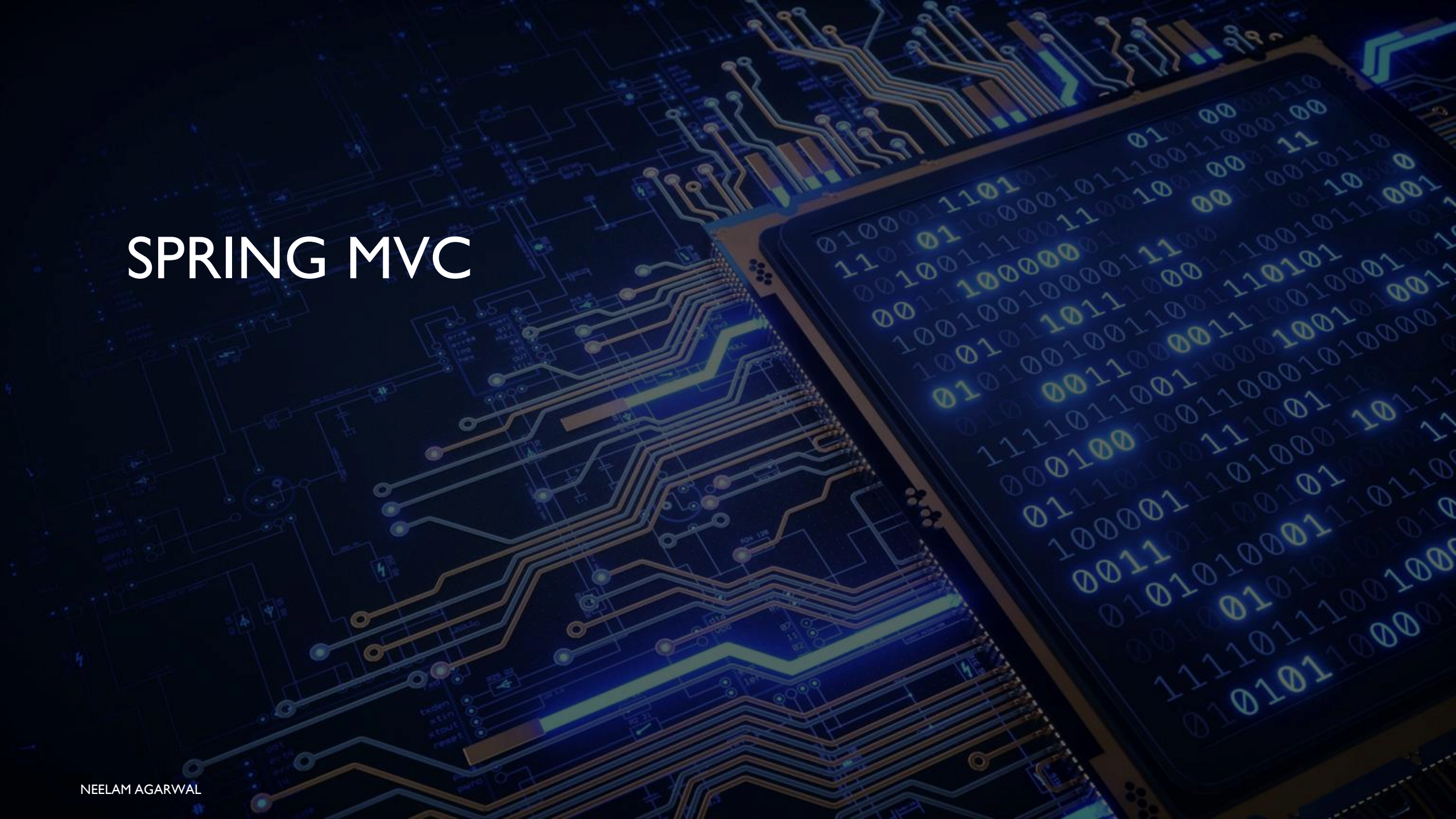
SPRING MVC



NEELAM AGARWAL

OUTLINE

- Introduction to MVC Design Pattern
- Configuring Spring in a Web application using Spring MVC
- DispatcherServlet front controller
- Defining Spring MVC controllers using annotations
- Spring MVC in the view layer
- Form rendering and type conversion
- Form validation using Spring and Bean validation

INTRODUCTION TO SPRING MVC

NEELAM AGARWAL

WHAT IS MVC?

