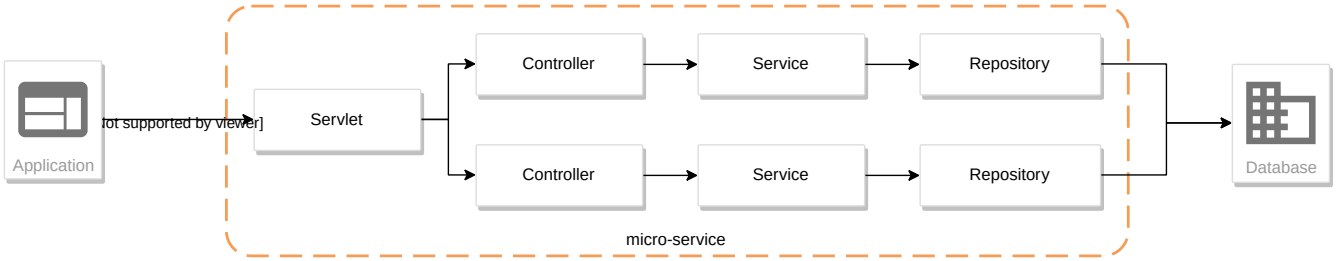
ALOM - TP - Spring-Boot

Table of Contents

1. Présentation et objectifs

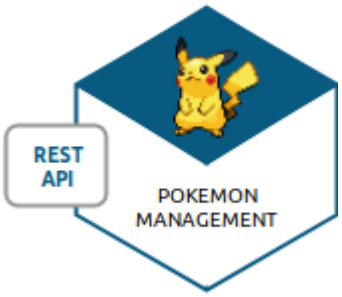


Le but est de reprendre à zéro le développement de notre architecture "à la microservice", en utilisant `spring-boot`

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 ! Nous allons réécrire le TP "Handcrafting" en utilisant Spring !

2. Github

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 !

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/1
45.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.pn
g"
    },
    "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.univ_lille.alom.pokemon_type_api.PokemonType.java

```java
package fr.univ_lille.alom.pokemon_type_api;

record PokemonType(
        int id,
        String name,
        Sprites sprites,
        List<String> types
) {}
```

fr.univ_lille.alom.pokemon_type_api.Sprites.java

```java
package fr.univ_lille.alom.pokemon_type_api;

record Sprites(String back_default, String front_default) {}
```

4. Le repository

4.1. Le fichier de données pokemon.

Récupérez le fichier pokemons.json et enregistrez le dans le répertoire `src/main/resources` de votre projet.

Attention, le fichier pokemons.json a été modifié depuis le dernier TP.

4.2. Le test unitaire

Implémentez le test unitaire suivant :

src/test/java/fr/univ_lille/alom/pokemon_type_api/PokemonRepositoryImplTest.java

```java
package fr.univ_lille.alom.pokemon_type_api;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

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/univ_lille/alom/pokemon_type_api/PokemonTypeRepository.java

```java
interface PokemonTypeRepository {
    PokemonType findPokemonTypeById(int id);
    PokemonType findPokemonTypeByName(String name);
    List<PokemonType> findAllPokemonTypes();
}
```

src/main/java/fr/univ_lille/alom/pokemon_type_api/PokemonTypeRepositoryImpl.java

```java
class PokemonTypeRepositoryImpl implements PokemonTypeRepository {

    private List<PokemonType> pokemons;

    PokemonTypeRepositoryImpl() {
        try {
            var pokemonsStream = this.getClass().getResourceAsStream
("/pokemons.json"); ①

            var objectMapper = new ObjectMapper(); ②
            objectMapper.configure
(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
            var pokemonsArray = objectMapper.readValue(pokemonsStream,
PokemonType[].class);
            this.pokemons = Arrays.asList(pokemonsArray);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public PokemonType findPokemonTypeById(int id) {
        System.out.println("Loading Pokemon information for Pokemon id " + id);

        // TODO ③
    }

    @Override
    public PokemonType findPokemonTypeByName(String name) {
        System.out.println("Loading Pokemon information for Pokemon name " + name);

        // TODO ③
    }

    @Override
    public List<PokemonType> findAllPokemonTypes() {
        // TODO ③
    }
}
```

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("fr.univ_lille.alom.pokemon_type_api");
5     var repoByName = context.getBean("pokemonTypeRepositoryImpl"); ②
6     var repoByClass = context.getBean(PokemonTypeRepository.class); ③
7
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/univ_lille/alom/pokemon_type_api/PokemonTypeServiceImplTest.java

