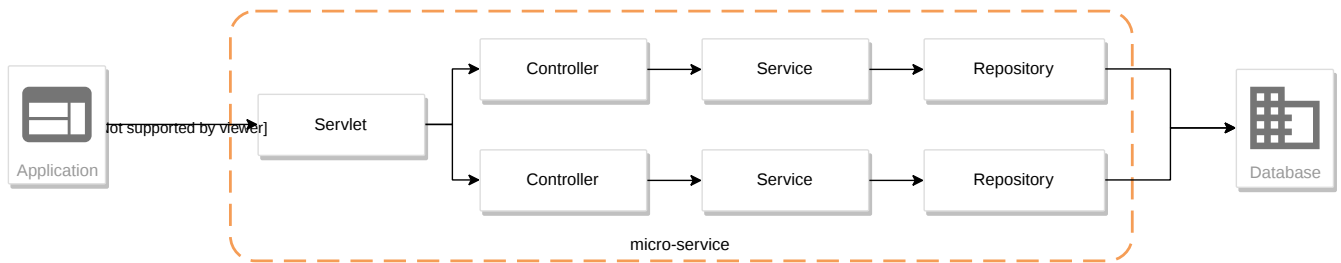


ALOM - TP - Spring-Boot

Table of Contents

1. Présentation et objectifs	1
2. GitLab	2
3. La classe PokemonType	3
3.1. La structure JSON	3
3.2. Les classes Java	5
4. Le repository "stub"	5
4.1. Le fichier de données pokemon.	5
4.2. Le test unitaire.	5
4.3. L'implémentation	6
4.4. Ajout de tests unitaires avec Spring	8
5. Le service	8
5.1. Le test unitaire.	9
5.2. L'implémentation	9
5.3. Implémentation avec Spring	10
6. Le Controlleur	11
6.1. Le test unitaire	11
6.2. L'implémentation	12
6.3. L'instrumentation pour spring-web !	13
6.3.1. Les tests unitaires	13
6.3.2. L'implémentation	13
6.4. L'exécution de notre projet !	14
6.4.1. Plus de logs !	15
6.4.2. Plus de test	16
7. Autres routes	17
8. Packager notre micro-service	17
8.1. Le plugin spring-boot maven.	18
8.2. Packager notre micro-service	18
9. Continuez à développer	19

1. Présentation et objectifs

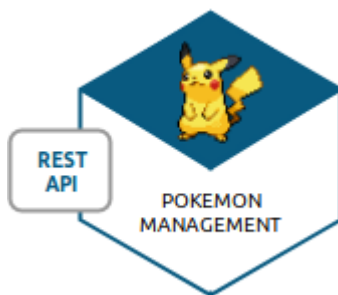


Le but est de reprendre à zéro le développement de notre architecture "à la microservice", en utilisant **spring-boot**

Pour rappel, dans cette architecture, chaque composant a son rôle précis :

- la servlet reçoit les requêtes HTTP, et les envoie au bon controller (rôle de point d'entrée de l'application)
- le controlleur implémente une méthode Java par route HTTP, récupère les paramètres, et appelle le service (rôle de routage)
- le service implémente le métier de notre micro-service
- le repository représente les accès aux données (avec potentiellement une base de données)

Et pour s'amuser un peu, nous allons réaliser un micro-service qui nous renvoie des données sur les Pokemons !



Nous allons développer :

1. un repository d'accès aux données de Pokemons (à partir d'un fichier JSON)
2. un service d'accès aux données
3. annoter ces composants avec les annotations de Spring et les tester
4. créer un controlleur spring pour gérer nos requêtes HTTP / REST
5. utiliser spring-boot pour instancier notre application !



Nous repartons de zéro pour ce TP !

2. GitLab

Identifiez-vous sur GitLab, et cliquez sur le lien suivant pour créer votre repository git: [GitLab classroom](#)

Clonez ensuite votre repository git sur votre poste !



N'oubliez pas de faire des commits et des push réguliers.