- Clearly separates business, navigation and presentation logic.
- MVC is commonly used design pattern that can be applied to many different architectures like GUI, Web application etc.
- MVC suggests that the application is broken up as
  - Model
    - Manage the data of the application
    - The contract between the controller and the view
    - Contains the data needed to render the view
    - Populated by the controller
  - View
    - Renders the response to the request
    - Pulls data from the model
    - Defines how to present the data
  - Controller
    - Controllers are responsible for controlling the flow of application execution
    - They control the application logic and act as the coordinator between the View and Model.
    - The also perform application-wide tasks like validation, security and navigation

SPRING WEB MVC

- Spring creates a lightweight container that handles incoming HTTP requests by mapping them to Controllers, which return a Model and View that produces the HTTP response.
- It works around a single Front Controller 'DispatcherServlet' that dispatches requests to handlers, with configurable handler mappings, view resolution, locale and theme resolution as well as support for upload files.
- Individual Controllers can be used to handle many different URLs.
- Controllers are POJOs and are managed exactly like any other bean in the Spring ApplicationContext.

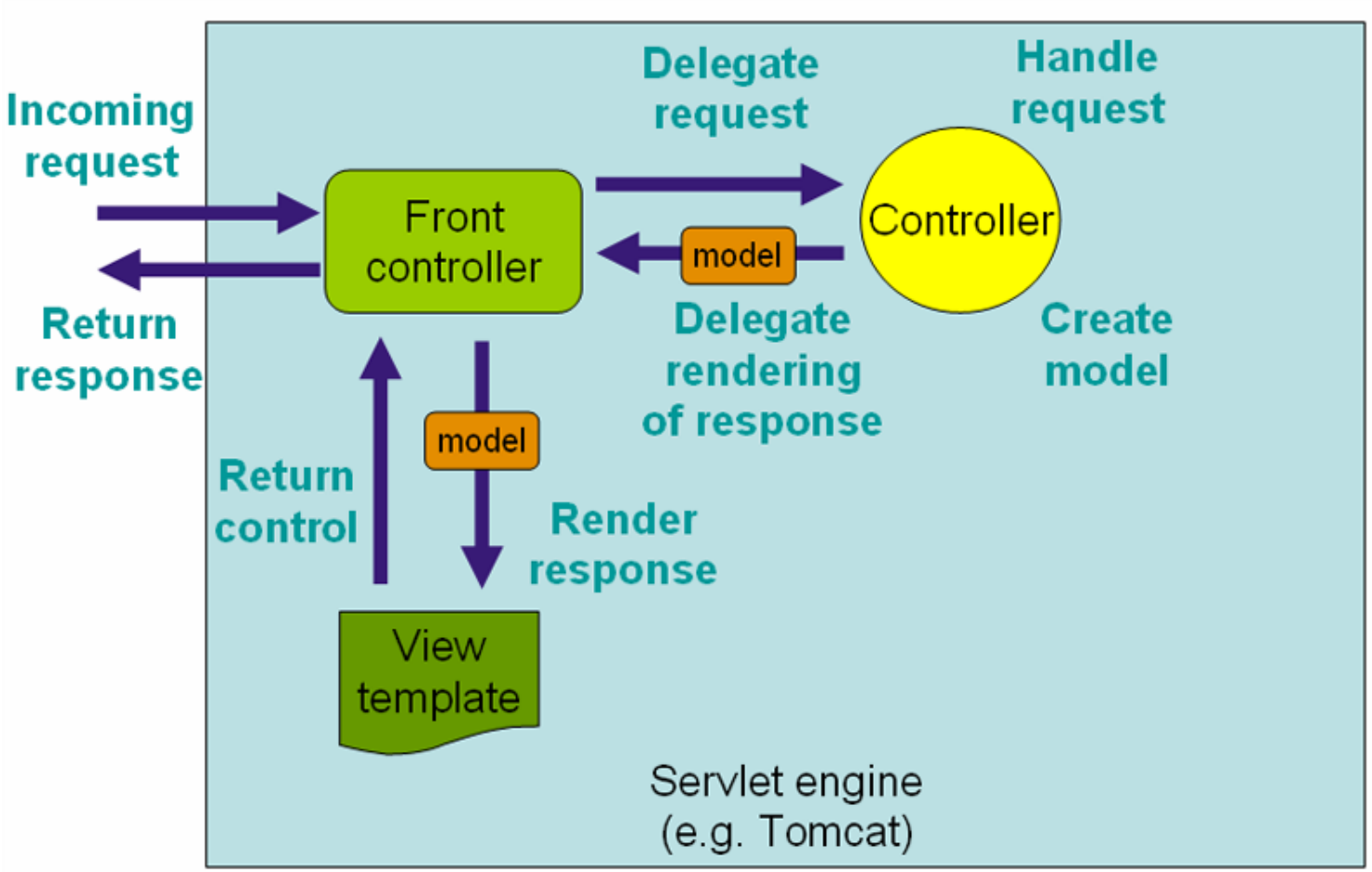
FEATURES OF SPRING WEB MVC

- Clear separation of roles - controller, validator, command object, form object, model object, DispatcherServlet, handler mapping, view resolver, etc. Each role can be fulfilled by a specialized object.
- Powerful and straightforward configuration of both framework and application classes as JavaBeans
- Adaptability, non-intrusiveness, and flexibility. Define any controller method signature you need, possibly using one of the parameter annotations such as @RequestParam, @RequestHeader, @PathVariable, and more.
- Reusable business code - no need for duplication.
- Flexible in supporting different view types like JSP, Velocity, XML, PDF, OGNL etc

FEATURES OF SPRING WEB MVC

- Customizable binding and validation.
- Customizable handler mapping and view resolution.
- Flexible model transfer. Model transfer with a name/value Map supports easy integration with any view technology.
- Customizable locale and theme resolution, support for JSPs with or without Spring tag library, support for JSTL, support for Velocity without the need for extra bridges, and so on.
