

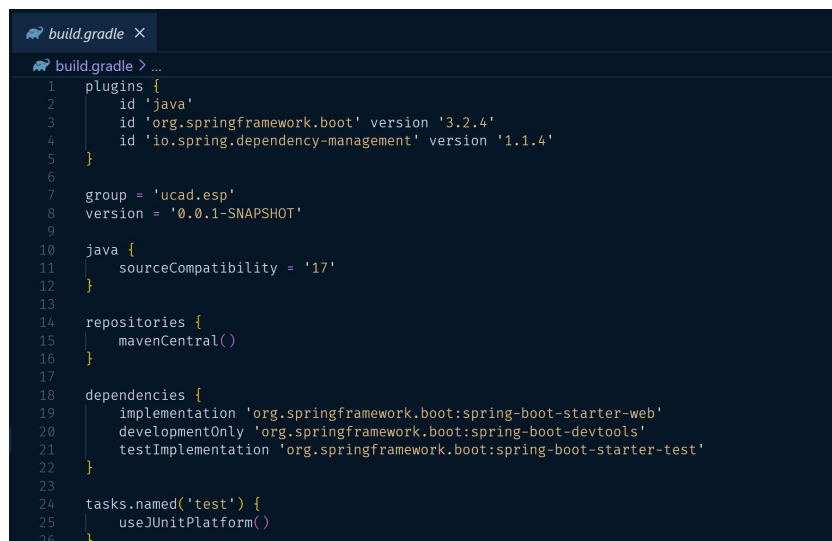
Développement d'applications Java

Lab_1 : Premiers pas avec Springboot CLI

Comprendre Gradle

Gradle est un outil d'automatisation de construction (build) qui simplifie le processus de développement logiciel et unifie également le processus de développement. Il gère les dépendances de notre projet et gère le processus de construction.

La configuration de Gradle est effectuée dans le fichier **build.gradle** du projet. Le fichier peut être personnalisé pour répondre aux besoins spécifiques du projet et peut être utilisé pour automatiser des tâches telles que la construction, les tests et le déploiement du logiciel. Le fichier **build.gradle** est une partie importante du système de construction Gradle et est utilisé pour configurer et gérer le processus de construction d'un projet logiciel. Le fichier build.gradle comprend généralement des informations sur les dépendances du projet, telles que les bibliothèques externes et les frameworks nécessaires au projet pour compiler. Vous pouvez utiliser soit les langages de programmation **Kotlin** ou **Groovy** pour écrire des fichiers build.gradle¹. Dans ce lab, nous utilisons **Groovy**. Voici un exemple de fichier build.gradle d'un projet Spring Boot :



```
1 plugins {
2     id 'java'
3     id 'org.springframework.boot' version '3.2.4'
4     id 'io.spring.dependency-management' version '1.1.4'
5 }
6
7 group = 'ucad.esp'
8 version = '0.0.1-SNAPSHOT'
9
10 java {
11     sourceCompatibility = '17'
12 }
13
14 repositories {
15     mavenCentral()
16 }
17
18 dependencies {
19     implementation 'org.springframework.boot:spring-boot-starter-web'
20     developmentOnly 'org.springframework.boot:spring-boot-devtools'
21     testImplementation 'org.springframework.boot:spring-boot-starter-test'
22 }
23
24 tasks.named('test') {
25     useJUnitPlatform()
26 }
```

Le fichier build.gradle contient généralement les parties suivantes :

- **Plugins** : Le bloc plugins définit les plugins Gradle utilisés dans le projet. Dans ce bloc, nous pouvons définir la version de Spring Boot.
- **Repositories** : Le bloc repositories définit les dépôts de dépendances utilisés pour résoudre les dépendances. Nous utilisons le dépôt Maven Central, à partir duquel Gradle extrait les dépendances.
- **Dependencies** : Le bloc dependencies spécifie les dépendances utilisées dans le projet.
- **Tasks** : Le bloc tasks définit les tâches qui font partie du processus de construction, telles que les tests.

