SEG3502 – Lab 2

Spring

Introduction Spring MVC

L'objectif de ce laboratoire est une introduction au framework Spring avec Springboot. Nous allons traiter de l'installation des outils nécessaires et créer une application Web avec Spring MVC.

Le code source du laboratoire est disponible dans le référentiel Github https://github.com/stephanesome/springBootIntro.

Setup de Spring

Spring est un framework qui s'exécute sur la Machine Virtuelle Java (JVM). Vous devez donc avoir installé un kit de développement Java.

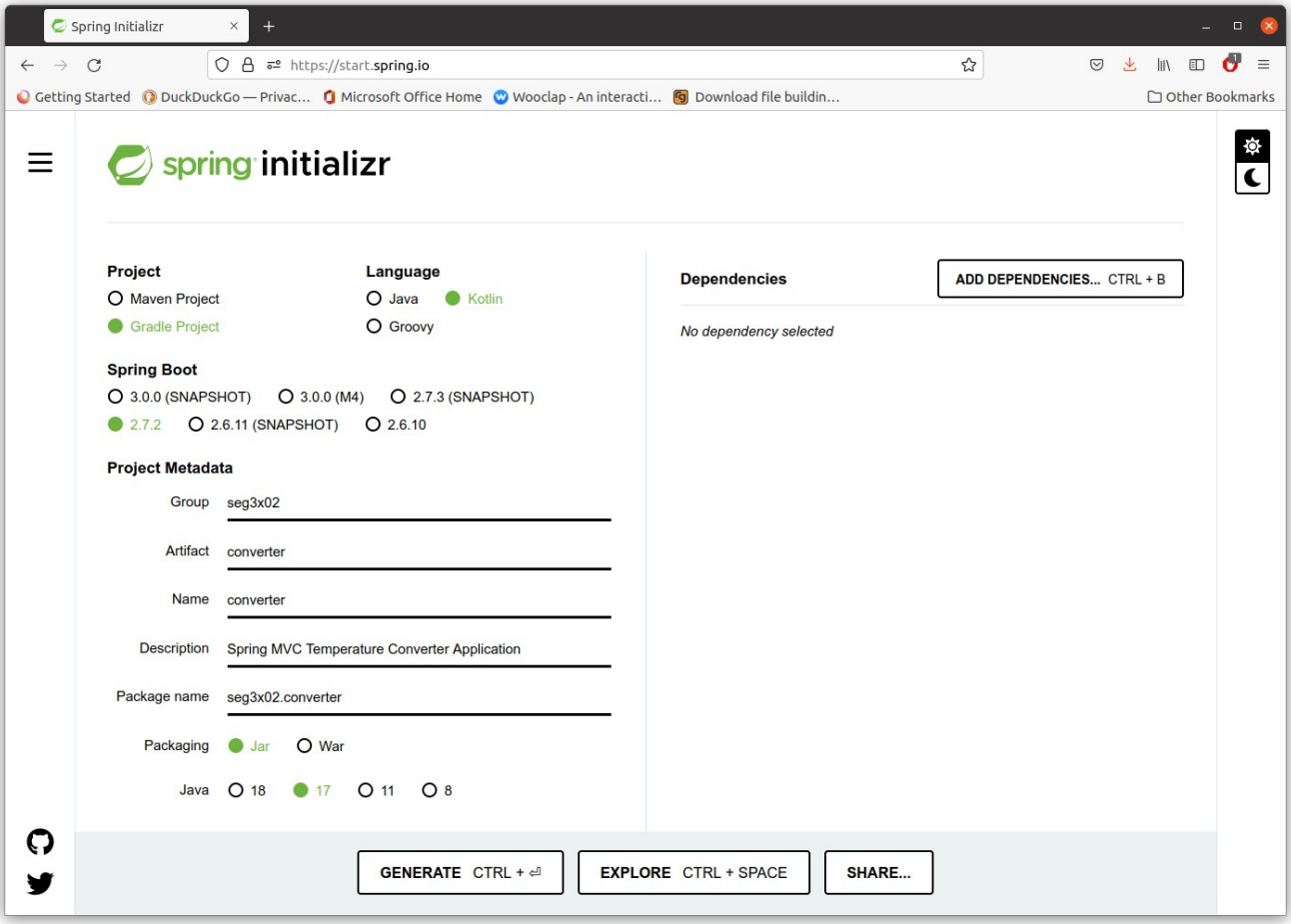
Un environnement de développement intégré (IDE) est fortement recommandé. Les choix incluent IntelliJ IDEA , Spring Tools, Visual Studio Code, ou Eclipse. Je suggère IntelliJ IDEA pour lequel vous pouvez obtenir une licence étudiante.