- A simple yet powerful JSP tag library known as the Spring tag library that provides support for features such as data binding and themes.
- A JSP form tag library, introduced in Spring 2.0, that makes writing forms in JSP pages much easier.

REQUEST PROCESSING WORKFLOW IN SPRING WEB MVC



DISPATCHERSERVLET CONTROLLER

- The DispatcherServlet is the Spring Front Controller
- It Initializes WebApplicationContext
- Uses /WEB-INF/[servlet-name]-servlet.xml by default
- WebApplicationContext is bound into ServletContext
- HandlerMapping - Routing of requests to handlers
- ViewResolver - Maps symbolic name to view
- LocaleResolver - Default uses HTTP accept header, cookie, or session

SPECIAL BEAN TYPES IN THE WEBAPPLICATIONCONTEXT

Bean Type	Explanation
HandlerMapping	Maps incoming requests to handlers and a list of pre- and post-processors (handler interceptors) based on some criteria the details of which vary by HandlerMapping implementation
HandlerAdapter	Helps the DispatcherServlet to invoke a handler mapped to a request regardless of the handler is actually invoked.
HandlerExceptionResolver	Maps exceptions to views also allowing for more complex exception handling code.
ViewResolver	Resolves logical String-based view names to actual View types.
LocaleResolver	Resolves the locale a client is using, in order to be able to offer internationalized views
ThemeResolver	Resolves themes your web application can use, for example, to offer personalized layouts
MultipartResolver	Parses multi-part requests for example to support processing file uploads from HTML forms.

LIFECYCLE OF SPRING MVC APPLICATION

1. Incoming HTTP request is mapped to the Spring DispatcherServlet.
2. The locale resolver is bound to the request to enable elements to resolve the locale to use.
3. The theme resolver is bound to the request to let elements such as views determine which theme to use.
4. If the request is in multiparts and there is a multipart file resolver is specified, the request is wrapped in a MultipartHttpServletRequest for further processing by other elements in the process.
5. An appropriate handler is searched for. If a handler is found, the execution chain associated with the handler (preprocessors, postprocessors, and controllers) is executed in order to prepare a model or rendering.
6. If a model is returned, the view is rendered, If no model is returned, no view is rendered.

