Spring MVC

Agenda

- Spring MVC overview
- Creating controllers
- Rendering frontend
- Spring Boot MVC

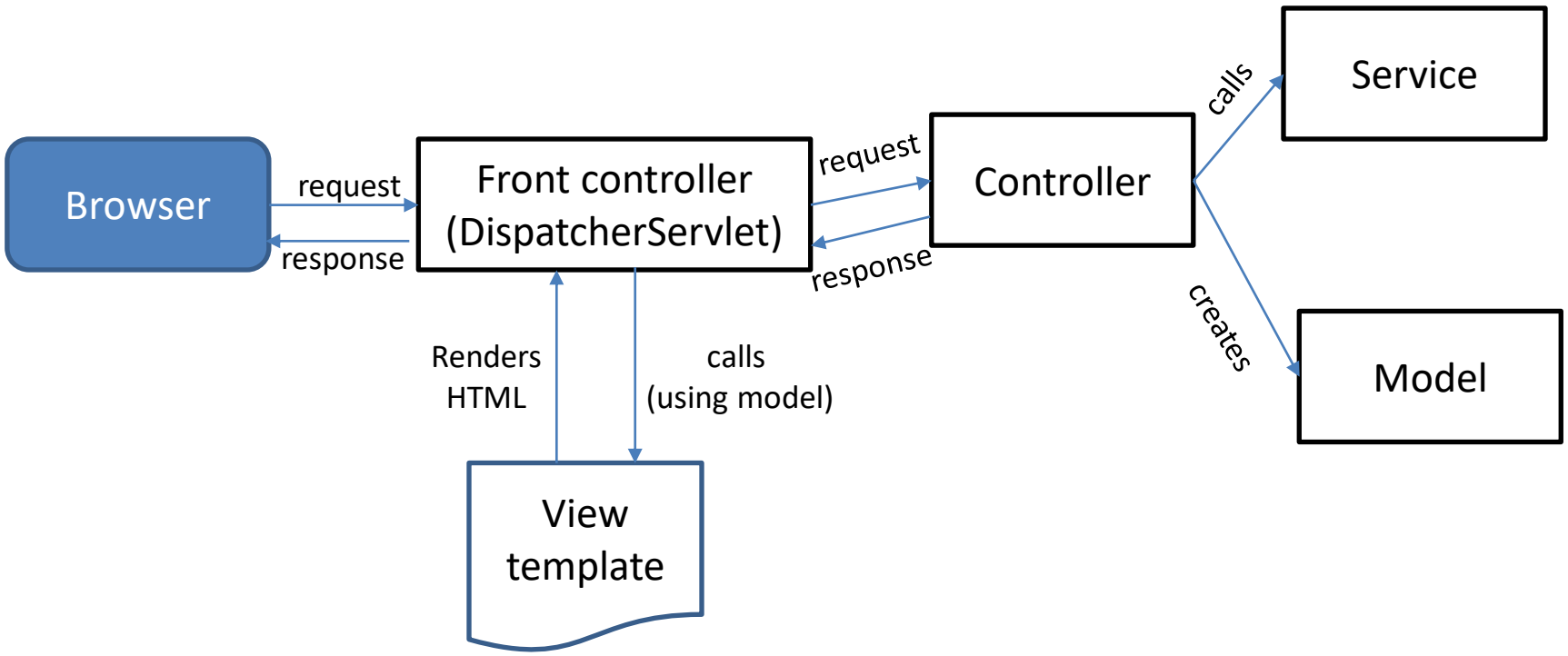
Spring MVC overview

Spring MVC

- Spring MVC is web development framework built on top of the Spring core DI framework

- MVC stands for “model-view-controller” and is an architectural pattern implemented by a number of so called web “MVC” frameworks

Spring MVC architecture



Spring MVC dependencies

```
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-web</artifactId>
	<version>${spring.version}</version>
</dependency>
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-webmvc</artifactId>
	<version>${spring.version}</version>
</dependency>
```

Spring MVC configuration

- To configure Spring MVC in a Spring application it is sufficient to add the @EnableWebMvc annotation on a @Configuration class

```
@Configuration
@EnableWebMvc
public class AppConfig {
        …
}
```

Spring MVC controllers

- A Spring MVC controller is a Spring bean annotated with the @Controller or @RestController annotations

```
@Controller
public class AppController {

	@RequestMapping(value = "/home")
	public String home() {
		return "index.jsp";
	}
}
```

Creating controllers

Mapping annotations

- Mapping of a controller method to an HTTP endpoint can be done with the @RequestMapping annotations
- It can be applied on the controller class or a controller method (or both)
- Convenience annotations can be used on the method level such as @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, and @PatchMapping

Method parameters

- Controller method parameters can be bound to different parts of the HTTP request:
  - @RequestParam: for mapping an argument to an HTTP request parameter
  - @RequestHeader: for mapping an argument to an HTTP request header
  - @RequestBody: for mapping an argument to the HTTP request body
  - @CookieValue: for mapping an argument to an HTTP cookie value
  - @PathParam: for mapping an argument to a URL path value
  - HttpEntity: for access to request headers and body

