SEG3102 – Lab 2

Spring

Introduction Spring MVC

The objective of this Lab is an introduction to the Spring framework with Springboot. We will cover the installation of the necessary tools and create a Web Application with Spring MVC.

The source code for the lab is available in the Github repository https://github.com/stephanesome/springBootIntro.

Spring Setup

Spring is a framework that runs on Java Virtual Machine (JVM). You therefore need to have a Java Development Kit installed.

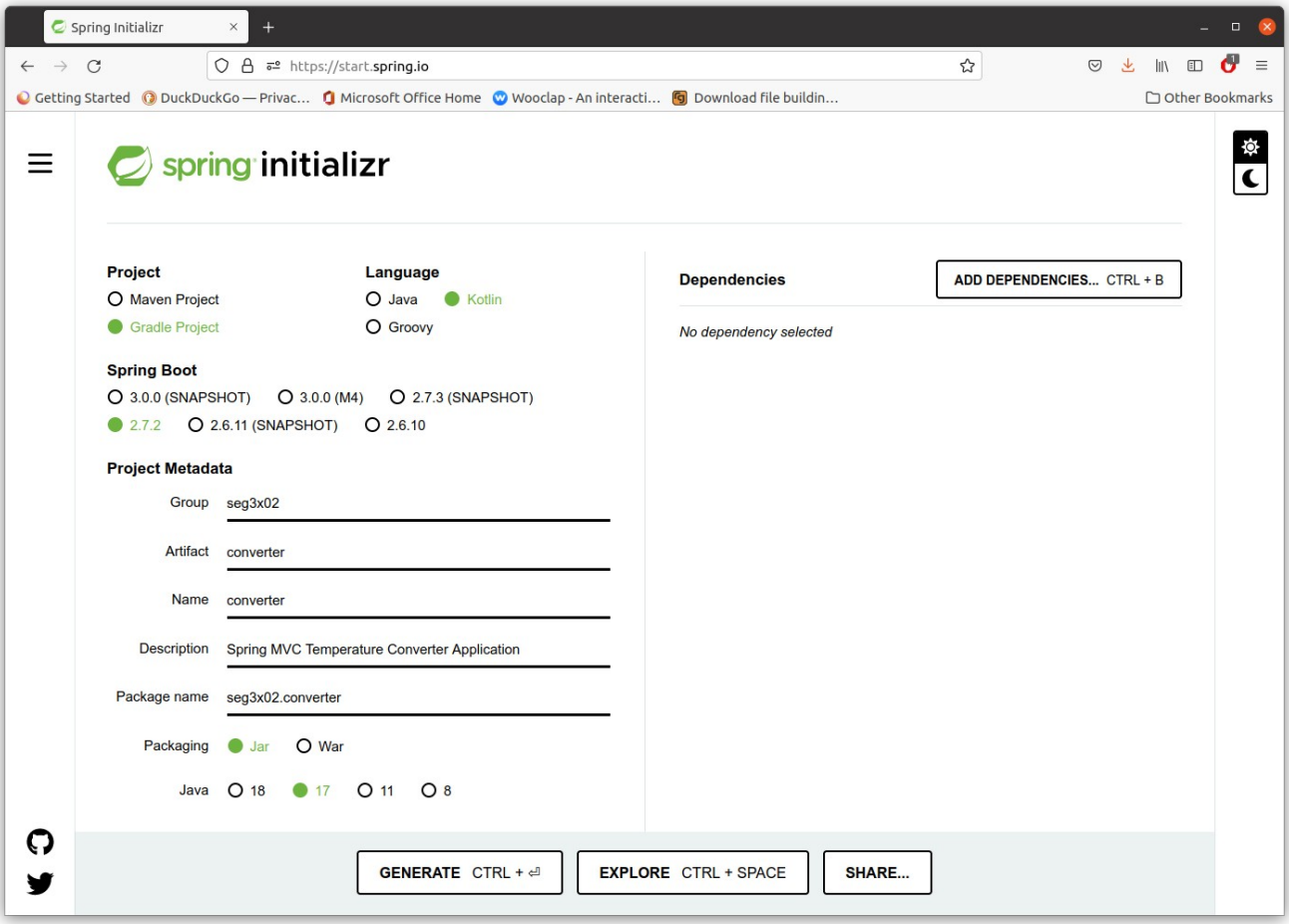
An Integrated Developer Environment (IDE) is strongly recommended. Choices include IntelliJ IDEA, Spring Tools, Visual Studio Code, or Eclipse, and many more. I suggest IntelliJ IDEA for which you can get a student licence.