3. La classe PokemonType

Pour commencer, nous allons créer notre objet métier.

3.1. La structure JSON

Pour implémenter notre objet, nous devons nous inspirer des champs que propose l'API <https://pokeapi.co>.

Par exemple, voici ce qu'on obtient en appelant l'API (un peu simplifiée) :

Electhor !

```
{
  "base_experience": 261,
  "height": 16,
  "id": 145,
  "moves": [],
  "name": "zapdos",
  "sprites": {
    "back_default":
    "https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/back/145.png",
    "back_shiny":
    "https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/back/shiny/145.png",
    "front_default":
    "https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/145.png",
    "front_shiny":
    "https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/shiny/145.png"
  },
  "stats": [
    {
      "base_stat": 100,
      "effort": 0,
      "stat": {
        "name": "speed",
        "url": "https://pokeapi.co/api/v2/stat/6/"
      }
    },
    {
      "base_stat": 90,
      "effort": 0,
      "stat": {
```

```

        "name": "special-defense",
        "url": "https://pokeapi.co/api/v2/stat/5/"
    },
    {
        "base_stat": 125,
        "effort": 3,
        "stat": {
            "name": "special-attack",
            "url": "https://pokeapi.co/api/v2/stat/4/"
        }
    },
    {
        "base_stat": 85,
        "effort": 0,
        "stat": {
            "name": "defense",
            "url": "https://pokeapi.co/api/v2/stat/3/"
        }
    },
    {
        "base_stat": 90,
        "effort": 0,
        "stat": {
            "name": "attack",
            "url": "https://pokeapi.co/api/v2/stat/2/"
        }
    },
    {
        "base_stat": 90,
        "effort": 0,
        "stat": {
            "name": "hp",
            "url": "https://pokeapi.co/api/v2/stat/1/"
        }
    }
],
"types": [
    {
        "slot": 2,
        "type": {
            "name": "flying",
            "url": "https://pokeapi.co/api/v2/type/3/"
        }
    },
    {
        "slot": 1,
        "type": {
            "name": "electric",
            "url": "https://pokeapi.co/api/v2/type/13/"
        }
    }
]

```

```
    },  
    ],  
    "weight": 526  
}
```

3.2. Les classes Java

Nous allons donc créer un **record** Java qui reprend cette structure, mais en ne conservant que les champs qui nous intéressent.

fr.univlille.miage.alom.pokemon_type_api.PokemonType.java

```
1 package fr.univlille.miage.alom.pokemon_type_api;  
2  
3 record PokemonType(  
4     int id,  
5     String name,  
6     Sprites sprites,  
7     List<String> types  
8 ) {}
```

fr.univlille.miage.alom.pokemon_type_api.Sprites.java

```
1 package fr.univlille.miage.alom.pokemon_type_api;  
2  
3 record Sprites(String back_default, String front_default) {}
```

4. Le repository "stub"

4.1. Le fichier de données pokemon.

Récupérez le fichier [pokemons.json](#) et enregistrez le dans le répertoire **src/main/resources** de votre projet.

4.2. Le test unitaire

Implémentez le test unitaire suivant :