Handling request parameters

```
@Controller
public class AppController {

	@GetMapping(value = "/home")
	public String home(@RequestParam String page)  {
		…
		return "index.jsp";
	}
}
```

e.g. localhost:8080/home?page=3

Handling path parameters

```
@Controller
public class AppController {

	@GetMapping(value = "/home/{page}")
	public String home(@PathParam String page)	{
		…
		return "index.jsp";
	}
}
```

e.g. localhost:8080/home/3

Handling headers

```
@Controller
public class AppController {

	@GetMapping(value = "/home/{page}")
	public String home(@PathParam String page,
	@RequestHeader(“Accept-Encoding") String encoding ) {
		…
		return "index.jsp";
	}
}
```

Handling cookies

```
@Controller
public class AppController {

	@GetMapping(value = "/home/{page}")
	public String home(@PathParam String page,
	@CookieValue("language") String lang) {
		…
		return "index.jsp";
	}
}
```

Handling HTTP body

```
@Controller
public class AppController {

	@GetMapping(value = "/home/{page}")
	public String home(@PathParam String page,
	@RequestBody Entity entity) {
		…
		return "index.jsp";
	}
}
```

Handling exceptions

- Exceptions are handled in Spring MVC by means of HandlerExceptionResolver beans
- A few different implementations are provided by Spring framework such as:
  - SimpleMappingExceptionResolver
  - DefaultHandlerExceptionResolver
  - ResponseStatusExceptionResolver
  - ExceptionHandlerExceptionResolver

Handling exceptions

- The most common strategy used by application is provided by the ExceptionHandlerExceptionResolver:
  - local: @Controller classes have one or more methods annotated with @ExceptionHandler
  - global (apply for all controllers): a class annotated with @ControllerAdvice with one or more @ExceptionHandler methods

Validation

- Bean validation supported by Spring DI framework can be used in the same manner in Spring MVC applications
- Entity classes are either marked with @Validated or controller method parameters with @Valid
- Bean validation annotations are used

Interceptors

- Interceptors can be registered in order to pre-process incoming requests

```
@Configuration
@EnableWebMvc
public class AppConfig implements WebMvcConfigurer
{
    @Override
    public void addInterceptors(InterceptorRegistry
registry) {
        registry.addInterceptor(…);
        registry.addInterceptor(…);
}
```

Rendering response

- The response can be rendered as follows:
  - Using the @ResponseBody annotation and a defined HttpMessageConverter bean used to convert the response to a proper format
  - A String that indicates the view template to render

Rendering frontend

Rendering frontend

- The result of controller methods can be interpreted by Spring MVC framework in any of the following ways:
  - by marshalling a Java bean in a proper representation (XML, JSON) or returning a ResponseEntity instance (as already discussed)
  - by rendering a proper frontend page that is returned using a proper ViewResolver for the target template being used

Configuring a view resolver

- In order to render a template a view resolver needs to be explicitly configured

- For example to configure a Thymeleaf view resolver:

```
@Controller
public class WebMvcConfig {

@Bean
    public ThymeleafViewResolver viewResolver() {
        final ThymeleafViewResolver resolver =
              new ThymeleafViewResolver();
         …
        return resolver;
    }
}
```

Configuring a view resolver

```
@Bean
public ThymeleafViewResolver viewResolver() {

        ThymeleafViewResolver viewResolver =
                new ThymeleafViewResolver();
        SpringResourceTemplateResolver templateResolver =
                new SpringResourceTemplateResolver();

        templateResolver.setPrefix("classpath:/views");
        templateResolver.setSuffix(".html");

        SpringTemplateEngine templateEngine =
                new SpringTemplateEngine();
        templateEngine.setTemplateResolver(
                templateResolver);
        viewResolver.setTemplateEngine(templateEngine);
        return viewResolver;
}
```

Template engines

- Spring MVC supports the following templating engines (among others) for rendering of frontend:
  - Thymeleaf
  - Velocity
  - Freemarker
  - Mustache

Passing the model

- The model can be modified before passed to the view as follows:
  - passing a Model argument to the controller method
  - passing a ModelMap argument to the controller method
  - passing a ModelAndView argument to the controller method and returning it

Spring Boot MVC

Spring Boot MVC

- Spring simplifies configuration of Spring MVC applications

- For example:

  - adding **spring-boot-starter-thymeleaf** as a dependency to the builds automatically configures Thymeleaf as a template engine

  - Adding **spring-boot-starter-json** as a dependency to the build automatically configures Jackson for marshaling/unmarshalling of Spring MVC data to/from JSON

Spring Boot MVC

- Spring Boot configuration can be provided by creating proper beans (provided by Spring Boot classes) in a @Configuration class with using @EnableWebMvc annotation

```
@Configuration
public class MvcConfiguration {

    @Bean
    public HttpMessageConverters customConverters() {
        HttpMessageConverter<?> first = ...
        HttpMessageConverter<?> second = ...
        return new HttpMessageConverters(first, second);
    }
}
```

Default converters for JSON and XML and provided by Spring Boot

Static content

- Spring Boot by default serves static content from the following directories on the classpath:
  - /static
  - /public
  - /resources
  - /META-INF/resources)

Static resources can be controlled with the spring.mvc.static-path-pattern property: i.e. spring.mvc.static-path-pattern=/resources/*.jpg

View templates

- Spring Boot serves view templates from the **src/main/resources/templates** directory

- Provides autoconfiguration for the following template engines:

  - Thymeleaf
  - FreeMaker
  - Groovy
  - Mustache

Error handling

- By default Spring Boot provides an **/error** mapping that handles application errors
- Error handling response can be customized
- Error pages can also be placed in the /error directory and must be named with HTTP error code

```
src/main/resources/public/error/404.html
```

Questions ?