¹ <https://stackshare.io/stackups/groovy-vs-kotlin>

La chose la plus importante est de comprendre la structure du fichier build.gradle et comment ajouter de nouvelles dépendances.

Création d'un projet Spring Boot

Pour la création d'un projet Spring Boot, nous pouvons évoquer les commandes ci-dessous qui créent d'abord un projet demo, puis le dézippe en renommant le projet Sb-app-o. Nous avons les dépendances **web** et **devtools**.

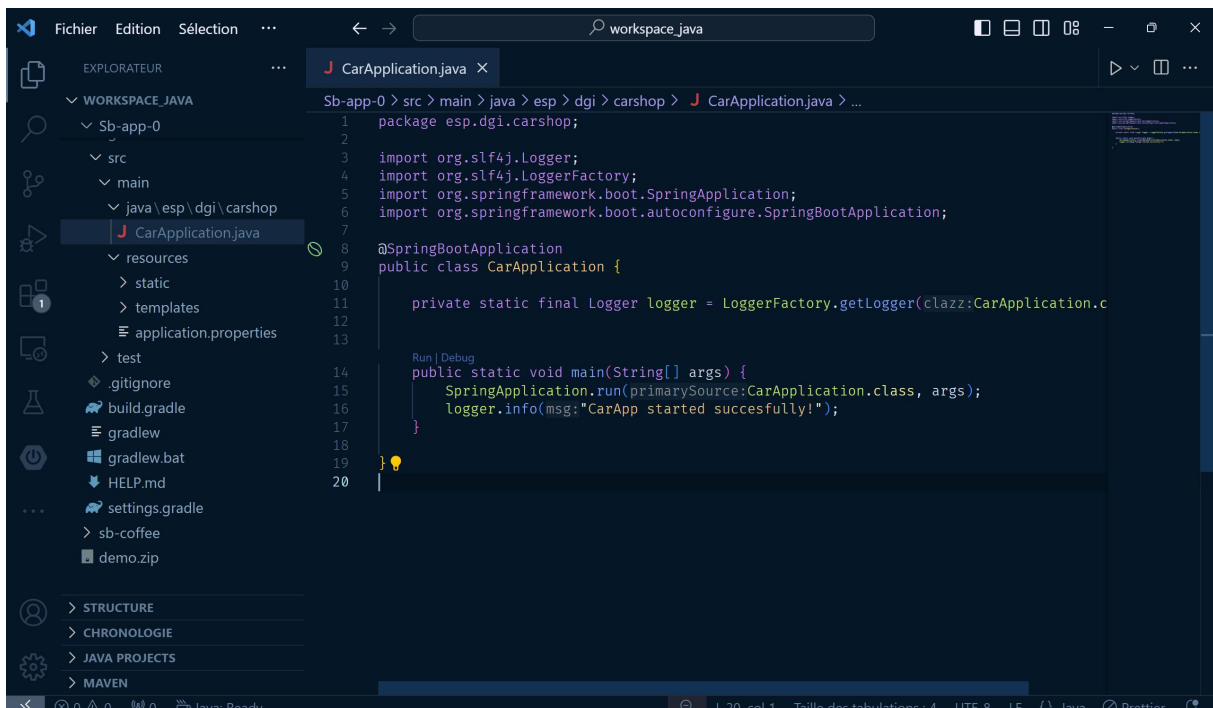
```
spring init --dependencies=web,devtools --type=gradle-project
```

```
unzip demo.zip -d Sb-app-o
```

Après la phase de création du projet, nous pouvons explorer le contenu du projet en se plaçant dans le projet puis l'ouvrir dans vs. Code en faisant la commande code suivi d'un point :

code .

Nous pouvons effectuer les petites modifications sur les noms de package utilisés (pas obligés). Nous pouvons également regarder la structure de notre projet. Il faut un certain temps avant que le projet ne soit prêt car toutes les dépendances seront téléchargées par Gradle après les avoir importées.