src/test/java/fr/univlille/miage/alom/pokemon_type_api/PokemonRepositoryImplTest.java

```
1 package fr.univlille.miage.alom.pokemon_type_api;  
2  
3 import org.junit.jupiter.api.Test;  
4  
5 import static org.junit.jupiter.api.Assertions.*;  
6
```

```

7 class PokemonTypeRepositoryImplTest {
8
9     private PokemonTypeRepositoryImpl repository = new PokemonTypeRepositoryImpl();
10
11     @Test
12     void findPokemonTypeById_with25_shouldReturnPikachu(){
13         var pikachu = repository.findPokemonTypeById(25);
14         assertNotNull(pikachu);
15         assertEquals("pikachu", pikachu.name());
16         assertEquals(25, pikachu.id());
17     }
18
19     @Test
20     void findPokemonTypeById_with145_shouldReturnZapdos(){
21         var zapdos = repository.findPokemonTypeById(145);
22         assertNotNull(zapdos);
23         assertEquals("zapdos", zapdos.name());
24         assertEquals(145, zapdos.id());
25     }
26
27     @Test
28     void findPokemonTypeByName_withEevee_shouldReturnEevee(){
29         var eevee = repository.findPokemonTypeByName("eevee");
30         assertNotNull(eevee);
31         assertEquals("eevee", eevee.name());
32         assertEquals(133, eevee.id());
33     }
34
35     @Test
36     void findPokemonTypeByName_withMewTwo_shouldReturnMewTwo(){
37         var mewtwo = repository.findPokemonTypeByName("mewtwo");
38         assertNotNull(mewtwo);
39         assertEquals("mewtwo", mewtwo.name());
40         assertEquals(150, mewtwo.id());
41     }
42
43     @Test
44     void findAllPokemonTypes_shouldReturn151Pokemons(){
45         var pokemons = repository.findAllPokemonTypes();
46         assertNotNull(pokemons);
47         assertEquals(151, pokemons.size());
48     }
49
50 }

```

4.3. L'implémentation

Ajouter l'interface du PokemonTypeRepository et son implémentation

src/main/java/fr/univlille/miage/alom/pokemon_type_api/PokemonTypeRepository.java

```
1 interface PokemonTypeRepository {
2     PokemonType findPokemonTypeById(int id);
3     PokemonType findPokemonTypeByName(String name);
4     List<PokemonType> findAllPokemonTypes();
5 }
```

src/main/java/fr/univlille/miage/alom/pokemon_type_api/PokemonTypeRepositoryImpl.java

```
1 class PokemonTypeRepositoryImpl implements PokemonTypeRepository {
2
3     private List<PokemonType> pokemons;
4
5     PokemonTypeRepositoryImpl() {
6         try {
7             var pokemonsStream = this.getClass().getResourceAsStream
8             ("/pokemons.json"); ①
9
10            var objectMapper = new ObjectMapper(); ②
11            objectMapper.configure(DeserializationFeature
12            .FAIL_ON_UNKNOWN_PROPERTIES, false);
13            var pokemonsArray = objectMapper.readValue(pokemonsStream,
14            PokemonType[].class);
15            this.pokemons = Arrays.asList(pokemonsArray);
16        } catch (IOException e) {
17            e.printStackTrace();
18        }
19
20        @Override
21        public PokemonType findPokemonTypeById(int id) {
22            System.out.println("Loading Pokemon information for Pokemon id " + id);
23
24            // TODO ③
25        }
26
27        @Override
28        public PokemonType findPokemonTypeByName(String name) {
29            System.out.println("Loading Pokemon information for Pokemon name " + name);
30
31            // TODO ③
32        }
33
34        @Override
35        public List<PokemonType> findAllPokemonTypes() {
36            // TODO ③
37        }
38    }
```

- ① On charge le fichier json depuis le classpath (maven ajoute le répertoire `src/main/resources` au classpath java !)
- ② On utilise l'ObjectMapper de `jackson-databind` pour transformer les objets JSON en objets JAVA
- ③ On a un peu de code à compléter !

4.4. Ajout de tests unitaires avec Spring

Modifiez le test unitaire de votre repository pour ajouter des éléments liés à Spring

PokemonRepositoryImplTest.java

```
1 @Test
2 void applicationContext_shouldLoadPokemonRepository(){
3     ①
4     var context = new AnnotationConfigApplicationContext
5     ("fr.univlille.miage.alom.pokemon_type_api");
6     var repoByName = context.getBean("pokemonTypeRepositoryImpl"); ②
7     var repoByClass = context.getBean(PokemonTypeRepository.class); ③
8     assertEquals(repoByName, repoByClass);
9     assertNotNull(repoByName);
10    assertNotNull(repoByClass);
11 }
```

- ① Ici, on instancie un `ApplicationContext` Spring, qui est capable d'analyser les annotations Java on lui donne le nom du package Java que l'on souhaite analyser !
- ② Une fois le context instancié, on lui demande de récupérer le repository en utilisant le nom du bean (par défaut le nom de la classe en CamelCase)
- ③ ou en utilisant directement une classe assignable pour notre objet (ici l'interface !)