```java
package fr.univ_lille.alom.tp.pokemon_type_api;

import fr.univ_lille.alom.tp.pokemon_type_api.PokemonTypeRepository;
import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.verify;

class PokemonTypeServiceImplTest {

    @Test
    void pokemonTypeRepository_shouldBeCalled_whenFindById(){
        var pokemonTypeRepository = mock(PokemonTypeRepository.class); ①
        var pokemonTypeService = new PokemonTypeServiceImpl(pokemonTypeRepository); ②

        pokemonTypeService.getPokemonType(25);

        verify(pokemonTypeRepository).findPokemonTypeById(25);
    }

    @Test
    void pokemonTypeRepository_shouldBeCalled_whenFindAll(){
        var pokemonTypeRepository = mock(PokemonTypeRepository.class); ①
        var pokemonTypeService = new PokemonTypeServiceImpl(pokemonTypeRepository); ②

        pokemonTypeService.getAllPokemonTypes();

        verify(pokemonTypeRepository).findAllPokemonTypes();
    }

}
```

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/univ_lille/alom/pokemon_type_api/PokemonTypeService.java

```java
public interface PokemonTypeService {
    PokemonType getPokemonType(int id);
```

```
    List<PokemonType> getAllPokemonTypes();
}
```

et son implémentation

src/main/java/fr/univ_lille/alom/pokemon_type_api/PokemonTypeServiceImpl.java

```
package fr.univ_lille.alom.tp.pokemon_type_api.service;

import fr.univ_lille.alom.tp.pokemon_type_api.bo.PokemonType;

import java.util.List;

class PokemonTypeServiceImpl implements PokemonTypeService{

    PokemonTypeServiceImpl(){ // TODO ①

    }

    @Override
    PokemonType getPokemonType(int id) {
        // TODO ①
    }

    @Override
    List<PokemonType> getAllPokemonTypes(){
        // TODO ①
    }
}
```

① à implémenter !

5.3. Implémentation avec Spring

Ajouter les tests suivants au `PokemonTypeServiceImplTest`.

PokemonTypeServiceImplTest

```
@Test
void applicationContext_shouldLoadPokemonTypeService(){
    var context = new
AnnotationConfigApplicationContext("fr.univ_lille.alom.tp.pokemon_type_api");
    var serviceByName = context.getBean("pokemonTypeServiceImpl");
    var serviceByClass = context.getBean(PokemonTypeService.class);

    assertEquals(serviceByName, serviceByClass);
    assertNotNull(serviceByName);
    assertNotNull(serviceByClass);
}

```

```
@Test
void pokemonTypeRepository_shouldBeAutowired_withSpring(){
    var context = new
AnnotationConfigApplicationContext("fr.univ_lille.alom.tp.pokemon_type_api");
    var service = context.getBean(PokemonTypeServiceImpl.class);
    assertNotNull(service.pokemonTypeRepository);
}
```

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 Controlleur

Implémentons un Controlleur afin d'exposer nos Pokemons en HTTP/REST/JSON.

6.1. Le test unitaire

Le controlleur 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/univ_lille/alom/pokemon_type_api/PokemonTypeControllerTest.java

```
package fr.univ_lille.alom.tp.pokemon_type_api;

import fr.univ_lille.alom.tp.pokemon_type_api.PokemonType;
import fr.univ_lille.alom.tp.pokemon_type_api.PokemonTypeService;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

```

```java
class PokemonTypeControllerTest {

    @Test
    void getPokemonType_shouldCallTheService(){
        var service = mock(PokemonTypeService.class);
        var controller = new PokemonTypeController(service);

        var pikachu = new PokemonType(25, "pikachu", null, List.of());
        when(service.getPokemonType(25)).thenReturn(pikachu);

        var pokemon = controller.getPokemonTypeFromId(25);
        assertEquals("pikachu", pokemon.name());

        verify(service).getPokemonType(25);
    }

    @Test
    void getAllPokemonTypes_shouldCallTheService(){
        var service = mock(PokemonTypeService.class);
        var controller = new PokemonTypeController(service);

        controller.getAllPokemonTypes();

        verify(service).getAllPokemonTypes();
    }

}
```

6.2. L'implémentation

Compléter l'implémentation du controller :

src/main/java/fr/univ_lille/alom/pokemon_type_api/PokemonTypeController.java

```java
class PokemonTypeController {

    PokemonTypeController() { ①
    }

    PokemonType getPokemonTypeFromId(int id){
        // TODO ①
    }

    List<PokemonType> getAllPokemonTypes() {
        // TODO ①
    }
}
```

① Implémentez !

6.3. L'instrumentation pour spring-web !

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

6.3.1. Les tests unitaires

Ajoutez les tests unitaires suivants à la classe `PokemonTypeControllerTest`

PokemonTypeControllerTest.java

```
@Test
void pokemonTypeController_shouldBeAnnotated(){
    var controllerAnnotation =
            PokemonTypeController.class.getAnnotation(RestController.class);
    assertNotNull(controllerAnnotation);

    var requestMappingAnnotation =
            PokemonTypeController.class.getAnnotation(RequestMapping.class);
    assertArrayEquals(new String[]{"/pokemon-types"},
requestMappingAnnotation.value());
}

@Test
void getPokemonTypeFromId_shouldBeAnnotated() throws NoSuchMethodException {
    var getPokemonTypeFromId =
            PokemonTypeController.class.getDeclaredMethod("getPokemonTypeFromId",
int.class);
    var getMapping = getPokemonTypeFromId.getAnnotation(GetMapping.class);

    assertNotNull(getMapping);
    assertArrayEquals(new String[]{"/{id}"}, getMapping.value());
}