Nous utiliserons Kotlin comme langage de développement. Kotlin est un langage qui compile vers la JVM et est compatible avec Java. Il ajoute cependant de nombreuses améliorations à Java, notamment la suppression du code boilerplate extensif qui est caractéristique de Java. Reportez-vous ici pour la documentation sur Kotlin.

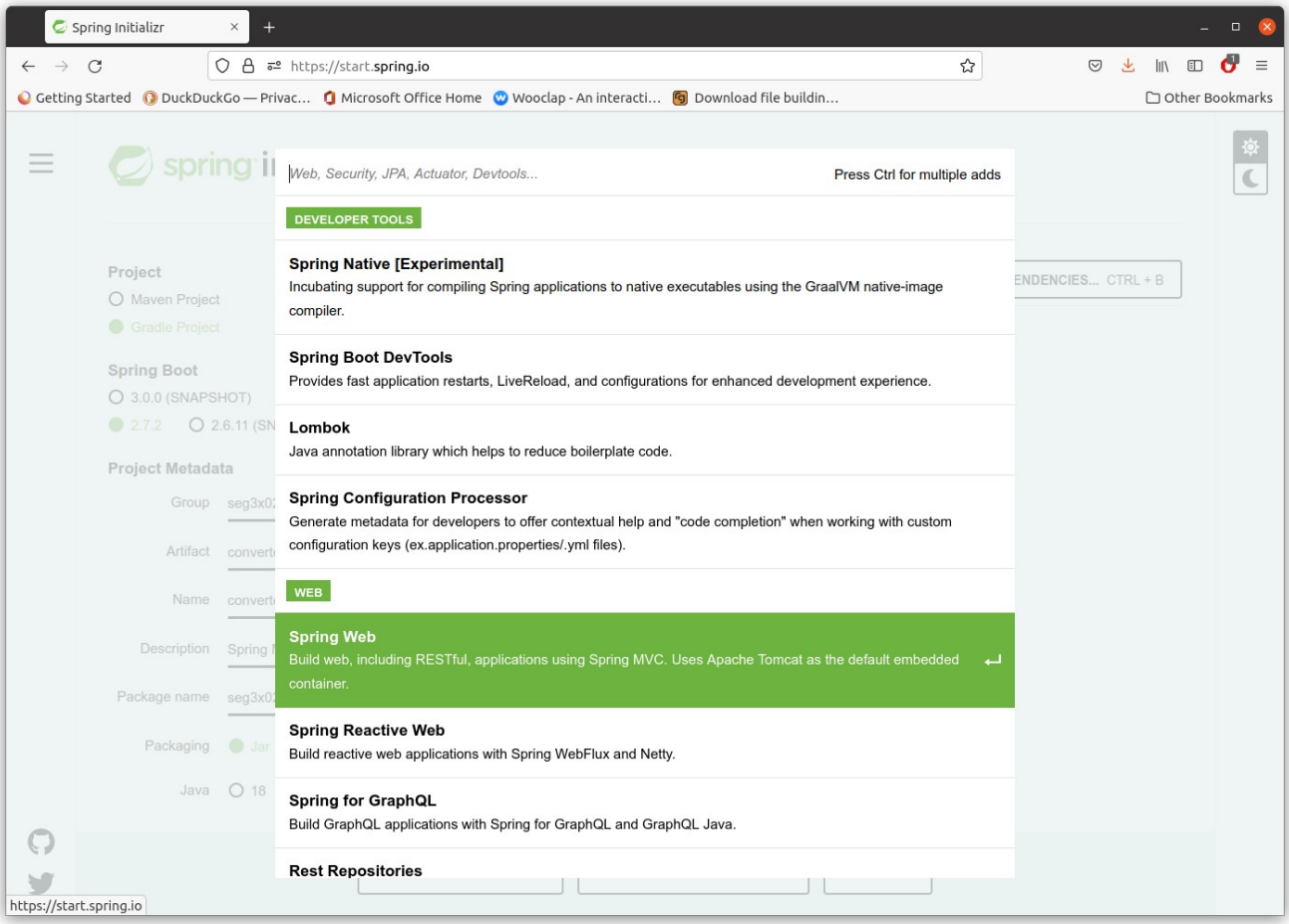
Pour une édition correcte du code Kotlin, vous devez ajouter le plugin approprié pour Kotlin à votre IDE (ex : pour IntelliJ IDEA, Spring Too Suite, VS Code).