Pour que Spring arrive à trouver notre classe de repository, il faut poser une annotation dessus !

PokemonTypeRepositoryImpl.java

```
1 @Repository
2 class PokemonTypeRepositoryImpl implements PokemonTypeRepository {
3     [...]
4 }
```



Cette phase doit bien être terminée avant de passer à la suite !

5. Le service

Maintenant que nous avons un repository fonctionnel, il est temps de développer un service qui consomme notre repository !

5.1. Le test unitaire

src/test/java/fr/univlille/miage/alom/pokemon_type_api/PokemonTypeServiceImplTest.java

```
1 package fr.univlille.miage.alom.pokemon_type_api;
2
3 import fr.univlille.miage.alom.pokemon_type_api.PokemonTypeRepository;
4
5 import org.junit.jupiter.api.Test;
6
7 import static org.mockito.Mockito.mock;
8 import static org.mockito.Mockito.verify;
9
10 class PokemonTypeServiceImplTest {
11
12     @Test
13     void pokemonTypeRepository_shouldBeCalled_whenFindById(){
14         var pokemonTypeRepository = mock(PokemonTypeRepository.class); ①
15         var pokemonTypeService = new PokemonTypeServiceImpl(pokemonTypeRepository);
16         ②
17         pokemonTypeService.getPokemonType(25);
18
19         verify(pokemonTypeRepository).findPokemonTypeById(25);
20     }
21
22     @Test
23     void pokemonTypeRepository_shouldBeCalled_whenFindAll(){
24         var pokemonTypeRepository = mock(PokemonTypeRepository.class); ①
25         var pokemonTypeService = new PokemonTypeServiceImpl(pokemonTypeRepository);
26         ②
27         pokemonTypeService.getAllPokemonTypes();
28
29         verify(pokemonTypeRepository).findAllPokemonTypes();
30     }
31
32 }
```

① On crée un mock du PokemonTypeRepository

② et on l'*injecte* via le constructeur !

5.2. L'implémentation

L'interface Java

src/main/java/fr/univlille/miage/alom/pokemon_type_api/PokemonTypeService.java

```
1 public interface PokemonTypeService {
```

```

2     PokemonType getPokemonType(int id);
3     List<PokemonType> getAllPokemonTypes();
4 }

```

et son implémentation

src/main/java/fr/univlille/miage/alom/pokemon_type_api/PokemonTypeServiceImpl.java

```

1 package fr.univlille.miage.alom.pokemon_type_api;
2
3 import fr.univlille.miage.alom.pokemon_type_api.PokemonType;
4
5 import java.util.List;
6
7 class PokemonTypeServiceImpl implements PokemonTypeService{
8
9     PokemonTypeServiceImpl(){ // TODO ①
10
11     }
12
13     @Override
14     PokemonType getPokemonType(int id) {
15         // TODO ①
16     }
17
18     @Override
19     List<PokemonType> getAllPokemonTypes(){
20         // TODO ①
21     }
22 }

```

① à implémenter !

5.3. Implémentation avec Spring

Ajouter les tests suivants au `PokemonTypeServiceImplTest`.

PokemonTypeServiceImplTest

```

1 @Test
2 void applicationContext_shouldLoadPokemonTypeService(){
3     var context = new AnnotationConfigApplicationContext
4         ("fr.univlille.miage.alom.pokemon_type_api");
5     var serviceByName = context.getBean("pokemonTypeServiceImpl");
6     var serviceByClass = context.getBean(PokemonTypeService.class);
7
8     assertEquals(serviceByName, serviceByClass);
9     assertNotNull(serviceByName);
10    assertNotNull(serviceByClass);
11 }

