension of the plain *ApplicationContext* with some extra features necessary for Web applications.



DispatcherServlet

- The DispatcherServlet class is important part of spring Web MVC framework.
- It is used for dispatching the request to application controllers.
- The DispatcherServlet class is configured in web.xml file of a web application.
- Map the request using Uniform Resource Locator (URL) mapping in the same *web.xml* file to handle any request

org.springframework.web.servlet

Class DispatcherServlet

java.lang.Object
- javax.servlet.GenericServlet
  - javax.servlet.http.HttpServlet
    - org.springframework.web.servlet.HttpServletBean
      - org.springframework.web.servlet.FrameworkServlet
        - org.springframework.web.servlet.DispatcherServlet

HadlerMapping/Controller/ViewResolver

- Handler mapping: Manages the execution of controllers, provided they match the specified criteria.
- Controller: It handles the client's request.
- View resolver: Resolves view names to view used by the DispatcherServlet.
  - The mapping between the Logical name and the Physical View Location is taken care by the View Resolver object.
  - Spring comes with a set of Built-In Spring Resolvers.
  - We can write Custom View Resolvers by implementing the org.springframework.web.servlet.ViewResolver interface

View Resolver

- BeanNameViewResolver
- FreeMarkerViewResolver
- InternalResourceViewResolver
- JasperReportsViewResolver
- ResourceBundleViewResolver
- UrlBasedViewResolver
- VelocityLayoutViewResolver
- VelocityViewResolver
- XmlViewResolver
- XsltViewResolver

View Resolver

- InternalResourceViewResolver

Controller returns - new ModelAndView("myView1")

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

- the prefix + the logical View Name + the suffix

  /WEB-INF/view/myView.jsp

Configure DispatcherServlet in web.xml

- Spring application context file, dispatcher-servlet.xml, will automatically be searched for and loaded by Spring for us.

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

Create Spring's Application Context XML File

- Web applications define servlets in web.xml
- Maps URL patterns to servlets
- WebApplicationContext is an extension of ApplicationContext for features of *Servlets* and *themes*

```
<web-app>
...
   <servlet>
      <servlet-name>mvc-dispatcher</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>

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

   <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/mvc-dispatcher-servlet.xml</param-value>
   </context-param>
...
</web-app>
```

The Spring DispatcherServlet

The URL to be "captured" by DispatcherServlet

Finds the file in WEB-INF *[servlet-name]-servlet.xml* to initiate beans

Overriding *DispatcherServlet* defaults

- DispatcherServlet initiates with default configuration. Overidding it through the *[servlet-name]-servlet.xml bean*
- Configuring *ViewResolver* is basic step

Different types of *ViewResolver*. Following 2 basic ones:

- *InternalResourceViewResolver (for jsp, css, images etc)*
- *ContentNegotiatingViewResolver (for ContentType response, useful for REST APIs)*
- If the Controller returns "**index**", InternalResourceViewResolver tries to find file as view **/WEBINF/pages/index.jsp**

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

Controllers

- The Controllers are central components of MVC
- Simply add **@Controller** annotation to a class
- Use **@RequestMapping** to map methods to url

```
<beans…>
        <context:component-scan
                base-package="fu.demo.spingmvc.controllers"/>
        ...
</beans>
```

→ Get all the **@Controller** annotated classes accessible as beans

```
@Controller
public class BaseController {

        @RequestMapping(value="/")
        public String welcome(ModelMap model) {

                model.addAttribute("message", "Whaddap!!");

                //Spring uses InternalResourceViewResolver and return back index.jsp
                return "index";
        }
...
}
```

Controllers

Controller Annotation

- The @Controller annotation defines the class as a Spring MVC controller.
- The @RequestMapping annotation is used to map URLs like '/hello' onto an entire class or a particular handler method.
- @RequestMapping(method = RequestMethod.GET) is used to declare the printHello() method as the controller's default service method to handle HTTP GET request.

Controller arguments

- @RequestParam, @PathVariable
- POJO – arbitrary java object that gets populated with request values
- @Valid – to enforce validation of the POJO
- BindingResult – to access the results of binding and validation
- Model, ModelMap, Map<String, ?> - acces to the model object
- Raw HttpServletRequest, response, session
- Locale, @RequestHeader, @RequestBody,...