CONFIGURE SPRING WEB MVC IN CASE OF XML CONFIGURATION

- Incoming HTTP request is mapped to the Spring DispatcherServlet.
- Edit web.xml as below –

```
<web-app>
        <servlet>
                <servlet-name>first</servlet-name>
                <servlet-lass>
                        org.springframework.web.servlet.DispatcherServlet
                </servlet-class>
                <load-on-startup>1</load-on-startup>
    </servlet>
<servlet-mapping>
                <servlet-name>first</servlet-name>
                <url-pattern>*.spring</url-pattern>
</servlet-mapping>
</web-app>
```

CONFIGURE SPRING WEB MVC

- The DispatcherServlet creates a container using the bean definitions found in the Servlet configuration file.
- The DispatcherServlet locates the configuration file using the naming convention <servletname>-servlet.xml.
- Create WEB-INF\first-servlet.xml as below

```
<mvc:annotation-driven/>
<context:component-scan base-package="com.spring" />
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
</bean>
```

CONFIGURE SPRING MVC USING JAVA-BASED CONFIGURATION

```
<servlet>
        <servlet-name>first</servlet-name>
        <servlet-lass>
                org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <init-param>
         <param-name>contextClass</param-name>
    <param-value>
  org.springframework.web.context.support.AnnotationConfigWebApplicationContext
    </param-value>
  </init-param>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value> spring.config.SpringConfig</param-value>
   </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
<servlet-mapping>
        <servlet-name>first</servlet-name>
        <url-pattern>*.spring</url-pattern>
</servlet-mapping>
```

ENABLING THE MVC JAVA CONFIG OR MVC NAMESPACE

- To enable MVC Java Config
  - @EnableWebMVC
    - Used at @Configuration class

- To achieve the same in XML
  - <mvc:annotation-driven>

IMPLEMENTING CONTROLLERS

- Controllers intercept user input and transform it into a model that is represented to the user by the view.
- Since 2.5, Controllers can be implemented using Annotations.
- Two important annotations which are used are:
  - @Controller
  - @RequestMapping
- Controllers implemented in this style do not have to extend specific base class or implement specific interfaces.

IMPLEMENTING CONTROLLER

```
@Controller
public class HelloWorldController {

    @RequestMapping(value="/sayHello")
    public String myHelloMethod(Model model){
        model.addAttribute("message", "Hello World");
        return "helloworld";
    }

}
```

- @Controller annotation indicates that a particular class serves the role of a controller
- The dispatcher scans such annotated classes for mapped methods and detects @RequestMapping annotation.
- The returned value need to be mapped as a JSP page using the View Resolver configuration.

CONFIGURING VIEW RESOLVERS

In xml file

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

In JavaConfig class

```
@Bean
public InternalResourceViewResolver getResolver(){
    InternalResourceViewResolver resolver = new InternalResourceViewResolver();
    //resolver.setPrefix("/WEB-INF/jsp/");
    resolver.setSuffix(".jsp");
    return resolver;
}
```

CREATE VIEW PAGE

- Create helloworld.jsp as below:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>

Message is ${message}

</body>
</html>
```

MAPPING REQUEST WITH @REQUESTMAPPING

- @RequestMapping annotation can be used to map URLs onto an entire class or a particular method.
  - Class-level annotation maps a specific request path on to the controller
  - Additional method-level annotations narrow down the primary mapping for a specific HTTP request method (GET, POST etc) or an HTTP request parameter condition

```
@Controller
@RequestMapping("/order")
Public class OrderController{

@RequestMapping(method=RequestMethod.GET)
Public List<Products> getProducts(){}

@RequestMapping(path="/place", method=RequestMethod.POST)
public String placeOrder(){}
```

Request through Get method for URL /order

Request through Post method for URL /order/place

URI TEMPLATE PATTERNS

- URI Template is a URI-like string, containing one or more variable names. When the values are substituted for these variables, the template becomes a URI.
- For ex,

  http://www.example.com/users/{userId} – URI template containing variable *userId*

Providing value to the variable, it becomes a URI

  http://www.example.com/users/Neelam

- To bind the value of a URI template variable, @PathVariable annotation is used on a method argument

```
@RequestMapping("/orders/{orderId}")
public String getOrder(@PathVariable String orderId, Model model){
        Order order = OrderService.findOrder(orderId)
        model.addAttribute("order",order);
        return displayOrder;
}
```

Use of PathVariable Annotation

URI TEMPLATE PATTERNS

- A method can have any number of @PathVariable annotations

```
@RequestMapping("customer/{customeId}/orders/{orderId}")
public String getOrder(@PathVariable String customerId, @PathVariable String
orderId, Model model){
}
```

- A URI template can be assembled from type and method level @RequestMapping annotations

```
@Controller
@RequestMapping("/customer/{customerId}")
Public class CustomerController{