```

```

11
12 @Test
13 void pokemonTypeRepository_shouldBeAutowired_withSpring(){
14     var context = new AnnotationConfigApplicationContext
15         ("fr.univlille.miage.alom.pokemon_type_api");
16     var service = context.getBean(PokemonTypeServiceImpl.class);
17     assertNotNull(service.pokemonTypeRepository);
18 }

```



Vous aurez également besoin d'importer les assertions de Junit en utilisant `import static org.junit.jupiter.api.Assertions.*`



N'oubliez pas que Spring utilise beaucoup les annotations Java, en voici quelques-unes :

- `@Component`
- `@Service`
- `@Repository`
- `@Autowired`

N'oubliez pas que certaines de ces annotations peuvent être posées sur des classes, sur des méthodes, ou sur des constructeurs !



Imaginez un peu comment on aurait pu utiliser cette mécanique au sein de la `DispatcherServlet` que nous avons écrit la semaine dernière...

6. Le Contrôleur

Implémentons un Contrôleur afin d'exposer nos Pokemon en HTTP/REST/JSON.

6.1. Le test unitaire

Le contrôleur est simple et s'inspire de ce que nous avons fait au TP précédent. Cependant, nous n'aurons plus à gérer les paramètres manuellement via une `Map<String, String>`, mais nous allons utiliser toute la puissance de Spring.

src/test/java/fr/univlille/miage/alom/pokemon_type_api/PokemonTypeControllerTest.java

```

1 package fr.univlille.miage.alom.pokemon_type_api;
2
3 import fr.univlille.miage.alom.pokemon_type_api.PokemonType;
4 import fr.univlille.miage.alom.pokemon_type_api.PokemonTypeService;
5
6 import org.junit.jupiter.api.Test;
7
8 import static org.junit.jupiter.api.Assertions.*;

```

```

9 import static org.mockito.Mockito.*;
10
11 class PokemonTypeControllerTest {
12
13     @Test
14     void getPokemonType_shouldCallTheService(){
15         var service = mock(PokemonTypeService.class);
16         var controller = new PokemonTypeController(service);
17
18         var pikachu = new PokemonType(25, "pikachu", null, List.of());
19         when(service.getPokemonType(25)).thenReturn(pikachu);
20
21         var pokemon = controller.getPokemonTypeFromId(25);
22         assertEquals("pikachu", pokemon.name());
23
24         verify(service).getPokemonType(25);
25     }
26
27     @Test
28     void getAllPokemonTypes_shouldCallTheService(){
29         var service = mock(PokemonTypeService.class);
30         var controller = new PokemonTypeController(service);
31
32         controller.getAllPokemonTypes();
33
34         verify(service).getAllPokemonTypes();
35     }
36
37 }

```

6.2. L'implémentation

Compléter l'implémentation du controller :

src/main/java/fr/univlille/miage/alom/pokemon_type_api/PokemonTypeController.java

```

1 class PokemonTypeController {
2
3     PokemonTypeController() { ①
4     }
5
6     PokemonType getPokemonTypeFromId(int id){
7         // TODO ①
8     }
9
10    List<PokemonType> getAllPokemonTypes() {
11        // TODO ①
12    }
13 }

```

① Implémentez !

6.3. L'instrumentation pour spring-web !

Une fois les tests passés, nous pouvons implementer notre controleur pour Spring web !

