Class 11: Intro to Spring

Resources

- Baeldung Spring Request Mappings

Learning Objectives

- Build a "Hello World" server with Spring
- Serve static content with Spring
- Receive information via URL queryparameters
- Use `@Application`, `@Controller`, `@QueryParameter` annotations

Spring Gradle Dependencies

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:2.0.3.RELEASE")
    }
}

apply plugin: 'java'
apply plugin: 'idea'
apply plugin: 'org.springframework.boot'
apply plugin: 'io.spring.dependency-management'

repositories {
    mavenCentral()
}

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    compile("org.springframework.boot:spring-boot-starter-thymeleaf")
    compile("org.springframework.boot:spring-boot-devtools")
    testCompile("junit:junit")
}
```

Project Structure

- Spring wants applications to appear in a certain structure.
- Create a package under `src/main/java` and add package statements to your `.java` files.

```
MyProject
└── src
    ├── main
    |   ├── java
    |   |   └── server
    |   |       ├── Application.java
    |   |       └── SearchController.java
    |   └── resources
    |       └── index.html
    └── test
        ├── java
        └── resources
```

Project Structure

If you don't follow this structure and add package statements you may see this error:

```
** WARNING ** : Your ApplicationContext is unlikely to start
               due to a `@ComponentScan` of the default package.
```

Notice that `Application.java` file on the next slide has a `package server;` statement at the very beginning of the file.

Application.java

This is the base file where your server starts. It uses the `SpringBootApplication` annotation to tell Spring this is where the server starts.

```
package server;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class);
    }
}
```

Serving Static Content

Spring Boot will automatically add static web resources located within any of the following directories:

- `/resources/`
- `/static/`
- `/public/`

Static content is things like images, HTML, CSS and JavaScript files. It's called static content because it never changes. Static files aren't pre-processed before they're served.

Controllers

We have to use the `@ResponseBody` annotation to tell Spring this method returns content for the body of the HTTP response.

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

@Controller
public class GreetingController {
    @GetMapping("/greeting")
    @ResponseBody
    public String greeting(
        @RequestParam(name="name", required=false, defaultValue="World") String na
    ) {
      return "Hello " + name + "!";
    }
}
```

Request Mappings

Spring has annotations for methods to mark that they handle requests.

- `@GetMapping` - the method only matches GET requests
- `@PostMapping` - the method only matches POST requests
- `@RequestMapping` - the method will respond for any request, no matter the HTTP method

```
@RequestMapping(
  value = "/ex/foos",
  method = RequestMethod.GET,
  produces = "application/json"
)
```

@Request Mappings

Request mappings can be applied to top-level controller files, and to methods within a controller.

This controller has a search method accessible at `/api/search`

```
@Controller
@RequestMapping("/api")
public class SearchController {
    @RequestMapping(value="/search")
    @ResponseBody
    public String search(
      @RequestParam(name="query", defaultValue="The Beatles") String query
    ) {
        return "Searching for " + query;
    }
}
```

@RequestParam

The `@RequestParam` annotation pells information out of requests and maps them to parameters on a method.

- `name` the name of the parameter in the querystring, like `?name=value`.
- `required` the application will throw an error if a request arrives without the parameter
- `defaultValue` allows you to provide a default value

Yes, the `@RequestPAram` makes methd signatures VERY long. Split the parameters across different lines for readability.

```
@RequestParam(name="name", required=false, defaultValue="World") String name,
```

Request Path Variables

```
@RequestMapping(value = "/ex/foos/{id}", method = GET)
@ResponseBody
public String getFoosBySimplePathWithPathVariable(
    @PathVariable("id") long id) {
  return "Get a specific Foo with id=" + id;
}
```

Mapping for fallback

Use `*` for the request mapping value to have it respond to any possible request. This is a good way to make one route that handles every request that didn't match anything else, like a displaying a custom 404 page.

```
@RequestMapping(value = "*")
@ResponseBody
public String getFallback() {
    return "Fallback for GET Requests";
}
```

Templates

- Spring uses a template engine called Thymeleaf by default.
- Spring looks for template pages under `/resources/templates`
- Add the XML namespace to the top-level `HTML` tag
- Use attributes like `th:text` and `th:src` on `<p>` and `<img>` tags

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Getting Started: Serving Web Content</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    </head>
    <body>
        <h1>Search Results</h1>
        <p th:text="'Searching for: ' + ${query}" />
        <div>
          <img th:src="${src}" />
        </div>
    </body>
</html>
```

Templates and Controllers

- Spring assumes that Controller methods with a `String` return type are trying to return the name of a template.
- We must use the `ResponseBody` annotation to specifically tell Spring methods are returning content and not a template name.
- Accept a `Model model` parameter and attach properties to it to make values accessible in the template.

```
@RequestMapping(value="/ui")
public String searchTemplate(
  @RequestParam(name="query", defaultValue="The Beatles") String query,
  Model model
) {
    String src = AlbumScraper.getAlbumArtUrl(query);
    model.addAttribute("query", query);
    model.addAttribute("src", src);
    return "result";
}
```

Deploying to Heroku

- Spring is actually very easy to deploy to Heroku.
- Heroku detects Spring appliations automatically.
- Follow their instructions

Make sure to download the Heroku CLI from their instructions, then:

```
heroku login
heroku create
git push heroku master
heroku open
```