        @RequestMapping("/orders/{orderId}")
        public String getOrder(@PathVariable String customerId, @PathVariable String
orderId, Model model){ }
}
```

URI TEMPLATE PATTERNS

- @RequestMapping annotation supports the use of regular expression

```
@RequestMapping("/spring-web/{symbolicName:[a-z-]+}-{version:\\d\\.\\d\\.\\d}{extension:\\.[a-z]+}" )
public String getOrder(@PathVariable String version, @PathVariable String extension){
}
```

REQUEST PARAMS AND HEADER VALUES

- Request Matching can also be narrowed down through request parameter conditions such as "myParams", "!myParams", or "myParam=myValue"
  - The first two test for request parameter presence/absence and the third for a specific parameter value
  - Example

```
@RequestMapping(path="/customer", params="myParam=myValue")
public String getCustomer(Model model){
}
```

- The same can be done to test for request header presence/absence to match based on a specific request header value

```
@RequestMapping(path="/customer", headers="myHeader=myValue")
public String getCustomer(Model model){
}
```

DEFINING @REQUESTMAPPING HANDLER METHODS

- @RequestMapping handler methods can have very flexible signatures.
- It supports a long list of method arguments and return types, Commonly used arguments and return types are described in next slides
- Most arguments can be used in arbitrary order with the only exception being **BindingResult** arguments (described later)

SUPPORTED METHOD ARGUMENT TYPES

Type	Description
Request or Response objects	Servlet API
Session Object	An argument of this type enforces the presence of a corresponding session
@PathVariable	annotated parameters for access to URI template variables
@RequestParam	annotated parameters for access to specific Servlet request parameters
@RequestHeader	annotated parameters for access to specific Servlet request HTTP headers
@RequestBody	annotated parameters for access to the HTTP request body
@RequestPart	annotated parameters for access to the content of a "multipart/form-data" request part
@SessionAttribute	annotated parameters for access to existing, permanent session attributes

SUPPORTED METHOD ARGUMENT TYPES

Type	Description
@RequestAttribute	annotated parameters for access to request attributes.
ModelMap/Model	for enriching the implicit model that is exposed to the web view.
@ModelAttribute	temporarily stored in the session as part of a controller workflow
BindingResult	validation results for a preceding command or form object

SUPPORTED METHOD RETURN TYPES

Type	Description
ModelAndView object	ModelAndView can store the Model data and view name to be displayed
Model object	Model object with view name implicitly determined
String	Logical view name
Void	If the method handle the response itself
View	A view object
HttpHeaders	To return a response with no body

DATA VALIDATION

NEELAM AGARWAL

DATA VALIDATIONS

- Validation in Spring can be done in either of two ways
  - Using Validator Interface provided by Spring
  - Using Annotations

VALIDATOR INTERFACE

- 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.
- Data binding is useful for allowing user input to be dynamically bound to the domain model of an application

SPRING VALIDATION API

- org.springframework.validation.Validator(Interface) - Validator for application-specific objects.
  - void validate(Object target,Errors errors)

- org.springframework.validation.ValidationUtils(class) - Utility class convenient methods for invoking a Validator and for rejecting empty fields.
  - static void rejectIfEmpty(Errors errors, String field, String errorCode)
  - static void rejectIfEmptyOrWhitespace(Errors errors, String field, String errorCode)

- org.springframework.validationErrors(Interface) - Stores and exposes information about data-binding and validation errors for a specific object.

VALIDATIONS USING JSR303 (HIBERNATE)

- JSR-303 standardizes validation constraint declaration and metadata for the Java platform.
- This API is annotate domain model properties with declarative validation constraints and the runtime enforces them.
- There are a number of built-in constraints you can can take advantage of.
- You may also define your own custom constraints.

VALIDATIONS USING JSR303 (HIBERNATE)

Create bean class as below –

```
public class RegistrationBean {
@NotEmpty(message="Name field is mandatory.")
private String name = null;

@NotEmpty(message="Username field is mandatory.")
private String username = null;

@NotEmpty(message="Email field is mandatory.")
private String email = null;

@Length(max=10,min=10,message="Phone number is not valid. Should be of length 10.")
@NotEmpty(message="Phone field is mandatory.") @NumberFormat(style= Style.NUMBER)
private String phone;