We will use Kotlin as development language. Kotlin is a language that compile to the JVM and is compatible with Java. It however adds lots of improvement to Java including removing the extensive boilerplate code that is characteristic of Java. Refer here for documentation about Kotlin.

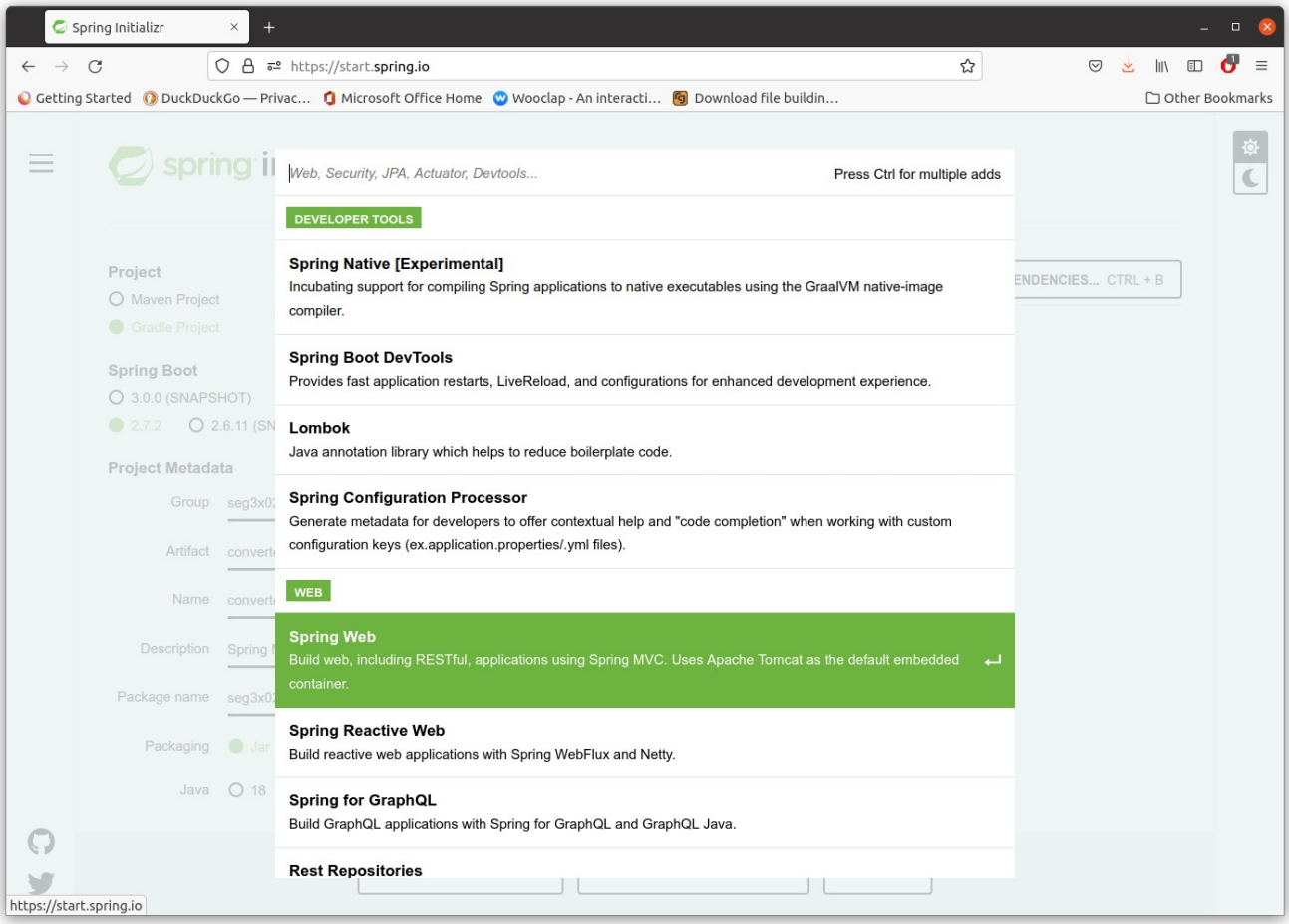
For proper editing of Kotlin code, you need to add the appropriate plugin for Kotlin to your IDE (e.g. for IntelliJ IDEA, Spring Too Suite, VS Code).