Controllers – return types

- String, View, ModelAndView – use a “view“
- @ResponseBody – any object

```
@Controller @RequestMapping("/users")
public class UserController {
	@Inject private UserService service;
	@RequestMapping(value="/ajaxView“
consumes="application/json")
	@ResponseBody
	public User view(@RequestParam String username){
		User u = service.find(username);
		return u;
	}
}
```

More detailed Url Mapping

- @RequestMapping also accepts the following parameters:
  - method (GET/POST/PUT/DELETE...)
  - produces (mimeType)
  - consumes (mimeType)
  - params
  - headers

```
...
     @RequestMapping(value="/", method = RequestMethod.GET, produces = "text/html")
     public String welcome(ModelMap model) {

          model.addAttribute("message", "Whaddap!!");

          //Spring uses InternalResourceViewResolver and return back index.jsp
          return "index";
     }
...
}
```

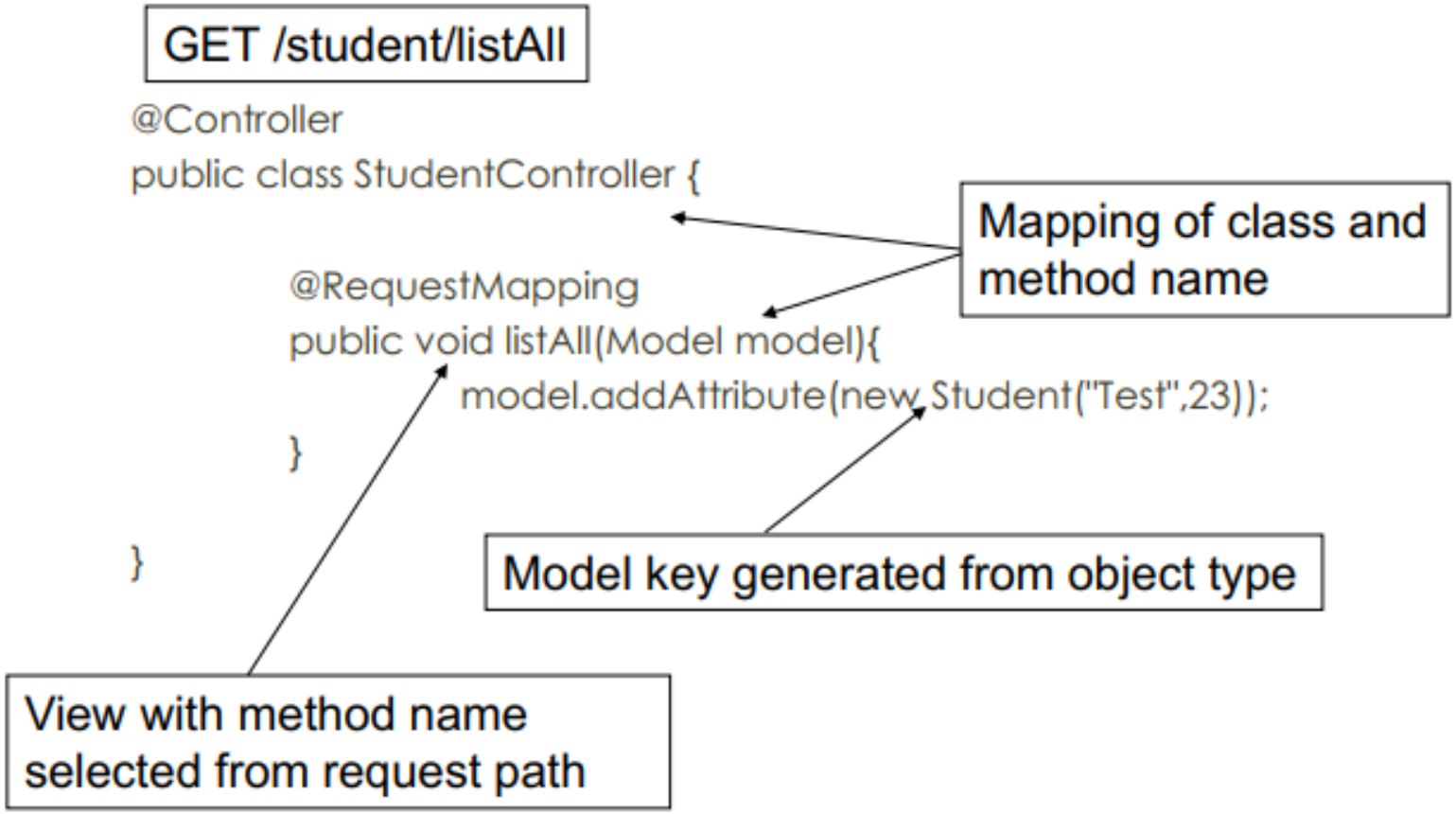
Url Templates in Controllers

- @PathVariable - to map variables in URL paths
- path variables can also be Regular Expressions
- Can also do as follows /message/*/user/{name}
- Can also use comma-separated URL parameters (also called Matrix-Variables)
  - To do this, make setRemoveSemicolonContent=false for *RequestMappingHandlerMapping*

```
// GET = /message/lars;friends=bob,rob,andy
@RequestMapping(value="/message/{name}", method = RequestMethod.GET)
public String welcome( @PathVariable String name,  @MatrixVariable String[] friends, ModelMap model) {

      model.addAttribute("message", "Hello " + name + " from " + friends[0] + " & " + friends[1]);
      //Spring uses InternalResourceViewResolver and return back index.jsp
      return "index";
}
```