Générer l’application

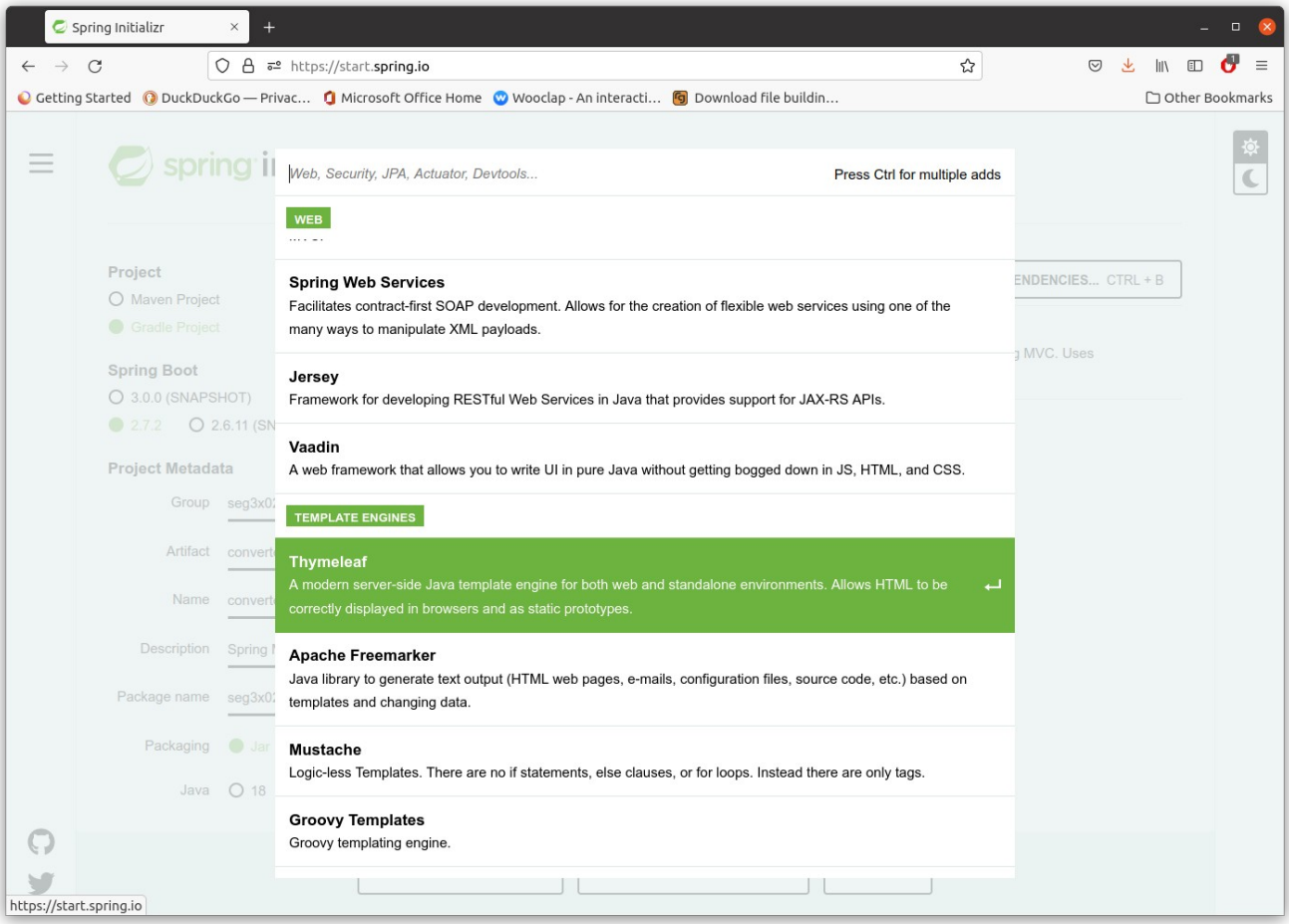
Spring fournit un service Web appelé Spring Intializr pour la configuration des applications. Vous pouvez accéder au service à l'adresse https://start.spring.io/.