Generate Application

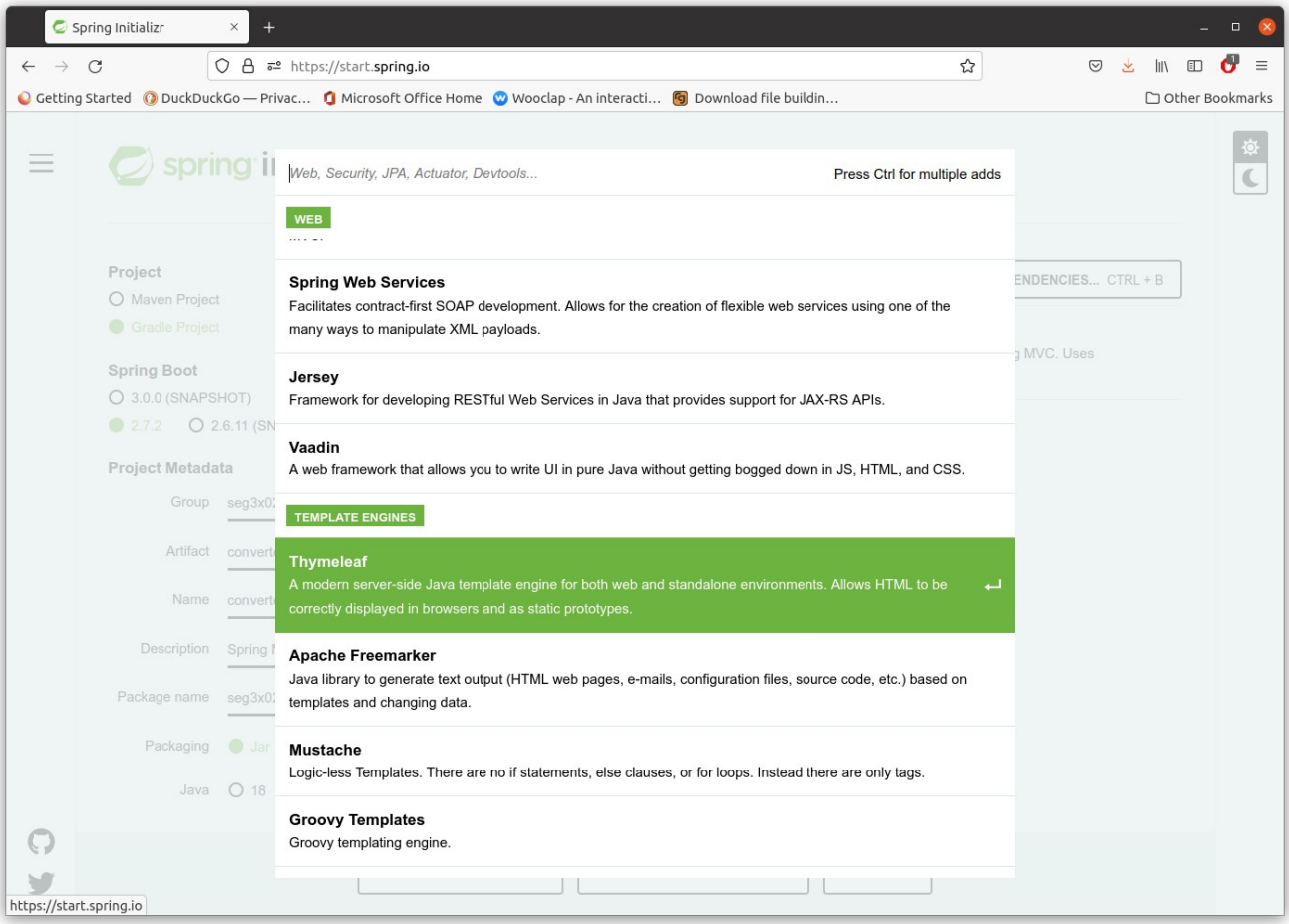
Spring provides a Web Service called Spring Intializr for the setting up of applications. You can access the service at https://start.spring.io/.