6.3.1. Les tests unitaires

Ajoutez les tests unitaires suivants à la classe `PokemonTypeControllerTest`

PokemonTypeControllerTest.java

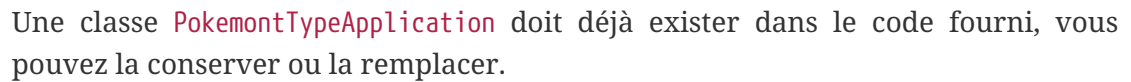
```
1 @Test
2 void pokemonTypeController_shouldBeAnnotated(){
3     var controllerAnnotation =
4         PokemonTypeController.class.getAnnotation(RestController.class);
5     assertNotNull(controllerAnnotation);
6
7     var requestMappingAnnotation =
8         PokemonTypeController.class.getAnnotation(RequestMapping.class);
9     assertEquals(new String[]{"/pokemon-types"}, requestMappingAnnotation
10         .value());
11 }
12 @Test
13 void getPokemonTypeFromId_shouldBeAnnotated() throws NoSuchMethodException {
14     var getPokemonTypeFromId =
15         PokemonTypeController.class.getDeclaredMethod("getPokemonTypeFromId",
16             int.class);
17     var getMapping = getPokemonTypeFromId.getAnnotation(GetMapping.class);
18     assertNotNull(getMapping);
19     assertEquals(new String[]{"/{id}"}, getMapping.value());
20 }
21
22 @Test
23 void getAllPokemonTypes_shouldBeAnnotated() throws NoSuchMethodException {
24     var getAllPokemonTypes =
25         PokemonTypeController.class.getDeclaredMethod("getAllPokemonTypes");
26     var getMapping = getAllPokemonTypes.getAnnotation(GetMapping.class);
27
28     assertNotNull(getMapping);
29     assertEquals(new String[]{"/"}, getMapping.value());
30 }
```

6.3.2. L'implémentation

Posez les bonnes annotations spring pour instrumenter votre Controller et faire passer les tests unitaires.



- Pour exécuter notre projet, nous devons écrire un `main` java !



```
src/main/java/fr/univlille/miage/alom/pokemon_type_api/Application.java
```

② On implémente un `main` pour démarrer notre application !

```

      .  _ _ _ _ _
  /\ /  _ _ _ _ _ ( ) _ _ _ _ _ \ \ \ \ \
 ( ( ) \ _ _ _ _ _ | ' _ | ' _ | | ' _ \ _ ' | \ \ \ \ \
 \ \ /  _ _ _ _ _ | | _ | | | | | | ( _ | | ) ) ) ) ①
  ' | _ _ _ _ _ | . _ _ | | _ _ | | _ \ _ , | / / / /
 =====|_|=====|_ _ _/_/_/_/_/_
 :: Spring Boot ::                (v2.1.2.RELEASE)

```

14

```

found on the java.library.path:
[/usr/java/packages/lib:/usr/lib64:/lib64:/lib:/usr/lib]
[..] [main] o.a.c.c.C.[Tomcat].[localhost].[/]      : Initializing Spring embedded
WebApplicationContext
[..] [main] o.s.web.context.ContextLoader            : Root WebApplicationContext:
initialization completed in 1617 ms
[..] [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService
'applicationTaskExecutor'
[..] [main] o.s.b.w.embedded.tomcat.TomcatWebServer   : Tomcat started on port(s): 8080
(http) with context path ''
[..] [main] c.m.a.pokemon_type_api.Application       : Started Application in 2.72
seconds (JVM running for 3.191)

```

- ① Wao!
- ② On voit que un Tomcat est démarré sur le port 8080

On peut maintenant tester manuellement les URLs suivantes:

- <http://localhost:8080/pokemon-types/>
- <http://localhost:8080/pokemon-types/25>

6.4.1. Plus de logs !

Nous voulons un peu plus de logs pour bien comprendre ce que fait spring-boot.

Pour ce faire, nous allons monter le niveau de logs au niveau **TRACE**.

Créer un fichier **application.properties** dans le répertoire **src/main/resources**.

src/main/resources/application.properties

```

1 # on demande un niveau de logs TRACE a spring-web
2 logging.level.web=TRACE

```

Relancez l'application, vous devriez voir spring logger ceci :

```

[main] s.w.s.m.m.a.RequestMappingHandlerMapping :
    c.m.a.t.p.c.PokemonTypeController: ①
    {GET /pokemon-types/{id}}: getPokemonTypeFromId(int)
    {GET /pokemon-types/}: getAllPokemonTypes()
[main] s.w.s.m.m.a.RequestMappingHandlerMapping :
    o.s.b.a.w.s.e.BasicErrorController: ②
    { /error, produces [text/html]}: errorHtml(HttpServletRequest,HttpServletResponse)
    { /error}: error(HttpServletRequest)