L'Explorateur de projets affiche également la structure des packages de notre projet. Au début, il n'y a qu'un seul package appelé **esp.dgi.carshop**. Sous ce package se trouve notre classe d'application principale, appelée **CarApplication.java**. Ignorer les lignes que j'ai ajoutées, j'y reviendrai plus tard.

Pour l'instant, notre application n'a pas de fonctionnalité, mais nous pouvons l'exécuter pour voir si tout démarre correctement. Pour exécuter le projet, nous faisons les deux commandes ci-dessous :

```
./gradlew /* va construire le projet */
```

```
./gradlew bootRun /* va lancer le projet */
```

```

MINGW64/c/codes/eclipse_codes/workspace_java/Sb-app-0
DELL XPS8560T0P-T42TNNR MINGW64 /c/codes/eclipse_codes/workspace_java/Sb-app-0
$ ./gradlew bootRun

> Task :bootRun

:: Spring Boot :: (v3.2.4)

2024-03-25T19:19:26.888Z INFO 25628 --- [car_app] [ restartedMain] esp.dgi.carshop.CarApplication : Starting CarApplication using Java 21 with PID 25628 (C:\codes\eclipse_codes\workspace_java\Sb-app-0\build\classes\java\main started by DELL XPS in C:\codes\eclipse_codes\workspace_java\Sb-app-0)
2024-03-25T19:19:26.888Z INFO 25628 --- [car_app] [ restartedMain] esp.dgi.carshop.CarApplication : No active profile set, falling back to 1 default profile: 'default'
2024-03-25T19:19:26.938Z INFO 25628 --- [car_app] [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devtools property defaults active! Set 'spring.devtools.add-properties' to 'false' to disable
2024-03-25T19:19:26.938Z INFO 25628 --- [car_app] [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider setting the 'logging.level.web' property to 'DEBUG'
2024-03-25T19:19:27.575Z INFO 25628 --- [car_app] [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8081 (http)
2024-03-25T19:19:27.590Z INFO 25628 --- [car_app] [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-03-25T19:19:27.590Z INFO 25628 --- [car_app] [ restartedMain] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.19]
2024-03-25T19:19:27.622Z INFO 25628 --- [car_app] [ restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-03-25T19:19:27.622Z INFO 25628 --- [car_app] [ restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 684 ms
2024-03-25T19:19:27.826Z INFO 25628 --- [car_app] [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2024-03-25T19:19:27.842Z INFO 25628 --- [car_app] [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8081 (http) with context path ''
2024-03-25T19:19:27.858Z INFO 25628 --- [car_app] [ restartedMain] esp.dgi.carshop.CarApplication : Started CarApplication in 1.291 seconds (process running for 1.594)
2024-03-25T19:19:27.858Z INFO 25628 --- [car_app] [ restartedMain] esp.dgi.carshop.CarApplication : CarApp started succesfully!
<-- 80% EXECUTING [4s]
> :bootRun

```

À la racine de notre projet, se trouve le fichier build.gradle, qui est le fichier de configuration Gradle de notre projet. Si vous examinez les dépendances à l'intérieur du fichier, vous pouvez voir qu'il y a maintenant des dépendances que nous avons sélectionnées en créant notre projet Il y a également une dépendance de test incluse automatiquement, comme illustré dans l'extrait de code suivant :

```

1 plugins {
2     id 'java'
3     id 'org.springframework.boot' version '3.2.4'
4     id 'io.spring.dependency-management' version '1.1.4'
5 }
6
7 group = 'ucad.esp'
8 version = '0.0.1-SNAPSHOT'
9
10 java {
11     sourceCompatibility = '17'
12 }
13
14 repositories {
15     mavenCentral()
16 }
17
18 dependencies {
19     implementation 'org.springframework.boot:spring-boot-starter-web'
20     developmentOnly 'org.springframework.boot:spring-boot-devtools'
21     testImplementation 'org.springframework.boot:spring-boot-starter-test'
22 }
23
24 tasks.named('test') {
25     useJUnitPlatform()
26 }
27

```

Dans les prochains travaux pratiques, nous allons ajouter plus de fonctionnalités à notre application, puis nous ajouterons plus de dépendances manuellement au fichier build.gradle. Spring utilise la philosophie du « convention over configuration² ».