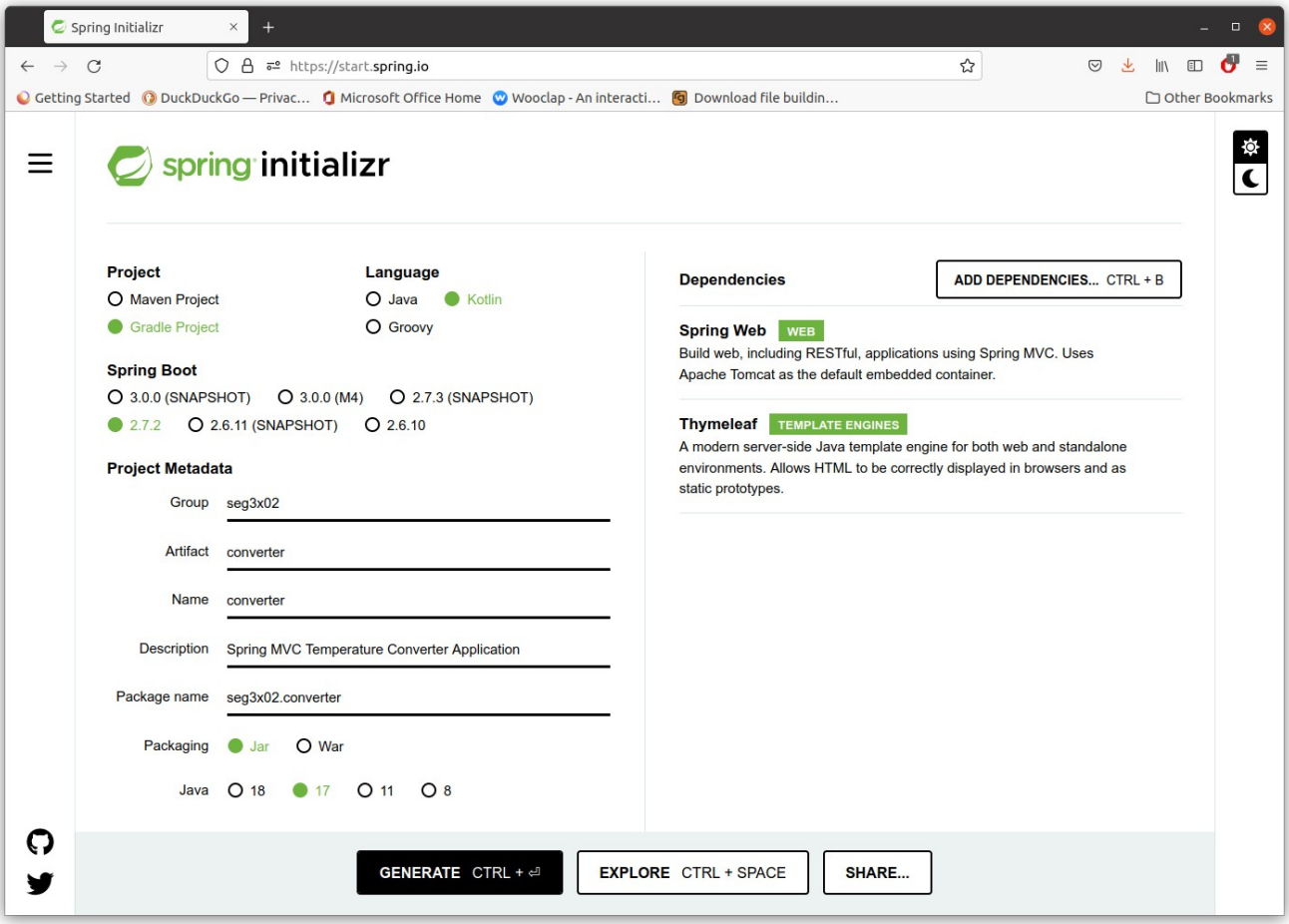
- Select Gradle Project as Project type and Kotlin as Language
- Leave the selected Spring Boot version to the default
- Enter the Project Metadata
- Make sure you have the right version of JDK installed
- In the Dependencies section, click Add Dependencies. Select Spring Web