```

- ① On voit que spring a bien pris en compte notre controleur
- ② On voit également que spring a instancié un controleur pour afficher des erreurs sous forme de page HTML

6.4.2. Plus de test

Nous allons également rajouter un dernier test, qui a pour but de :

- démarrer l'application spring en utilisant un port aléatoire
- invoquer dynamiquement notre URL

Ajoutez la dépendance suivante à votre `pom.xml`

pom.xml

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-test</artifactId>
4 </dependency>
```



L'ajout de `spring-boot-starter-test`, depuis la version 2.2.0, ajoute également `junit-jupiter` et `mockito`. Vous pouvez donc supprimer ces dépendances de votre pom.



Ce genre de test, qui démarre une base de données ou un serveur par exemple, est appelé test d'intégration

Implémentez le test unitaire suivant :

fr.univlille.miage.alom.pokemon_type_api.PokemonTypeControllerIntegrationTest

```
1 package fr.univlille.miage.alom.pokemon_type_api;
2
3 import fr.univlille.miage.alom.pokemon_type_api.PokemonType;
4 import org.junit.jupiter.api.Test;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.boot.test.context.SpringBootTest;
7 import org.springframework.boot.test.web.client.TestRestTemplate;
8 import org.springframework.boot.test.web.server.LocalServerPort;
9
10 import static org.junit.jupiter.api.Assertions.*;
11
12 @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT) ①
13 class PokemonTypeControllerIntegrationTest {
14
15     @LocalServerPort ②
16     private int port;
17
18     @Autowired
19     private TestRestTemplate restTemplate; ③
20
21     @Autowired
22     private PokemonTypeController controller; ④
```

```

23
24     @Test
25     void pokemonTypeController_shouldBeInstanciaded(){ ④
26         assertNotNull(controller);
27     }
28
29     @Test
30     void getPokemon_withId25_ShouldReturnPikachu() throws Exception {
31         var url = "http://localhost:" + port + "/pokemon-types/25"; ⑤
32
33         var pikachu = this.restTemplate.getForObject(url, PokemonType.class); ⑥
34
35         assertNotNull(pikachu); ⑦
36         assertEquals(25, pikachu.id());
37         assertEquals("pikachu", pikachu.name());
38     }
39 }

```

- ① On utilise un `SpringBootTest` pour exécuter ce test. Ce test va donc instancier Spring. On précise également que l'environnement Spring doit utiliser un port aléatoire.
- ② On demande à Spring de nous donner le port sur lequel le serveur aura été démarré
- ③ On demande à Spring de nous donner un `TestRestTemplate`, qui nous permettra de jouer une requête HTTP
- ④ On peut faire directement de l'injection de dépendance dans notre test, nous en profitons pour valider que notre controller est bien chargé.
- ⑤ On construit dynamiquement l'url à invoquer
- ⑥ On utilise le `TestRestTemplate` pour appeler notre API ! Le `TestRestemplate` va également se charger de convertir le JSON reçu, en objet Java en utilisant `jackson-databind`.
- ⑦ Enfin, on valide que Pikachu est arrivé en bon état !

7. Autres routes

Implémentez la route qui permet de récupérer un pokemon par son nom.

Elle doit être disponible via ces url de test :

- <http://localhost:8080/pokemon-types/?name=pikachu>
- <http://localhost:8080/pokemon-types/?name=mew>

8. Packager notre micro-service

Une fois notre service fonctionnel, nous pouvons le packager. Notre micro-service sera packagé dans un *jar* exécutable !

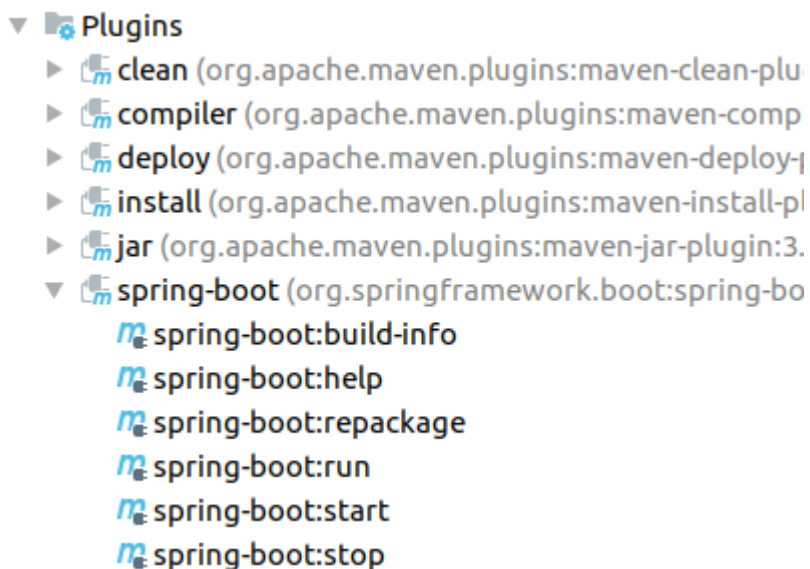
8.1. Le plugin spring-boot maven

Notre `pom.xml` contient le bloc suivant :

pom.xml