Convention over Configuration



Convention over Configuration

- Mapping By Convention

```
@Controller
public class StudentController {

    URL → /student/showList ,  View → showList.jsp
    @RequestMapping
    public void showList(Model model){}


    URL → /student/getStudent,  View → getStudent.jsp
    @RequestMapping
    public Student getStudent(){
        ...
        return stu;
    }
}
```

@RestController vs. @Controller

Spring MVC REST Workflow

1. The client sends a request to a web service in URI form.
2. The request is intercepted by the DispatcherServlet which looks for Handler Mappings and its type.
   - The Handler Mappings section defined in the application context file tells DispatcherServlet which strategy to use to find controllers based on the incoming request.
   - Spring MVC supports three different types of mapping request URIs to controllers: annotation, name conventions, and explicit mappings.
3. Requests are processed by the Controller and the response is returned to the DispatcherServlet which then dispatches to the view.

@RestController vs. @Controller

Using the @ResponseBody Annotation

- When you use the @ResponseBody annotation on a method, Spring converts the return value and writes it to the http response automatically. Each method in the Controller class must be annotated with @ResponseBody.
- @ResponseBody annotation instructs Spring MVC to serialize the Student to the client.
- Spring MVC automatically serializes to JSON because the client accepts that content type

```
@RequestMapping(value="/student")
public @ResponseBody Student getStudent(
                                    @RequestParam int id) {
        return studentService.getStudent(id);
}
```

Model

- Controllers and view share a Java object referred as model, (‘M’ in MVC)
- A model can be of the type *Model* or can be a *Map* that can represent the model.
- The view uses this to display dynamic data that has been given by the controller

```
// Controller
@RequestMapping(value = "/{name}", method = RequestMethod.GET)
        public String welcome(@PathVariable String name, ModelMap model) {

        model.addAttribute("message", "Hello " + name);

        return "index";
}
```

- In View:

```
<html>
      body><h1>${message}</h1></body>
</html>
```

ModelAndView object in Spring MVC

- ModelAndView is an object that holds both the model and view. The handler returns the ModelAndView object and DispatcherServlet resolves the view using View Resolvers and View.

- The View is an object which contains view name in the form of the String and model is a map to add multiple objects.

```
ModelAndView model = new ModelAndView("employeeDetails");
model.addObject("employeeObj", new EmployeeBean(123));
model.addObject("msg", "Employee information.");
return model;
```

@ModelAttribute from Controller

- You can also use ***@ModelAttribute*** in controller to directly load URL value into the model
- A Model can represent objects that can be retrieved from database or files as well
- Model should not have logic, rather the controller should get the model and “transform” the model based on the request, while sending it to the View

View

- Spring MVC integrates with many view technologies:
  - JSP
  - Velocity
  - Freemarker
  - JasperReports
- Values sent to controller with POST or GET as usual
- Values made available to the view by the controller

Spring's form tag library

- Binding-aware JSP tags for handling form elements
- Integrated with Spring MVC to give the tags access to the model object and reference data
- Comes from spring-webmvc.jar
- Add the following to make the tags available:
- `<%@ taglib prefix="form" uri="http://www.springfra`

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

The attributes: *path* & *modelAttribute/commandName*

- commandName/modelAttribute: name of a variable in the request scope or session scope that contaims the information about this form, it should be a been.
- path: name of a bean property that should be accessed in order to pass the information to from and to the controller.

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
<title>Spring MVC Form Handling</title>
</head>

<body>
  <h2>Employee</h2>
  <form:form method="POST" action="addEmployee" modelAttribute="employee">
    Id: <form:input path="id" />
    <br/>
    Name: <form:input path="name" />
    <input type="submit" value="Submit" />
  </form:form>
</body>
</html>
```

The *form* tag

- Renders a form tag and exposes the binding to inner tags
- You can specify any HTML attributes that are valid for an HTML form
- You can also tell it what the form backing object is (it uses 'command' by default).