@Test
void getAllPokemonTypes_shouldBeAnnotated() throws NoSuchMethodException {
    var getAllPokemonTypes =
            PokemonTypeController.class.getDeclaredMethod("getAllPokemonTypes");
    var getMapping = getAllPokemonTypes.getAnnotation(GetMapping.class);

    assertNotNull(getMapping);
    assertArrayEquals(new String[]{"/"}, getMapping.value());
}
```

6.3.2. L'implémentation

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

Pour vous aider, voici des liens vers la documentation de spring-web :

- @RequestMapping

6.4. L'exécution de notre projet !

Pour exécuter notre projet, nous devons écrire un main java ! Implémentez la classe suivante :

src/main/java/fr/univ_lille/alom/pokemon_type_api/Application.java

```
1 @SpringBootApplication ①
2 public class Application {
3
4     public static void main(String... args){
5         SpringApplication.run(Application.class, args); ②
6     }
7 }
```

① On annote la classe comme étant le point d'entrée de notre application

② On implémente un main pour démarrer notre application !

Démarrez le main, et observez les logs :

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

[..] [main] c.m.a.tp.pokemon_type_api.Application    : Starting Application on
jwittouck-N14xWU with PID 12414 (/home/jwittouck/workspaces/alom/alom-2020-
2021/tp/pokemon-type-api/target/classes started by jwittouck in
/home/jwittouck/workspaces/alom/alom-2021-2022)
[..] [main] c.m.a.tp.pokemon_type_api.Application    : No active profile set, falling
back to default profiles: default
[..]  INFO 12414 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat
initialized with port(s): 8080 (http)
[..] [main] o.apache.catalina.core.StandardService   : Starting service [Tomcat] ②
[..] [main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine:
[Apache Tomcat/9.0.14]
[..] [main] o.a.catalina.core.AprLifecycleListener   : The APR based Apache Tomcat
Native library which allows optimal performance in production environments was not
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.tp.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 logguer 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 controlleur

② On voit également que spring a instancié un controlleur 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

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
</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.univ_lille.alom.pokemon_type_api.PokemonTypeControllerIntegrationTest

```java
package fr.univ_lille.alom.pokemon_type_api;

import fr.univ_lille.alom.pokemon_type_api.PokemonType;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.boot.test.web.server.LocalServerPort;

import static org.junit.jupiter.api.Assertions.*;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT) ①
class PokemonTypeControllerIntegrationTest {

    @LocalServerPort ②
    private int port;

    @Autowired
    private TestRestTemplate restTemplate; ③

    @Autowired
    private PokemonTypeController controller; ④

    @Test
    void pokemonTypeController_shouldBeInstanciated(){ ④
        assertNotNull(controller);
    }

    @Test
    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

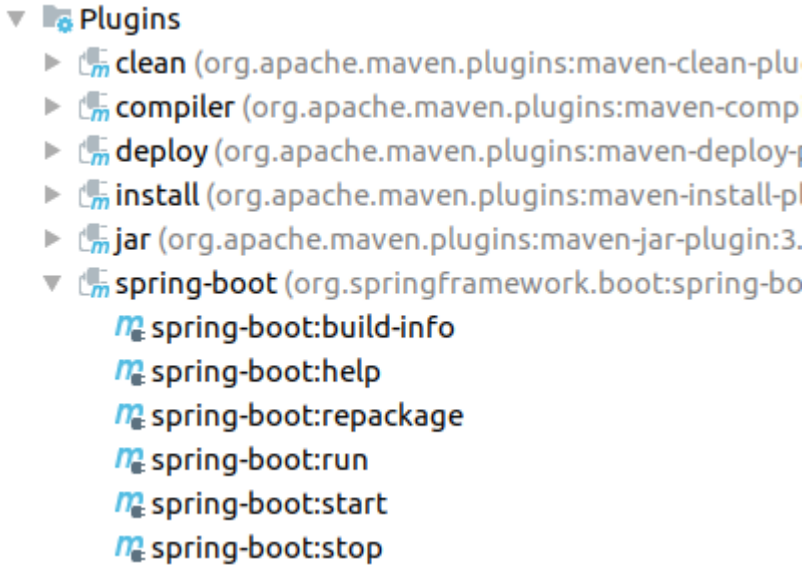
Notre `pom.xml` contient le bloc suivant :

pom.xml

```
1 <build>
2     <plugins>
3         <plugin>
```

```
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</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 classpaths 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

```
PokemonTypeRepositoryImpl() {
    try {
        var pokemonsStream = new ClassPathResource("pokemons.json").
getInputStream();

        var objectMapper = new ObjectMapper();
        objectMapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES,
false);
        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 contruction 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 recherche plus finement un pokémon !

Par défaut, la liste des pokémons doit également être triée par `id`.

Développez les routes suivantes pour notre jeu :

- http://localhost:8080/pokemon-types?types=electric (9 pokémons ont le type électrique)
- http://localhost:8080/pokemon-types?types=bug,poison (5 pokémons 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