- Click Add Dependencies again and Select Thymeleaf.

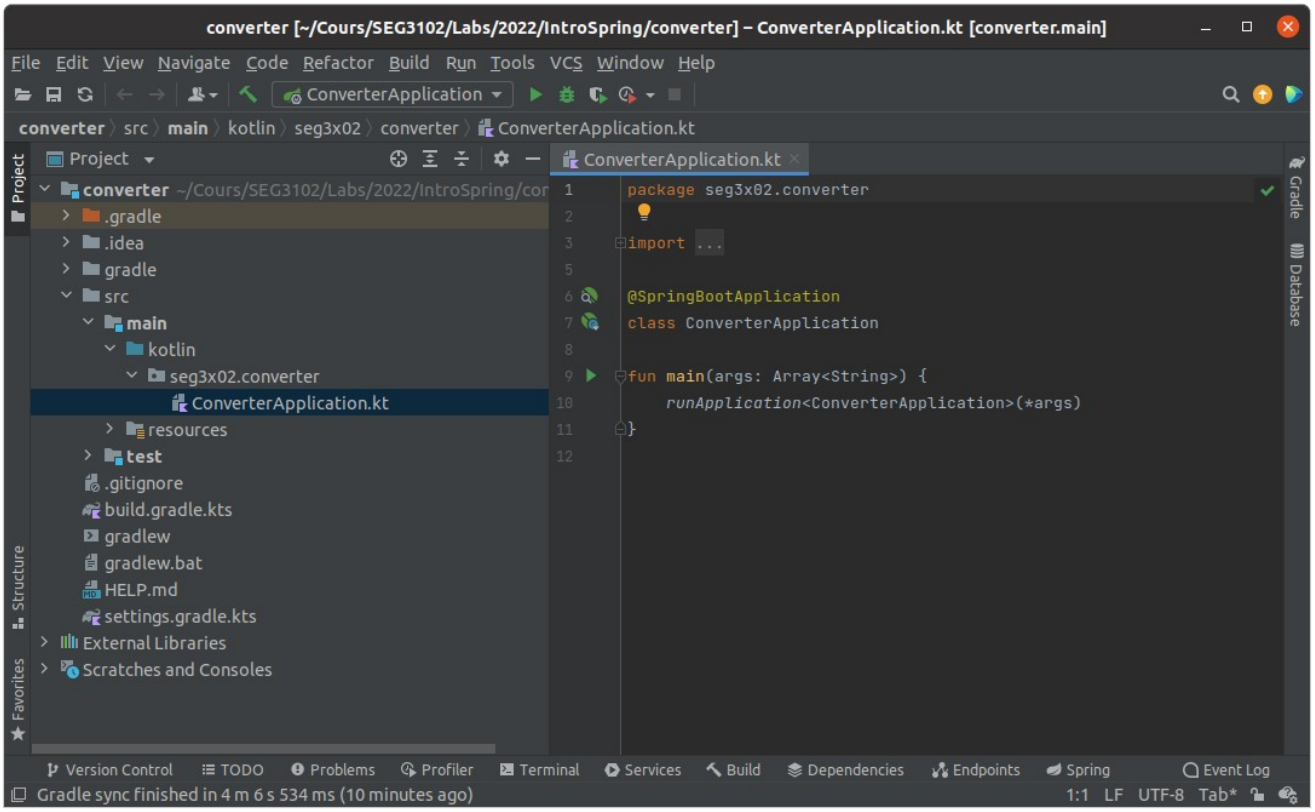


- Generate the project and download the generated zipped file.



Development of Application

Unzip the downloaded zip file and import the project in your IDE.



The generated application includes all the necessary set-up including a Gradle configuration file (build.gradle.kts), executable scripts for Gradle (gradlew, gradlew.bat). Additionally, the application includes a main file (ConverterApplication.kt) with a class annotated @SpringBootApplication. This marks the class as a configuration class and enables auto-configuration and component scanning. The main file also includes the main function that is executed to launch the application.

Controller

In the main source folder, create a Kotlin source file in package seg3x02.converter called WebController. Edit as follow.

```
1.  package seg3x02.converter
2.
3.  import org.springframework.stereotype.Controller
4.  import org.springframework.ui.Model
5.  import org.springframework.web.bind.annotation.GetMapping
6.  import org.springframework.web.bind.annotation.ModelAttribute
7.  import org.springframework.web.bind.annotation.RequestMapping
8.  import org.springframework.web.bind.annotation.RequestParam
9.
10. @Controller
11. class WebController {
12.     @ModelAttribute
```

```
13.     fun addAttributes(model: Model) {
14.         model.addAttribute("error", "")
15.         model.addAttribute("celsius", "")
16.         model.addAttribute("fahrenheit", "")
17.     }
18.
19.     @RequestMapping("/")
20.     fun home(): String {
21.         return "home"
22.     }
23.
24.     @GetMapping(value = ["/convert"])
25.     fun doConvert(
26.         @RequestParam(value = "celsius", required = false) celsius: String,
27.         @RequestParam(value = "fahrenheit", required = false) fahrenheit: String,