```
@RequestMapping("/person/add")
public String addPerson(Model model) {
        Person person = new Person();
        model.addAttribute(person);
        return "addPerson";
}
```

```
<form:form method="get" commandName="person">
        <form:input path="name" />
</form:form>
```

```
<form method="get">
        <input type="text" name="name" />
</form>
```

The *input* tag

- Renders an HTML input tag with a type of ‘text’
- You can specify any HTML attributes valid for an HTML input
- You bind it to your model object by specifying the path relative to the backing object

```
@RequestMapping("/person/add")
public String addPerson(Model model) {
        Person person = new Person();
        person.setName(”Spencer”);
        model.addAttribute(person);
        return "addPerson";
}
```

```
<form:form method=”get” commandName=”person”>
        <form:input path=”name” />
</form:form>
```

```
<form method=”get”>
        <input type=”text” name=”name” value=”Spencer” />
</form>
```

The *checkbox* tag

- Renders an HTML input tag with a type of ‘checkbox’

```
public class Person {
        private boolean admin;
        private String[] languages;
}
```

```
<form:form commandName=”person”>
    <form:checkbox path=”admin” />
    <form:checkbox path=“languages” value=“Java” />
    <form:checkbox path=“languages” value=“Scala” />
</form:form>
```

```
<form>
    <input type=”checkbox” name=”admin” value=”true” />
    <input type=”checkbox” name=”languages” value=”Java” />
    <input type=”checkbox” name=”languages” value=”Scala” />
</form>
```

The *checkboxes* tag

- Similar to *checkbox* tag, but creates multiple checkboxes instead of one

```
public class Person {
        private boolean admin;
        private String[] favoriteLanguages;
        private List<String> allLanguages;
}
```

```
<form:form commandName="person">
    <form:checkbox path="admin" />
    <form:checkboxes path="favoriteLanguages" items="${allLanguages}" />
</form:form>
```

```
<form>
    <input type="checkbox" name="admin" value="true" />
    <input type="checkbox" name=" favoriteLanguages" value="Java" />
    <input type="checkbox" name=" favoriteLanguages" value="Scala" />
</form>
```

The *password* tag

- Renders an HTML input tag with a type of ‘password’

```
<form:form>
        <form:input path=”username” />
        <form:password path=”password” />
</form:form>
```

```
<form>
        <input type=”text” name=”username” />
        <input type=”password” name=”password” />
</form>
```

The *select* tag

- Renders an HTML select tag
- It can figure out whether multiple selections should be allowed
- You can bind options using this tag, as well as by nesting *option* and *options* tags

```
<form:select path="favoriteLanguage" items="${allLanguages}" />
```

```
<select name="favoriteLanguage">
        <option value="Java">Java</option>
        <option value="Scala">Scala</option>
</form>
```

The *option/options* tags

- Renders an HTML option (or multiple options)
- Nested within a select tag
- Renders ‘selected’ based on the value bound to the select tag

```
<form:select path=”favoriteLanguage”>
        <form:option value=”3” label=”.NET” />
        <form:options items=”${languages}” itemValue=”id” itemLabel=”name”
/>
</form:select>
```