Examinons la classe principale de Spring Boot de manière plus approfondie :

Au début de la classe, il y a l'annotation @SpringBootApplication, qui est en réalité une combinaison de plusieurs annotations :

² Je vous demanderai ce que cela veut dire !!!

Annotation	Description
@EnableAutoConfiguration	Cela permet la configuration automatique de Spring Boot, donc votre projet sera automatiquement configuré en fonction des dépendances. Par exemple, si vous avez la dépendance spring-boot-starter-web (web), Spring Boot suppose que vous développez une application web et configure votre application en conséquence.
@ComponentScan	Cela permet à l'analyse des composants de Spring Boot de trouver tous les composants de votre application.
@Configuration	Cela définit une classe qui peut être utilisée comme source de définitions de beans.

L'exécution de l'application démarre à partir de la méthode `main()`, comme dans les applications Java standard.

NB :

Il est recommandé de placer la classe d'application principale dans le package racine au-dessus des autres classes. Tous les packages sous le package contenant la classe d'application seront couverts par l'analyse des composants de Spring Boot. Une raison courante pour laquelle une application ne fonctionne pas correctement est que Spring Boot ne parvient pas à trouver des classes critiques.

Outils de développement Spring Boot

Les outils de développement Spring Boot simplifient le processus de développement de l'application. La fonction la plus importante des outils de développement est le redémarrage automatique chaque fois que des fichiers sur le chemin de classe sont modifiés. Les projets incluront les outils de développement si la dépendance suivante est ajoutée au fichier `build.gradle` de Gradle :

developmentOnly 'org.springframework.boot:spring-boot-devtools'

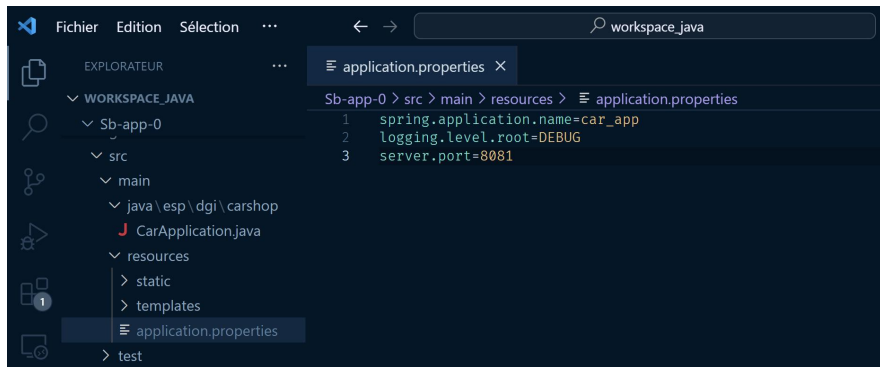
Les outils de développement sont désactivés lorsque vous créez une version entièrement empaquetée destinée à la production de votre application.

Logs et résolution de problèmes

Les journaux peuvent être utilisés pour surveiller le flux de votre application, et c'est un bon moyen de capturer les erreurs inattendues dans votre code de programme. Le package de démarrage de Spring Boot fournit Logback, que nous pouvons utiliser pour la journalisation sans aucune configuration. Le code d'exemple précédent montre comment vous pouvez utiliser la journalisation. Logback utilise Simple Logging Façade for Java (**SLF4J**) comme interface native.

La méthode `logger.info` imprime un message de journal dans la console. Les messages de journal peuvent être consultés dans la console après avoir exécuté un projet, comme illustré dans la capture d'exécution ci-haut.

Il existe sept niveaux différents de journalisation : TRACE, DEBUG, INFO, WARN, ERROR, FATAL et OFF. Vous pouvez configurer le niveau de journalisation dans votre fichier **application.properties** de Spring Boot. Le fichier peut être trouvé dans le dossier /resources à l'intérieur de votre projet.