28.         @RequestParam(value = "operation", required = false) operation: String,
29.         model: Model
30.     ): String {
31.         var celsiusVal: Double
32.         var fahrenheitVal: Double
33.         when (operation) {
34.             "CtoF" ->
35.                 try {
36.                     celsiusVal = celsius.toDouble()
37.                     fahrenheitVal = ((celsiusVal * 9) / 5 + 32)
38.                     model.addAttribute("celsius", celsius)
39.                     model.addAttribute("fahrenheit", String.format("%.2f", fahrenheitVal))
40.                 } catch (exp: NumberFormatException) {
41.                     model.addAttribute("error", "CelsiusFormatError")
42.                     model.addAttribute("celsius", celsius)
43.                     model.addAttribute("fahrenheit", fahrenheit)
44.                 }
45.             "FtoC" ->
46.                 try {
47.                     fahrenheitVal = fahrenheit.toDouble()
48.                     celsiusVal = ((fahrenheitVal - 32) * 5) / 9
49.                     model.addAttribute("celsius", String.format("%.2f", celsiusVal))
50.                     model.addAttribute("fahrenheit", fahrenheit)
51.                 } catch (exp: NumberFormatException) {
52.                     model.addAttribute("error", "FahrenheitFormatError")
53.                     model.addAttribute("celsius", celsius)
54.                     model.addAttribute("fahrenheit", fahrenheit)
55.                 }
56.             else -> {
57.                 model.addAttribute("error", "OperationFormatError")
58.                 model.addAttribute("celsius", celsius)
59.                 model.addAttribute("fahrenheit", fahrenheit)
60.             }
61.         }
```

```
62.         return "home"
63.     }
64. }
```

Annotation @Controller (line 10) specifies the class as a MVC Controller.