```
<select name=”favoriteLanguage”>
        <option value=”.NET”>.NET</option>
        <option value=”Java”>Java</option>
        <option value=”Scala”>Scala</option>
</form>
```

The *errors* tag

- Renders an HTML span tag, containing errors for given fields
- You specify which fields to show errors for by specifying a path
  - path="name" – Would display errors for name field.
  - path="*" – Would display all errors

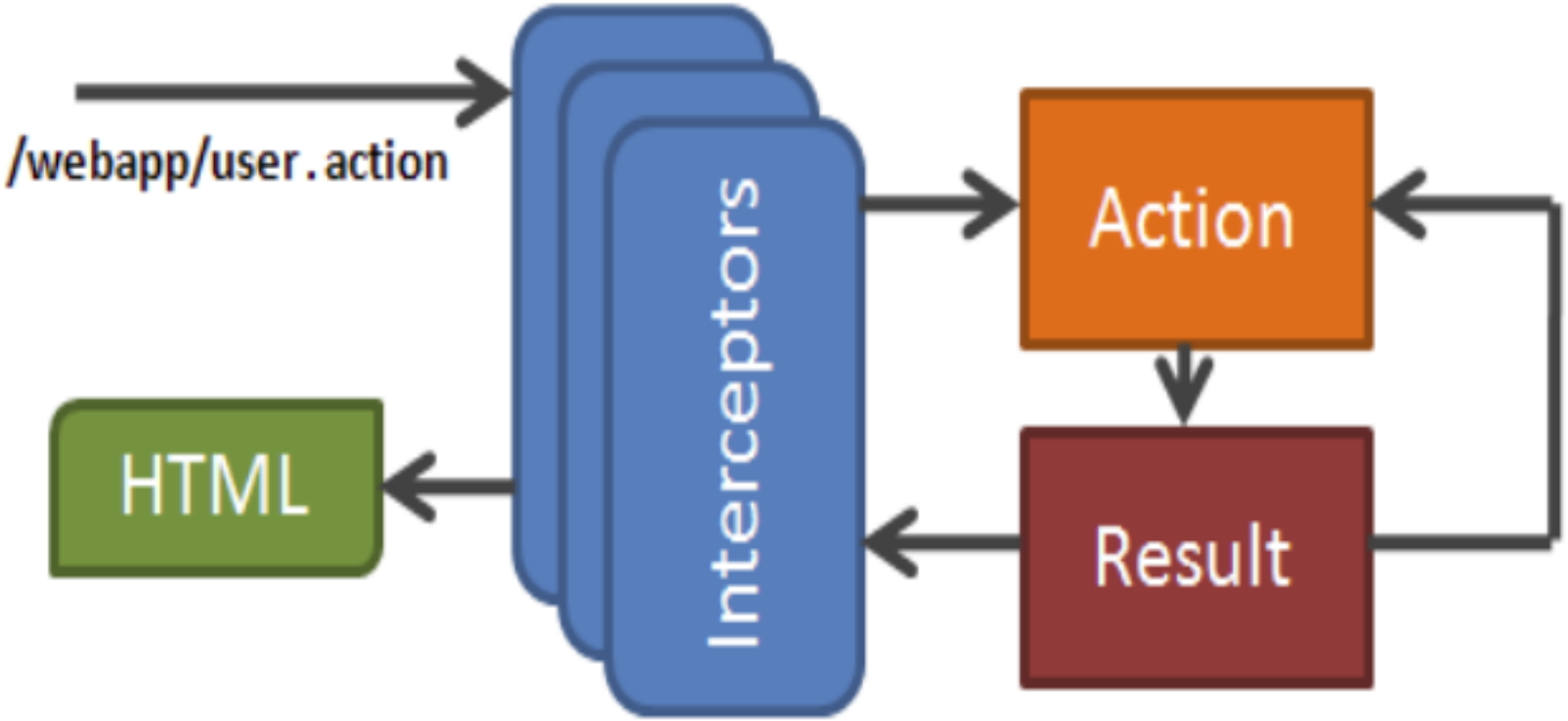
```
<form:errors path="*" cssClass="errorBox" />
<form:errors path="name" cssClass="specificErrorBox" />
```