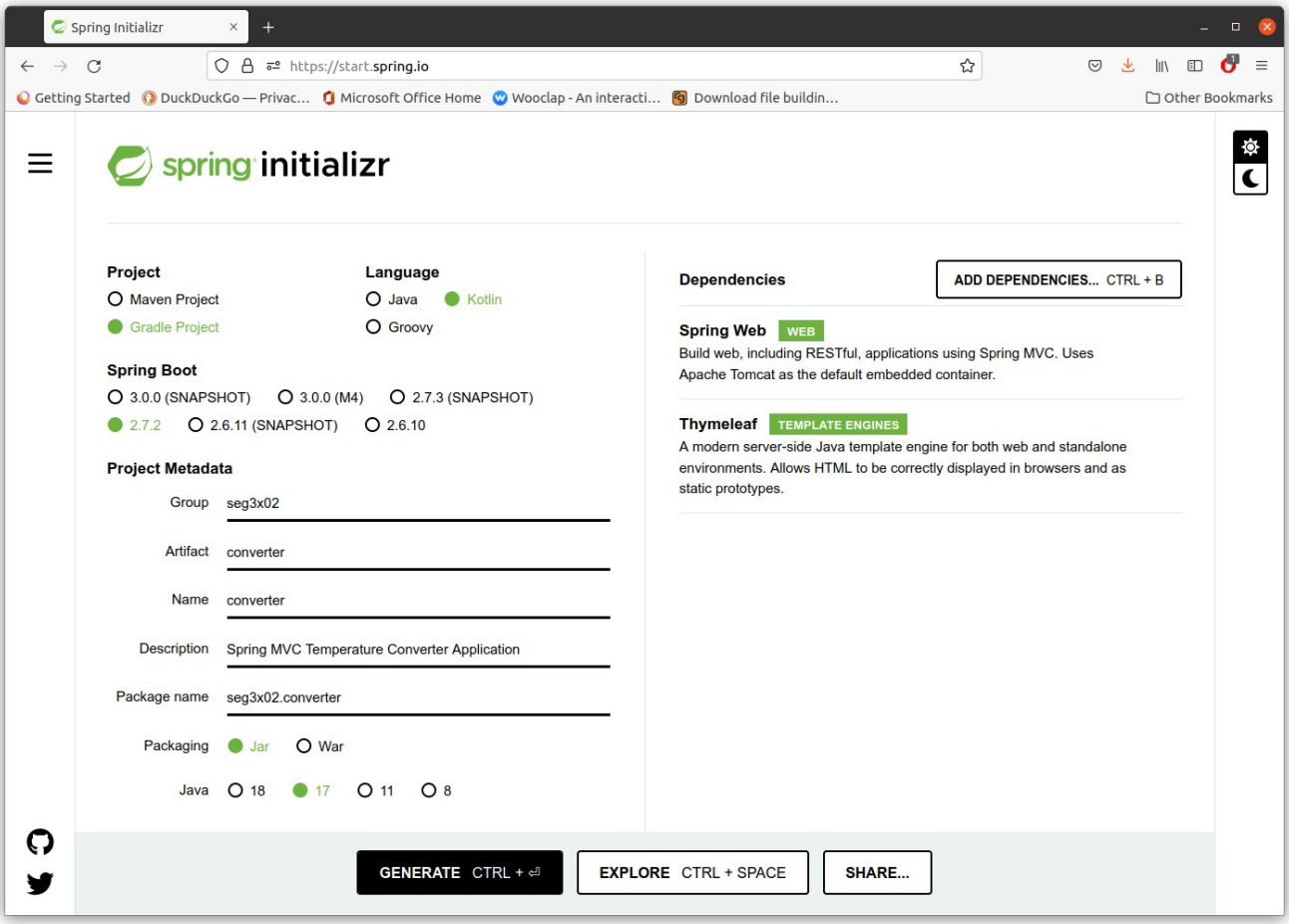
- Sélectionnez Gradle Projet comme type de projet et Kotlin comme langage
- Laissez la version Spring Boot sélectionnée par défaut
- Entrez les métadonnées (Metadata) du projet
- Assurez-vous que la version de JDK sélectionnée est installée
- Dans la section Dependencies, cliquez sur Add. Sélectionnez Spring Web



