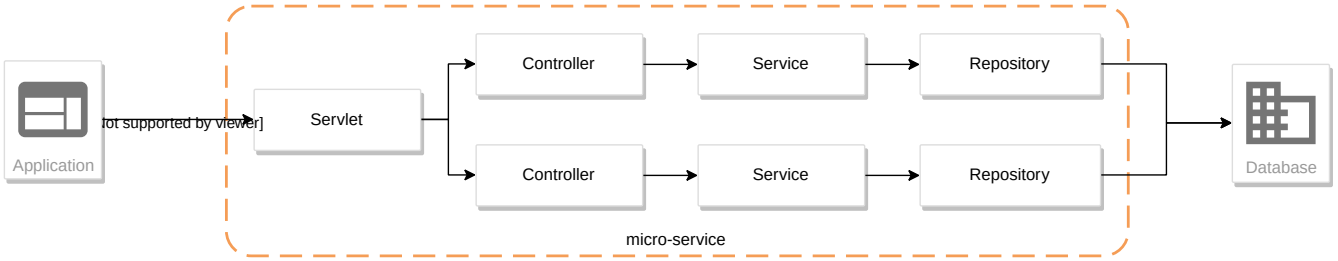
ALOM - TP 4 - JPA & Repositories

Table of Contents

1. Présentation et objectifs



Le but est de continuer le développement de notre architecture "à la microservice".

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 dresseurs de Pokemon !

Nous allons développer :

1. un repository d'accès aux données de Trainers (à partir d'une base de données)
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. charger quelques données

Nous repartons de zéro pour ce TP !

2. GitLab

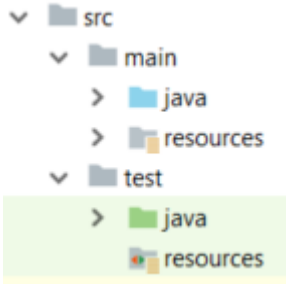
Identifiez vous sur GitLab, et cliquez sur le lien suivant pour créer votre repository git: GitLab classroom

Clonez ensuite votre repository git sur votre poste !

A partir de ce TP, votre repository nouvellement créé contiendra un squelette de projet contenant:

- un fichier `pom.xml` basique
- l'arborescence projet:
  - `src/main/java`
  - `src/main/resources`

- src/test/java
- src/test/resources



3. Le pom.xml

Modifiez le fichier pom.xml à la racine du projet

```xml
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>fr.univ-lille.alom</groupId>
    <artifactId>trainer-api</artifactId> ①
    <version>0.1.0</version>
    <packaging>jar</packaging> ②

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.1.4</version> ②
    </parent>

    <properties>
        <java.version>17</java.version> ③
    </properties>

    <dependencies>

        <!-- spring-boot web-->
        <dependency>
            <groupId>org.springframework.boot</groupId> ②
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <!-- testing --> ④
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
        </dependency>

    </dependencies>

```

```
34      <build> ⑤
35          <plugins>
36              <plugin>
37                  <groupId>org.springframework.boot</groupId>
38                  <artifactId>spring-boot-maven-plugin</artifactId>
39              </plugin>
40          </plugins>
41      </build>
42
43  </project>
```

① Modifiez votre `artifactId`

② Cette fois, on utilise directement `spring-boot` pour construire un `jar`

③ en java 17…

④ On positionne spring-boot-starter-test qui nous importe JUnit et Mockito !

⑤ La partie build utilise le `spring-boot-maven-plugin`

Notre projet est prêt !

4. Le repository

Lors du TP précédent, nous avions écrit un repository qui utilisait un fichier `JSON` comme source de données.

Cette semaine, nous utiliserons directement une base de données, embarquée dans un premier temps.

Nous commençons les développements avec une base de données embarquée, puis nous testerons ensuite une base de données managée sur un cloud public.

Cette base de données est H2. H2 est écrit en Java, implémente le standard SQL, et peut fonctionner directement en mémoire !

4.1. L'ajout de la dépendance spring-boot-data-jpa et H2

Ajoutez les dépendance suivantes dans votre `pom.xml`

- spring-boot-starter-data-jpa
- h2 (en scope test)

4.2. Les business objects

Nous allons manipuler, dans ce microservice, des dresseurs de Pokemon (Trainer), ainsi que leur équipe de Pokemons préférée (id de pokémon type + niveau).

Nous allons donc commencer par écrire deux classes Java pour représenter nos données : `Trainer` et `Pokemon`

src/main/java/fr/univ_lille/alom/trainers/Trainer.java

```
1 // TODO
2 public class Trainer { ①
3
4     private String name; ②
5
6     private List<Pokemon> team; ③
7
8     public Trainer() {
9     }
10
11    public Trainer(String name) {
12        this.name = name;
13    }
14
15    [...] ④
16 }
```

① Notre classe de dresseur de Pokemon

② Son nom (qui servira d'identifiant en base de données :))

③ La liste de ses pokemons

④ Les getters/setters habituels (à générer avec `Alt` + `Inser` !)

Nous ne pouvons pas utiliser les `record` de Java pour représenter les Trainers/Pokemon. Les *Entity* JPA doivent:

- être des classes non `final`
- avoir un constructeur `public` sans argument
- les attributs doivent être non `final`

Les records ne respectent pas ces conditions, et donc on ne peut pas les utiliser pour le moment .

src/main/java/fr/univ_lille/alom/trainers/Pokemon.java

```
1 // TODO
2 public class Pokemon {
3
4     private int pokemonTypeId; ①
5
6     private int level; ②
7
8     public Pokemon() {
9     }
```

```
10
11      public Pokemon(int pokemonTypeId, int level) {
12          this.pokemonTypeId = pokemonTypeId;
13          this.level = level;
14      }
15
16      [...] ④
17  }
```

① le numéro de notre Pokemon dans le Pokedex (référence au service pokemon-type-api !)