```
1 <build>
2   <plugins>
3     <plugin>
4       <groupId>org.springframework.boot</groupId>
5       <artifactId>spring-boot-maven-plugin</artifactId>
6     </plugin>
7   </plugins>
8 </build>
```

Ce plugin nous met à disposition de nouvelles tâches maven !



Nous pouvons lancer notre application en exécutant la commande suivante :

```
mvn spring-boot:run
```

8.2. Packager notre micro-service

Avant de packager notre micro-service, nous devons modifier le chargement des ressources dans `PokemonTypeRepositoryImpl`. La mécanique d'exécution de spring-boot utilise 2 classpath Java, ce qui impose que les fichiers de ressources (en particulier notre fichier JSON), doivent être chargés différemment.

Modifiez le constructeur du repository pour être le suivant :

PokemonTypeRepositoryImpl.java

```
1 PokemonTypeRepositoryImpl() {
```

```

2    try {
3        var pokemonsStream = new ClassPathResource("pokemons.json").
        getInputStream();
4
5        var objectMapper = new ObjectMapper();
6        objectMapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES,
        false);
7        var pokemonsArray = objectMapper.readValue(pokemonsStream, PokemonType
        [].class);
8        this.pokemons = Arrays.asList(pokemonsArray);
9    } catch (IOException e) {
10        e.printStackTrace();
11    }
12 }

```

Pour créer un jar de notre service, il faut maintenant lancer la commande :

```
mvn package
```

Et pour l'exécuter, il suffit alors de lancer :

```
java -jar target/pokemon-type-api-0.1.0.jar
```



La construction de *jar* "autoporté" spring-boot, est aujourd'hui l'état de l'art des approches micro-service !

9. Continuez à développer

Les types de Pokemons sont des données "référentielles". Cela signifie qu'elles seront le plus souvent accédées en lecture seule. Cependant, nous pouvons développer des routes supportant des paramètres supplémentaires pour être capable de rechercher plus finement un pokémon !

Par défaut, la liste des Pokemons doit également être triée par *id*.

Développez les routes suivantes pour notre jeu :

- <http://localhost:8080/pokemon-types?types=electric> (9 Pokemons ont le type électrique)
- <http://localhost:8080/pokemon-types?types=bug,poison> (5 Pokemons ont les types insecte et poison)

Ajoutez les *stats*, *baseExperience*, *weight* et *height*.

Ajoutez des routes permettant de récupérer les PokemonType triés selon différents critères :

- <http://localhost:8080/pokemon-types?orderBy=id>
- <http://localhost:8080/pokemon-types?orderBy=id,desc>

- <http://localhost:8080/pokemon-types?orderBy=stats.attack,desc>
- <http://localhost:8080/pokemon-types?orderBy=stats.speed,asc>