- Cliquez Add Dependencies une fois de plus puis sélectionnez Thymeleaf.

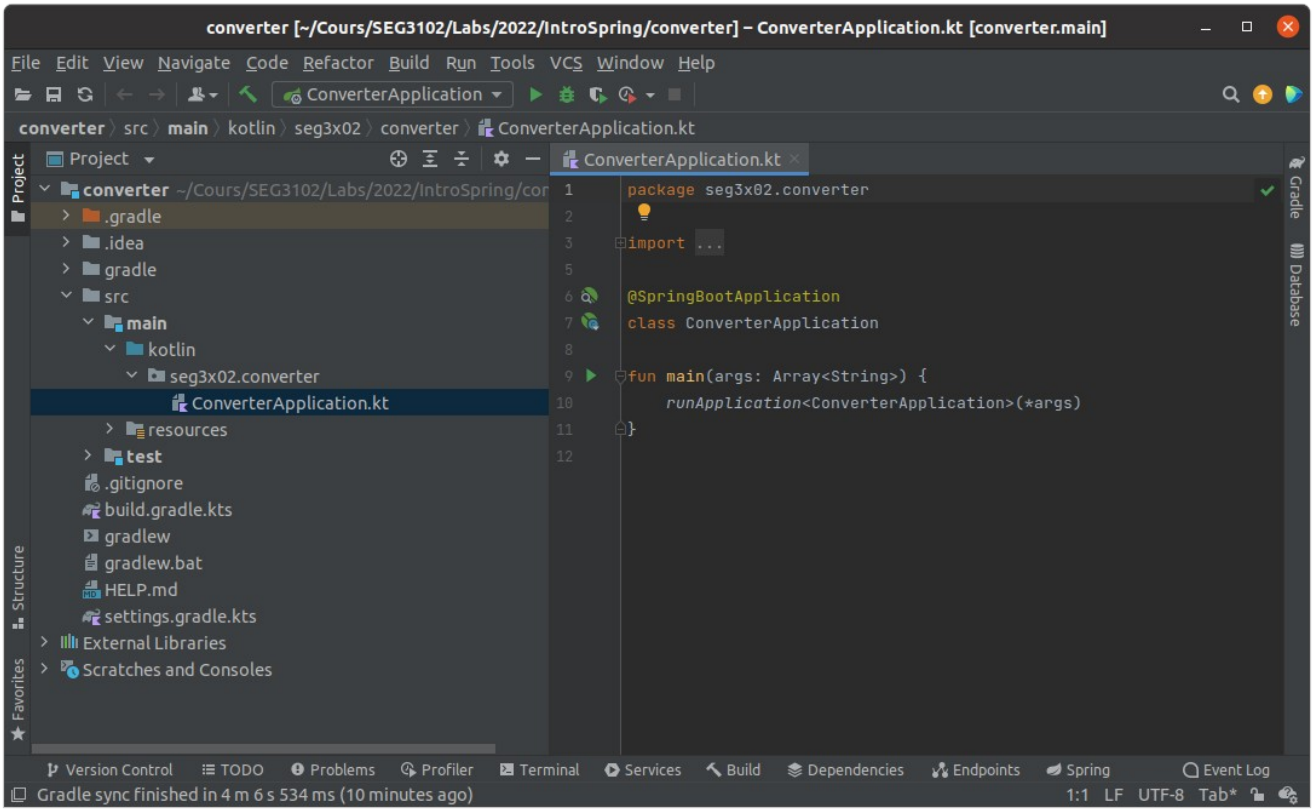


- Générez le projet et téléchargez le fichier compressé zip généré.