We use a @ModelAttribute annotated function on lines 12-17 to initialize model attributes. Function addAttributes will be executed prior to the invocation of any handler function in the class to set or reset the model attributes.

Function home on lines 19-22, is a handler for the root path "/" as specified by the annotation @RequestMapping. It handles all requests to the URL constructed by prefixing the path with the server deployment URL and the application port. For a local deployment, this would be by default `http://localhost:8080/`. The function returns the logical view "*home*" that is mapped to template src/resources/templates/home.html.

Function doConvert on lines 24-63, handles HHTP GET requests to "*/convert*". Requests can pass three parameters (*celsius*, *fahrenheit*, *operation*) that are mapped to the function arguments with annotation @RequestParam. These parameters are used to determine the type of conversion and the value to convert. Celsius to Fahrenheit conversion is performed on lines 34-44 and Fahrenheit to Celsius conversion on lines 45-55. The conversion result is stored by replacing the value of the model attribute (lines 38 and 49). Model attribute "*error*" is used to report error situations on lines 41, 52 and 57. The function returns the logical view "*home*" in which the model is to be rendered.

Controller Test

In the test source folder, create a Kotlin source file in package seg3x02.converter called WebControllerTest. Edit as follow.

```
1.  package seg3x02.converter
2.
3.  import org.junit.jupiter.api.Test
4.  import org.springframework.beans.factory.annotation.Autowired
5.  import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest
6.  import org.springframework.test.web.servlet.MockMvc
7.  import org.springframework.test.web.servlet.request.MockMvcRequestBuilders
8.  import org.springframework.test.web.servlet.result.MockMvcResultMatchers
9.
10. @WebMvcTest
11. class WebControllerTest {
12.     @Autowired
13.     lateinit var mockMvc: MockMvc
14.
15.     @Test
16.     fun request_to_home() {
17.         mockMvc.perform(MockMvcRequestBuilders.get("/"))
```

```
18.             .andExpect(MockMvcResultMatchers.status().isOk)
19.             .andExpect(MockMvcResultMatchers.view().name("home"))
20.     }
21.
22.     @Test
23.     fun celsius_to_fahrenheit_conversion() {
24.         mockMvc.perform(
25.             MockMvcRequestBuilders.get("/convert")
26.             .param("celsius", "0")
27.             .param("fahrenheit", "")
28.             .param("operation", "CtoF"))
29.             .andExpect(MockMvcResultMatchers.status().isOk)
30.             .andExpect(MockMvcResultMatchers.model().attribute("fahrenheit", "32.00"))
31.             .andExpect(MockMvcResultMatchers.view().name("home"))
32.
33.     }
34. }
```

The test class is annotated @WebMvcTest to provide the necessary context to test Web MVC. This makes a MockMvc bean available for injection (lines 12-13). The MockMvc bean allows us to issue web requests and have these handled by the controller without the need for a running server. The status response codes and response content can be verified using MockMvcResultMatchers.

Test request_to_home (lines 15-20) checks the proper handling of requests to path "/". We use the MockMvc bean to perform a GET request on line 17. This returns a result on which we can perform expectations with MockMvcResultMatchers. The expected response status is checked at line 18 and the returned view name at line 19.

Run the test to verify that it passes. You may run tests in your IDE or in the command line with Gradle. In the root folder of the project, enter command `./gradlew test` this will build the project and run all the tests.