@NotEmpty(message="Password should not be empty.")
private String password = null;
// all getter/setter
}
```

FILE UPLOAD

NEELAM AGARWAL

MULTIPART(FILEUPLOAD) SUPPORT

- Spring's built-in multipart support handles file uploads in web applications
- This support is enabled with MultipartResolver objects.
- One MultipartResolver implementation is for use with Commons FileUpload

USING A MULTIPART RESOLVER WITH COMMONS FILEUPLOAD

```
@Bean
public CommonsMultipartResolver multipartResolver(){
    CommonsMultipartResolver resolver = new CommonsMultipartResolver();
    //the maximum file size in bytes
    resolver.setMaxUploadSize(2000);
    return resolver;
}
```

- The jar file required for the same is commons-fileupload.jar
- When DispatcherServlet detects a multi-part request, it activates the resolver and hands over the request
- The resolver then wraps the current HttpServletRequest into a MultipartHttpServletRequest that supports multiple file uploads.

EXCEPTION HANDLING IN SPRING MVC

NEELAM AGARWAL

HANDLING EXCEPTIONS

- Implement interface – HandlerExceptionResolver
  - Allows to map Exceptions to specific views declaratively along with some optional Java logic code before forwarding to these views
- Use already provided implementation – SimpleMappingExceptionResolver
  - Enables to take the class name of any exception that might be thrown and map it to a view name.
- Create @ExceptionHandler methods
  - Can be used on methods that should be invoked to handle an exception
  - These methods may be defined locally within an @Controller

HANDLEREXCEPTIONRESOLVER

- Any Spring Bean that implements HandlerExceptionResolver will be used to intercept and process any exception raised in the MVC system and not handler by a Controller.

```
public interface HandlerExceptionResolver {
    ModelAndView resolveException(HttpServletRequest request,
            HttpServletResponse response, Object handler, Exception ex);
}
```

- The handler refers to the controller that generated the exception

- The interface can be implemented to set up our own custom exception handling system.

SIMPLEMAPPINGEXCEPTIONRESOLVER

- Convenient implementation of HandlerExceptionResolver. It provide options to:
  - Map exception class names to view names
  - Specify a default error page for any exception not handled anywhere else
  - Log a message (not enabled by default)
  - Set the name of the exception attribute to add to the Model so that it can be used inside a View. Default name is ‘exception’

CONFIGURING SIMPLEMAPPINGEXCEPTIONRESOLVER

```
@Bean
public SimpleMappingExceptionResolver exceptionResolver(){
    SimpleMappingExceptionResolver sp = new SimpleMappingExceptionResolver();
    Properties mappings = new Properties();
    mappings.put("java.io.IOException", "error");
    mappings.put("spring.exception.MyException", "error1");
    sp.setExceptionMappings(mappings);  //none by default
    sp.setDefaultErrorView("error");    //No default
    sp.setExceptionAttribute("ex");     //Default is "exception"
    sp.setWarnLogCategory("spring.MvcLogger");  //No Default
    return sp;
}
```

The *defaultErrorView* property is especially useful as it ensures any uncaught exception generates a suitable application defined error page.

@EXCEPTIONHANDLER

- @ExceptionHandler methods can be added to the controllers to specifically handle exceptions thrown by request handling methods in the same controller

- Such methods can
  - Handle exceptions without @ResponseStatus annotation
  - Redirect the user to a dedicated error view
  - Build a totally custom error response

@EXCEPTIONHANDLER

```java
// Convert a predefined exception to an HTTP Status code
@ResponseStatus(value=HttpStatus.CONFLICT,reason="Data integrity violation")
@ExceptionHandler(DataIntegrityViolationException.class)
public void conflict() {
  // Nothing to do
}

// Specify name of a specific view that will be used to display the error:
@ExceptionHandler({SQLException.class,DataAccessException.class})
public String databaseError() {
  // Nothing to do.  Returns the logical view name of an error page, passed
  // to the view-resolver(s) in usual way.
  // Note that the exception is NOT available to this view
  return "databaseError";
}

// Total control - setup a model and return the view name yourself.
@ExceptionHandler(Exception.class)
public ModelAndView handleError(HttpServletRequest req, Exception ex) {
  System.err.println("Request: " + req.getRequestURL() + " raised " + ex);
  ModelAndView mav = new ModelAndView();
  mav.addObject("exception", ex);
  mav.addObject("url", req.getRequestURL());
  mav.setViewName("error");
  return mav;
}
```