Développement de l'Application

Décompressez le fichier compressé téléchargé et importez le projet dans votre IDE.



L'application générée comprend toute la configuration nécessaire, y compris un fichier de configuration Gradle (build.gradle.kts), des scripts exécutables pour Gradle (gradlew, gradlew.bat). De plus, l'application comprend un fichier principal (ConverterApplication.kt) avec une classe annotée @SpringBootApplication. Cela marque la classe en tant que classe de configuration et active la configuration automatique (auto-configuration) et la découverte des composants (component scanning). Le fichier principal comprend également la fonction principale qui est exécutée pour lancer l'application.

Contrôleur

Dans le dossier source principal, créez un fichier source Kotlin dans le package seg3x02.converter de nom WebController. Éditez comme suit.

```
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
```

```
class WebController {
    @ModelAttribute
    fun addAttributes(model: Model) {
        model.addAttribute("error", "")
        model.addAttribute("celsius", "")
        model.addAttribute("fahrenheit", "")
    }

    @RequestMapping("/")
    fun home(): String {
        return "home"
    }

    @GetMapping(value = ["/convert"])
    fun doConvert(
        @RequestParam(value = "celsius", required = false) celsius: String,
        @RequestParam(value = "fahrenheit", required = false) fahrenheit: String,
        @RequestParam(value = "operation", required = false) operation: String,
        model: Model
    ): String {
        var celsiusVal: Double
        var fahrenheitVal: Double
        when (operation) {
            "CtoF" ->
                try {
                    celsiusVal = celsius.toDouble()
                    fahrenheitVal = ((celsiusVal * 9) / 5 + 32)
                    model.addAttribute("celsius", celsius)
                    model.addAttribute("fahrenheit", String.format("%.2f", fahrenheitVal))
                } catch (exp: NumberFormatException) {
                    model.addAttribute("error", "CelsiusFormatError")
                    model.addAttribute("celsius", celsius)
                    model.addAttribute("fahrenheit", fahrenheit)
                }
            "FtoC" ->
                try {
                    fahrenheitVal = fahrenheit.toDouble()
                    celsiusVal = ((fahrenheitVal - 32) * 5) / 9
                    model.addAttribute("celsius", String.format("%.2f", celsiusVal))
                    model.addAttribute("fahrenheit", fahrenheit)
                } catch (exp: NumberFormatException) {
                    model.addAttribute("error", "FahrenheitFormatError")
                    model.addAttribute("celsius", celsius)
                    model.addAttribute("fahrenheit", fahrenheit)
                }
            else -> {
                model.addAttribute("error", "OperationFormatError")
                model.addAttribute("celsius", celsius)
                model.addAttribute("fahrenheit", fahrenheit)
```

```
60.             }
61.         }
62.         return "home"
63.     }
64. }
```

L'annotation @Controller (ligne 10) spécifie la classe en tant que contrôleur MVC.

Nous utilisons une fonction annotée @ModelAttribute aux lignes 12 à 17 pour initialiser les attributs du modèle. La fonction addAttributes sera exécutée avant l'invocation de toute fonction de gestionnaire dans la classe pour définir ou réinitialiser les attributs du modèle.

La fonction home aux lignes 19-22, est un gestionnaire pour le chemin racine "/" comme spécifié par l'annotation *@RequestMapping*. Il gère toutes les requêtes vers l'URL construite en préfixant le chemin avec l'URL de déploiement du serveur et le port de l'application. Pour un déploiement local, ce serait par défaut `http://localhost:8080/`. La fonction retourne la vue logique "*home*" qui est mappée au template *src/resources/templates/home.html*.