```
<span name="*.errors" class="errorBox">Name is required.</span>
<span name="name.errors" class="specificErrorBox">Name is required.</span>
```

Spring Exception

- @ExceptionHandler only handles exception getting raised from the controller where it is defined.
- It will not handle exceptions getting raised from other controllers. @ControllerAdvice annotation solves this problem.
- @ControllerAdvice annotation is used to define @ExceptionHandler,@InitBinder, and @ModelAttribute methods that apply to all @RequestMapping methods.

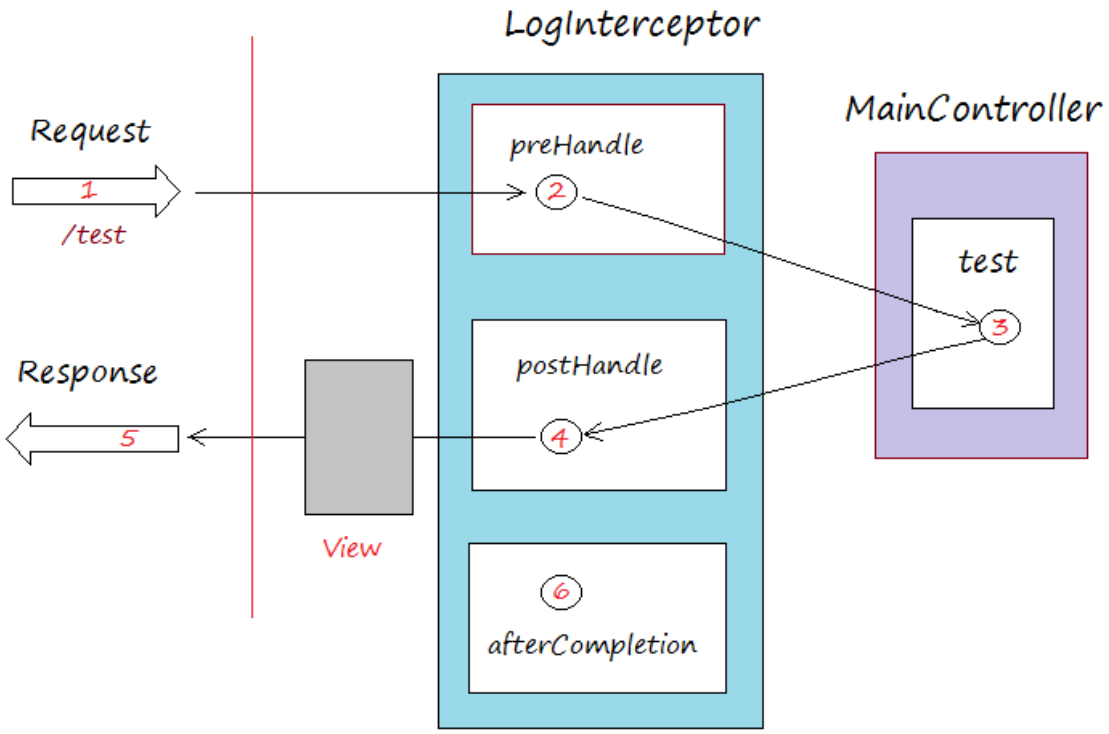
Spring Interceptor

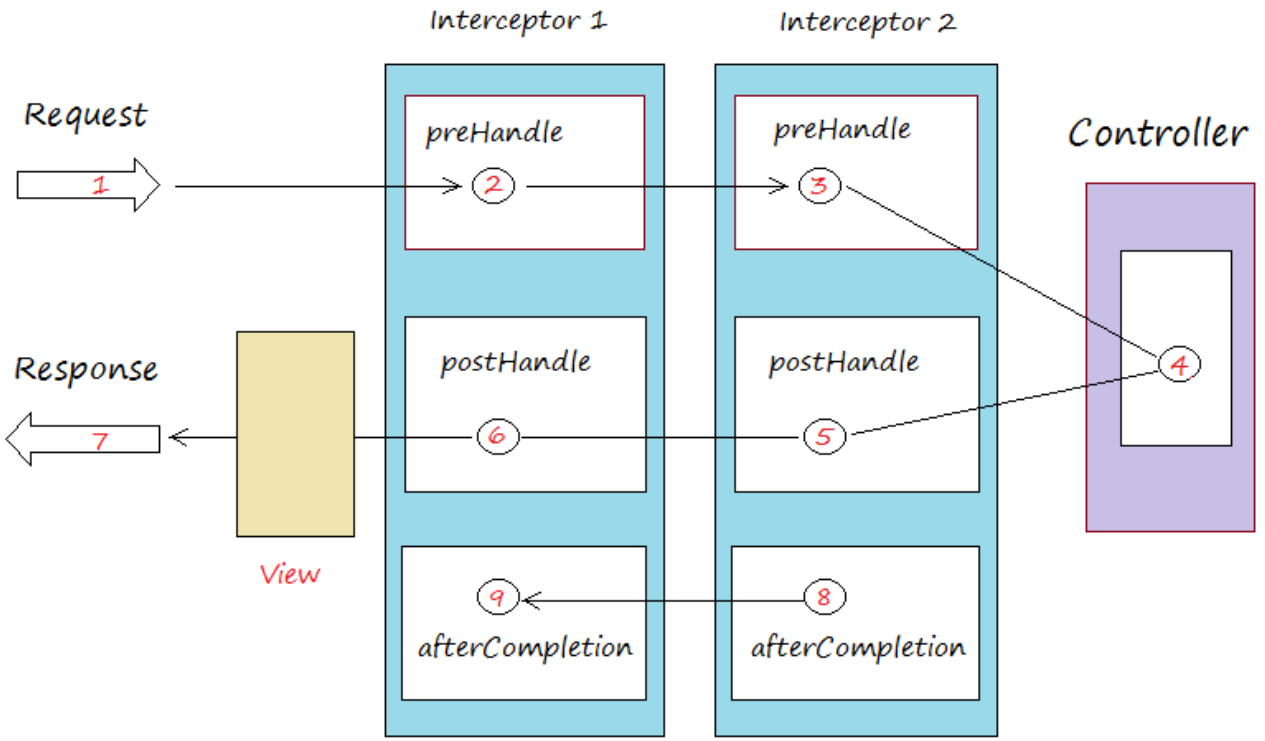


Spring Interceptor

- Spring MVC provides a powerful mechanism to intercept an http request
- Each interceptor you define must implement *org.springframework.web.servlet.HandlerInterceptor* interface.
- HandlerInterceptor – an interface, which must be implemented by the Spring interceptor classes, has the following three methods.
  - preHandle(...) – called just before the controller
  - postHandle(...) – called immediately after the controller
  - afterCompletion(...) – called just before sending response to view
- HandlerInterceptorAdaptor – an implementation class of HandlerInterceptor interface provided by Spring as a convenient class. By extending this we can override only the necessary methods out of the three.

Spring Interceptor





Locales

- LocaleResolver
  - AcceptHeaderLocaleResolver
  - CookieLocaleResolver
  - SessionLocaleResolver
  - LocaleChangeInterceptor