Thymeleaf Template

We need to create a template to match the logical name "*home*" as expected by the Web Controller. Create a folder called templates in src/resources and create a home.html file in src/resources/template.

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.      <meta charset="UTF-8">
5.      <title>Temperature Converter</title>
6.      <link rel="stylesheet" th:href="@{/css/style.css}" />
7.  </head>
```

```
8.  <body>
9.  <div>
10.    <h1>Temperature Converter</h1>
11.    <div th:switch="${error}">
12.       <p th:case="'FahrenheitFormatError'">
13.          Wrong value provided for Fahrenheit - must be a number
14.       </p>
15.       <p th:case="'CelsiusFormatError'">
16.          Wrong value provided for Celsius - must be a number
17.       </p>
18.       <p th:case="'OperationFormatError'">
19.          Wrong operation
20.       </p>
21.    </div>
22.    <form th:action="@{/convert}" method="get">
23.       <table>
24.          <tr>
25.             <td><label for="celsius">Celsius:</label></td>
26.             <td><input name="celsius"
27.                        id="celsius"
28.                        th:name="celsius" th:value="${celsius}"></td>
29.          </tr>
30.          <tr>
31.             <td><label for="fahrenheit">Fahrenheit:</label></td>
32.             <td><input name="fahrenheit"
33.                        id="fahrenheit"
34.                        th:name="fahrenheit" th:value="${fahrenheit}"></td>
35.          </tr>
36.          <tr>
37.             <td><button type="submit" th:name="operation" th:value="CtoF">
38.                Celsius to Fahrenheit</button></td>
39.             <td><button type="submit" th:name="operation" th:value="FtoC">
40.                Fahrenheit to Celsius</button></td>
41.          </tr>
42.       </table>
43.    </form>
44. </div>
45. </body>
46. </html>
```

A thymeleaf template is a HTML page with additional thymeleaf attributes. Context variables including those set as model attributes are accessible in the template. We use a th:switch on lines 11-21 to display an appropriate error message for the value of the model attribute error. Each <*p*> element in the <*div*> is rendered only when the embedded th:case matches attribute *error*.

A form on lines 22-43 captures user input and requested conversion. The action of the form is set with thymeleaf attribute th:action to the URI path "*/convert*". The <*input*> elements for Celsius (26-28) and

Fahrenheit values (32-34) are bound to the model attributes *celsuis* and *fahrenheit* respectively with the thymeleaf attribute th:value on lines 28 and 34. We use thymeleaf attribute th:name to submit the parameter values for celsius (line 28), fahrenheit (line 34) and operation (lines 37, 39) with the form action.

CSS File

Create a folder called css in src/resources/static and create a style.css file in src/resources/static/css. Edit as follow.

```
1.  body {
2.      background-color: lightgray;
3.      margin-left: 70px;
4.      margin-top: 20px;
5.  }
6.
7.  h1 {
8.      font-size: 30px;
9.      font-family: serif;
10.     font-weight: bold;
11.     background-color: aliceblue;
12. }
13.
14. input {
15.     border-style: double;
16.     font-size: 18px;
17.     width: 150px;
18. }
19. button {
20.     background-color: darkcyan;
21.     padding: 20px 25px;
22.     font-size: 18px;
23.     color: white;
24. }
25.
26. label {
27.     font-size: 20px;
28. }
29.
30. p {
31.     font-size: 20px;
32.     color: red;
33.     font-weight: bold;
34. }
```

Building and running

Your IDE likely provides an integrated way to build and run Springboot applications. You may also build and run on the command line with Gradle.

In the root folder of the project, enter command `./gradlew bootRun` this will build the project and start the application. Navigate to `http://localhost:8080/` to execute.

`Exercise`

The following exercise is the deliverable for the lab. Complete and check-in the code to Github Classroom before the deadline. Only this exercise will be evaluated.

Reimplement the calculator from the previous lab as a Spring MVC application with Thymeleaf.