Si nous définissons le niveau de journalisation sur DEBUG, nous pouvons voir les messages de journal des niveaux qui sont de niveau DEBUG ou supérieur (c'est-à-dire DEBUG, INFO, WARN et ERROR).

logging.level.root=DEBUG

Maintenant, lorsque vous exécutez le projet, vous ne pouvez plus voir les messages TRACE. Le niveau TRACE contient tous les détails du comportement de l'application, ce qui n'est pas nécessaire à moins que vous n'ayez besoin d'une visibilité complète de ce qui se passe dans votre application. Cela pourrait être un bon paramètre pour une version de développement de votre application. Le niveau de journalisation par défaut est INFO si vous ne définissez rien d'autre.

```
MINGW64/c:/codes/eclipse_codes/workspace_java/Sb-app-0
Did not match:
- @ConditionalOnClass did not find required class 'org.eclipse.jetty.ee10.websocket.jakarta.server.config.JakartaWebSocketServletContainerInitializer' (OnClassCondition)

WebSocketServletAutoConfiguration.UndertowWebSocketConfiguration:
Did not match:
- @ConditionalOnClass did not find required class 'io.undertow.websockets.jsr.Bootstrap' (OnClassCondition)

XDataSourceAutoConfiguration:
Did not match:
- @ConditionalOnClass did not find required class 'jakarta.transaction.TransactionManager' (OnClassCondition)

Exclusions:
None

Unconditional classes:

org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration
org.springframework.boot.autoconfigure.ssl.SslAutoConfiguration
org.springframework.boot.autoconfigure.context.LifecycleAutoConfiguration
org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration
org.springframework.boot.autoconfigure.availability.ApplicationAvailabilityAutoConfiguration
org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration

2024-03-25T19:40:04.274Z INFO 6888 --- [car_app] [ restartedMain] esp.dgi.carshop.CarApplication : Started CarApplication in 1.297 seconds (process running for 1.597)
2024-03-25T19:40:04.274Z DEBUG 6888 --- [car_app] [ restartedMain] o.s.b.a.ApplicationAvailabilityBean : Application availability state LivenessState changed to CORRECT
2024-03-25T19:40:04.274Z DEBUG 6888 --- [car_app] [ restartedMain] o.s.boot.devtools.restart.Restarter : Creating new Restarter for thread Thread[#1,main,5,main]
2024-03-25T19:40:04.274Z DEBUG 6888 --- [car_app] [ restartedMain] o.s.boot.devtools.restart.Restarter : Immediately restarting application
2024-03-25T19:40:04.274Z DEBUG 6888 --- [car_app] [ restartedMain] o.s.boot.devtools.restart.Restarter : Starting application esp.dgi.carshop.CarApplication with URLs [file:/C:/codes/eclipse_codes/workspace_java/Sb-app-0/build/classes/java/main/, file:/C:/codes/eclipse_codes/workspace_java/Sb-app-0/build/resources/main/]
2024-03-25T19:40:04.274Z DEBUG 6888 --- [car_app] [ restartedMain] o.s.b.a.ApplicationAvailabilityBean : Application availability state ReadinessState changed to ACCEPTING_TRAFFIC
2024-03-25T19:40:04.274Z INFO 6888 --- [car_app] [ restartedMain] esp.dgi.carshop.CarApplication : CarApp started successfully!
<====> 80% EXECUTING [5s]
> :bootRun
```