La fonction doConvert aux lignes 24-63, gère les requêtes HHTP GET à "*/convert*". Les requêtes peuvent transmettre trois paramètres (*celsius*, *fahrenheit*, *operation*) qui sont mappés aux arguments de la fonction avec l'annotation @RequestParam. Ces paramètres permettent de déterminer le type de conversion et la valeur à convertir. La conversion Celsius en Fahrenheit est effectuée aux lignes 34-44 et la conversion Fahrenheit en Celsius aux lignes 45-55. Le résultat de la conversion est stocké en remplaçant la valeur de l'attribut de model (lignes 38 et 49). L'attribut de modèle "*error*" est utilisé pour signaler les situations d'erreur aux lignes 41, 52 et 57. La fonction retourne la vue logique "*home*" dans laquelle le modèle doit être rendu.

Test du contrôleur

Dans le dossier source test, créez un fichier Kotlin dans le package seg3x02.converter appelé WebControllerTest. Modifier comme suit.

```
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. }
```

La classe de test est annotée @WebMvcTest pour fournir le contexte nécessaire pour tester Web MVC. Cela rend un bean MockMvc, disponible pour l'injection (lignes 12-13). Le bean MockMvc nous permet d'émettre des requêtes Web et de les faire traiter par le contrôleur sans avoir besoin d'un serveur en cours d'exécution. Les codes de réponse d'état et le contenu de la réponse peuvent être vérifiés à l'aide de MockMvcResultMatchers.

Le test request_to_home (lignes 15-20) vérifie le bon traitement des requêtes vers le chemin "/". Nous utilisons le bean MockMvc pour effectuer une requête GET à la ligne 17. Cela retourne un résultat sur lequel nous pouvons vérifier des attentes avec MockMvcResultMatchers. Le statut de réponse attendu est vérifié à la ligne 18 et le nom de la vue renvoyée à la ligne 19.

Exécutez le test pour vérifier qu'il réussit. Vous pouvez exécuter des tests dans votre IDE ou en ligne de commande avec Gradle. Dans le dossier racine du projet, entrez la commande `./gradlew test` cela construira le projet et exécutera tous les tests.

Template Thymeleaf

Nous devons créer un template correspondant au nom logique "*home*" attendu par le contrôleur Web. Créez un dossier appelé templates dans src/resources et créez un fichier home.html dans 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>
```

Un template thymeleaf est une page HTML avec des attributs thymeleaf supplémentaires. Les variables de contexte, y compris celles définies comme attributs de modèle, sont accessibles dans le template. Nous utilisons un th:switch aux lignes 11 à 21 pour afficher un message d'erreur approprié pour la valeur de l'attribut de modèle *error*. Chaque élément <*p*> dans le <*div*> est rendu uniquement lorsque le *th:case* intégré correspond à l'attribut *error*.

Un formulaire aux lignes 22 à 43 capture les entrées de l'utilisateur et la conversion demandée. L'action du formulaire est définie avec l'attribut thymeleaf th:action sur le chemin URI "*/convert*". Les éléments *<input>* pour les valeurs Celsius (26-28) et Fahrenheit (32-34) sont liés aux attributs de modèle *celsuis* et *fahrenheit* respectivement avec l'attribut thymeleaf th:value sur les lignes 28 et 34. Nous utilisons l'attribut thymeleaf th:name pour soumettre les valeurs des paramètres pour Celsius (ligne 28), Fahrenheit (ligne 34) et operation (lignes 37, 39) avec l'action du formulaire.

Fichier CSS

Créez un dossier appelé css dans src/resources/static et créez un fichier style.css dans src/resources/static/css. Modifier comme suit.

```
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

Votre IDE fournit probablement un moyen intégré pour construire et exécuter des applications Springboot. Vous pouvez également construire et exécuter sur la ligne de commande avec Gradle.

Dans le dossier racine du projet, entrez la commande `./gradlew bootRun` cela construira le projet et demarre l'application. Naviguez à `http://localhost:8080/` pour exécuter.

`Exercise`

L'exercice suivant est le livrable pour le laboratoire. Complétez et téléversez le code dans Github Classroom avant la date limite. Seul cet exercice sera évalué.

Reimplementez la calculatrice du lab précédent comme une application Spring MVC avec Thymeleaf.