```
<bean id="localeChangeInterceptor"
class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
        <property name="paramName" value="lang"/>
</bean>
```

*http://localhost:8080/carbase/?**lang=ru***

Spring Validation

- Spring provides a simplified set of APIs and supporting classes for validating domain objects.
- Spring features a Validator interface that you can use to validate objects. The Validator interface works using an Errors object so that while validating, validators can report validation failures to the Errors object.
- *Validation using Spring's Validator interface*

```
public interface Validator {

    /** Can this instances of the supplied clazz */
    boolean supports(Class<?> clazz);

    /**
    * Validate the supplied target object, which must be
    * @param target the object that is to be validated
    * @param errors contextual state about the validation process
    */
    void validate(Object target, Errors errors);
}
```

Spring Validation

- Bean Validation
  - Standard Constraints

Annotation	Type	Description
`@Min(10)`	Number	must be higher or equal
`@Max(10)`	Number	must be lower or equal
`@AssertTrue`	Boolean	must be true, null is valid
`@AssertFalse`	Boolean	must be false, null is valid
`@NotNull`	any	must not be null
`@NotEmpty`	String / Collection's	must be not null or empty
`@NotBlank`	String	`@NotEmpty` and whitespaces ignored
`@Size(min,max)`	String / Collection's	must be between boundaries
`@Past`	Date / Calendar	must be in the past
`@Future`	Date / Calendar	must be in the future
`@Pattern`	String	must math the regular expression

```
    @Size(min=2, max=30)
private String name;

@NotEmpty @Email
private String email;

@NotNull @Min(18) @Max(100)
private Integer age;

@NotNull
private Gender gender;

@DateTimeFormat(pattern="MM/dd/yyyy")
@NotNull @Past
private Date birthday;

@Phone
private String phone;
```

import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Past;
import javax.validation.constraints.Size;
import org.hibernate.validator.constraints.Emai
import org.hibernate.validator.constraints.NotE
import org.springframework.format.annotation.D
@Phone: defined custom validator.

@Autowired

- @Autowired on Setter Methods
- @Autowired on Properties
- @Autowired on Constructors
- @Autowired on Constructors

```java
public class MyController {

    private MyService myService;

    public MyController(MyService aService) { // constructor based injection
        this.myService = aService;
    }

    public void setMyService(MySerice aService) { // setter based injection
        this.myService = aService;
    }

    @Autowired
    public void setMyService(MyService aService) { // autowired by Spring
        this.myService = aService;
    }

    @RequestMapping("/blah")
    public String someAction()
    {
        // do something here
        myService.foo();

        return "someView";
    }
}
```

Magic tags

- <mvc:annotation-driven>
- <mvc:interceptors>
- <mvc:view-controller>
- <mvc:resources>
- <mvc:default-servlet-handler>
- <context:component-scan>

Magic tags

<mvc:annotation-driven>

- registers necessary beans
- support formatting
  - Number fields using the @NumberFormat
  - Date, Calendar, Long fields using the @DateTimeFormat
- support for reading and writing
  - XML, if JAXB is present in classpath
  - JSON, if Jackson is present in classpath
  - support validating with @Valid

Magic tags

```
<mvc:interceptors>
<!-- register "global" interceptor beans to apply to all registered HandlerMappings -->
<mvc:interceptors>
        <!–- applied to all URL paths -->
        <bean
class="org.springframework.web.servlet.theme.ThemeChangeInterceptor"/>
        <!–- applied to a specific URL path -->
        <mvc:interceptor>
                <mvc:mapping path="/secure/*"/>
                <bean class="org.example.MyInterceptor" />
        </mvc:interceptor>
</mvc:interceptors>
```

Magic tags

- <mvc:view-controller>: immediately forwards to a view when invoked