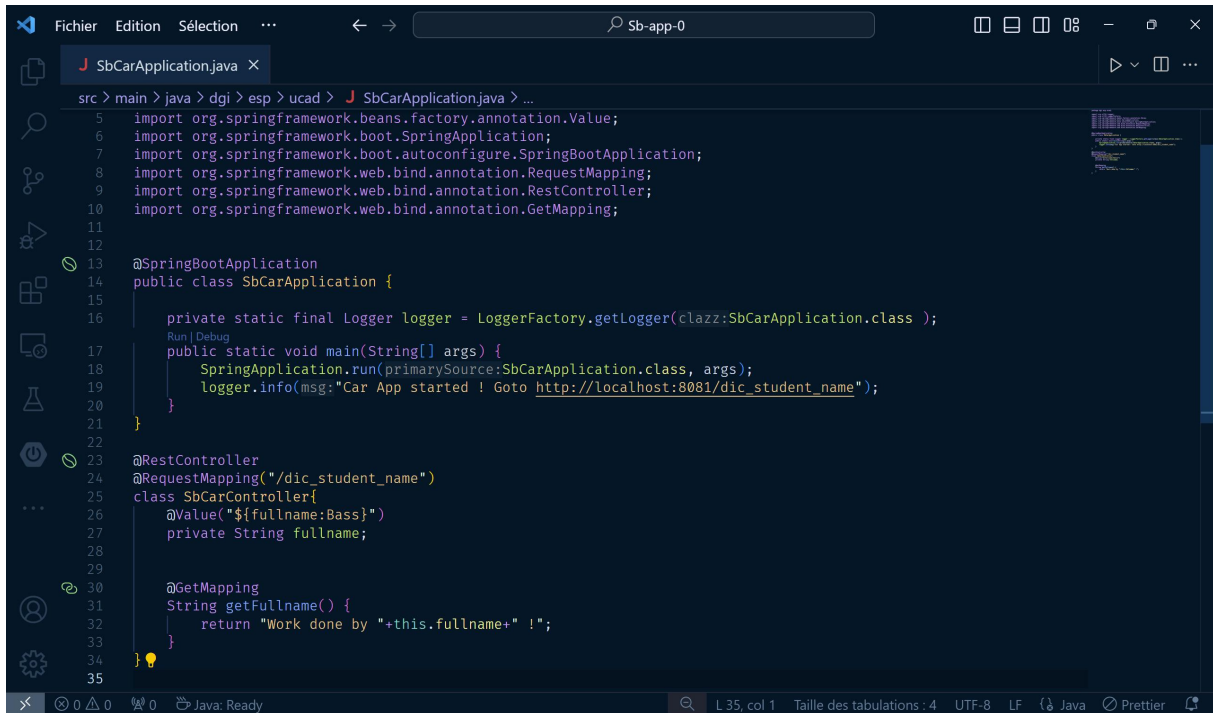
Il y a une défaillance courante que vous pourriez rencontrer lors de l'exécution d'une application Spring Boot. Spring Boot utilise Apache Tomcat (<http://tomcat.apache.org/>) comme serveur d'application par défaut, qui fonctionne sur le port 8080 par défaut. Vous pouvez changer le port dans le fichier application.properties. Le paramètre suivant démarrera Tomcat sur le port 8081 :

server.port=8081

Si le port est occupé, l'application ne démarrera pas, et vous verrez le message suivant APPLICATION FAILED TO START dans la console.

Exercice

Modifier le code de la classe principale, comme ci-dessous, puis envoyer la capture dans classroom.



```
src > main > java > dgi > esp > ucad > J SbCarApplication.java > ...
5  import org.springframework.beans.factory.annotation.Value;
6  import org.springframework.boot.SpringApplication;
7  import org.springframework.boot.autoconfigure.SpringBootApplication;
8  import org.springframework.web.bind.annotation.RequestMapping;
9  import org.springframework.web.bind.annotation.RestController;
10 import org.springframework.web.bind.annotation.GetMapping;
11
12
13 @SpringBootApplication
14 public class SbCarApplication {
15
16     private static final Logger logger = LoggerFactory.getLogger(clazz:SbCarApplication.class );
17     public static void main(String[] args) {
18         SpringApplication.run(primarySource:SbCarApplication.class, args);
19         logger.info(msg:"Car App started ! Goto http://localhost:8081/dic\_student\_name");
20     }
21
22
23 @RestController
24 @RequestMapping("/dic_student_name")
25 class SbCarController{
26     @Value("${fullname:Bass}")
27     private String fullname;
28
29
30     @GetMapping
31     String getFullname() {
32         return "Work done by "+this.fullname+" !";
33     }
34
35 }
```

En résultat, nous aurons la capture ci-dessous :