```
<mvc:view-controller path="/" view-name="index"/>
<mvc:view-controller path="/resourceNotFound"/>
```

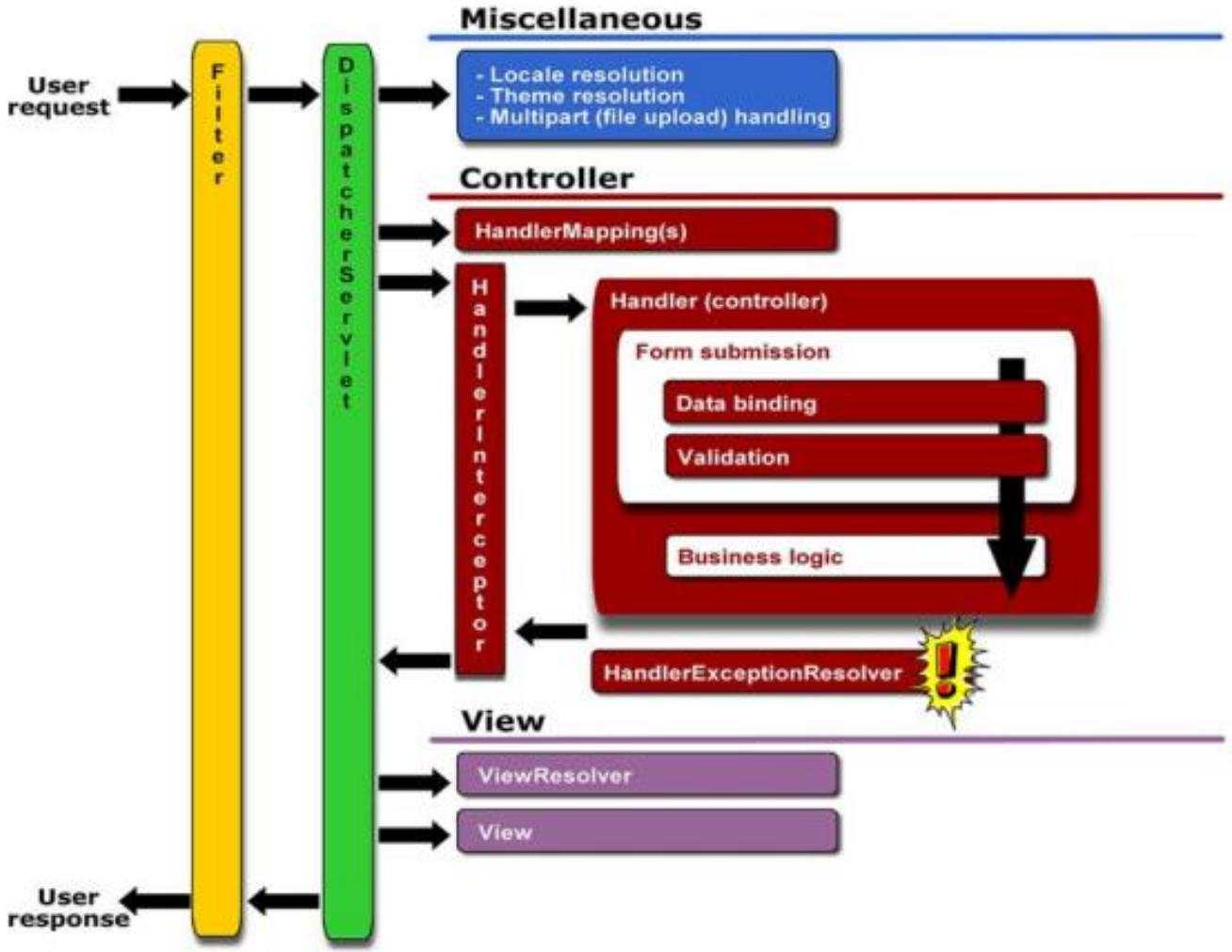
- <mvc:resources>: handles HTTP GET requests for /resources/** by efficiently serving up static resources <mvc:**resources** location="/, classpath:/META-INF/web-resources/“ mapping="/resources/**"/>

- <mvc:default-servlet-handler>: allows for mapping the DispatcherServlet to "/" by forwarding static resource requests to the container's default Servlet <mvc:**default-servlet-handler**/>

Magic tags

- <context:component-scan>: activates the annotations and scans the packages to find and register beans within the application context (will be use to activate Spring MVC annotation scanning capability which allows to make use of annotations like @Controller and @RequestMapping etc.)

Spring MVC request life cycle



Summary

Concepts were introduced:

- Spring MVC Basics
- Spring MVC Framework
- Controller/Model/View
- Spring Interceptor
- Spring Validator