② le niveau de notre Pokemon !

4.3. Les test unitaires

Implémentez les tests unitaires suivant :

src/test/java/fr/univ_lille/alom/trainers/TrainerTest.java

```
1  package fr.univ_lille.alom.trainers.bo;
2
3  import org.junit.jupiter.api.Test;
4
5  import jakarta.persistence.*;
6
7  import static org.junit.jupiter.api.Assertions.*;
8
9  class TrainerTest {
10
11      @Test
12      void trainer_shouldBeAnEntity(){
13          assertNotNull(Trainer.class.getAnnotation(Entity.class)); ①
14      }
15
16      @Test
17      void trainerName_shouldBeAnId() throws NoSuchFieldException {
18          assertNotNull(Trainer.class.getDeclaredField("name").getAnnotation(Id.
    class)); ②
19      }
20
21      @Test
22      void trainerTeam_shouldBeAElementCollection() throws NoSuchFieldException {
23          assertNotNull(Trainer.class.getDeclaredField("team").
    getAnnotation(ElementCollection.class)); ③
24      }
25
26  }
```

① Notre classe `Trainer` doit être annotée `@Entity` pour être reconnue par JPA

② Chaque classe annotée `@Entity` doit déclarer un de ses champs comme étant un `@Id`. Dans le cas

du `Trainer`, le champ `name` est idéal

③ La relation entre `Trainer` et `Pokemon` doit également être annotée. Ici, un `Trainer` possède une collection de `Pokemon`.

src/test/java/fr/univ_lille/alom/trainers/PokemonTest.java

```
1 class PokemonTest {
2
3     @Test
4     void pokemon_shouldBeAnEmbeddable(){
5         assertNotNull(Pokemon.class.getAnnotation(Embeddable.class)); ①
6     }
7
8 }
```

① Notre classe `Pokemon` doit aussi être annotée `@Embeddable` pour être reconnue par JPA

src/test/java/fr/univ_lille/alom/trainers/TrainerRepositoryTest.java

```
1  package fr.univ_lille.alom.trainers;
2
3  import [...]
4
5  import static org.junit.jupiter.api.Assertions.*;
6
7  @DataJpaTest ①
8  class TrainerRepositoryTest {
9
10     @Autowired ②
11     private TrainerRepository repository;
12
13     @Test
14     void trainerRepository_shouldExtendsCrudRepository() throws
   NoSuchMethodException {
15         assertTrue(CrudRepository.class.isAssignableFrom(TrainerRepository.class));
   ③
16     }
17
18     @Test
19     void trainerRepositoryShouldBeInstanciedBySpring(){
20         assertNotNull(repository);
21     }
22
23     @Test
24     void testSave(){ ④
25         var ash = new Trainer("Ash");
26
27         repository.save(ash);
28
29         var saved = repository.findById(ash.getName()).orElse(null);
```

```
30
31          assertEquals("Ash", saved.getName());
32      }
33
34      @Test
35      void testSaveWithPokemons(){ ⑤
36          var misty = new Trainer("Misty");
37          var staryu = new Pokemon(120, 18);
38          var starmie = new Pokemon(121, 21);
39          misty.setTeam(List.of(staryu, starmie));
40
41          repository.save(misty);
42
43          var saved = repository.findById(misty.getName()).orElse(null);
44
45          assertEquals("Misty", saved.getName());
46          assertEquals(2, saved.getTeam().size());
47      }
48
49  }
```

① On utilise un `@DataJpaTest` test, qui va démarrer spring (uniquement la partie gestion des repositories et base de données).

② On utilise l'injection de dépendances spring dans notre test !

③ On valide que notre repository hérite du `CrudRepository` proposé par spring.

④ On test la sauvegarde simple

⑤ et la sauvegarde avec des objets en cascade !

Ce type de test, appelé test d'intégration, a pour but de valider que l'application se contruit bien. Le démarrage de spring étant plus long que le simple couple JUnit/Mockito, on utilise souvent ces tests uniquement sur la partie repository

Notre test sera exécuté avec une instance de base de données H2 instanciée à la volée !

4.4. L'exécution de notre test

Pour s'exécuter, notre test unitaire a besoin d'une application Spring-Boot !

Vérifiez que vous avez bien une classe `TrainerApiApplication.java`, sinon créez la :

src/main/java/fr/univ_lille/alom/trainers/TrainerApiApplication.java

```
1 @SpringBootApplication ①
2 public class TrainerApiApplication {
3
4     public static void main(String... args){ ②
```

```
5         SpringApplication.run(TrainerApiApplication.class, args);
6     }
7
8 }
```

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

② On implémente un main pour démarrer notre application !

4.5. L'implémentation

Ajouter l'interface du TrainerRepository !

src/main/java/fr/univ_lille/alom/trainers/TrainerRepository.java

```
1 // TODO
2 public interface TrainerRepository {
3 }
```

Attention, ici, nous ne développerons pas l'implémentation du repository ! C'est *Spring* qui se chargera de nous en créer une instance à l'exécution !

Pour vous aider, voici deux liens intéressants :

- La documentation officielle de spring-data : https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories
- Et un tutoriel officiel : https://spring.io/guides/gs/accessing-data-jpa/

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/trainers/TrainerServiceImplTest.java

```
1 class TrainerServiceImplTest {
2
3     @Test
4     void getAllTrainers_shouldCallTheRepository() {
5         var trainerRepo = mock(TrainerRepository.class);
6         var trainerService = new TrainerServiceImpl(trainerRepo);
7
8         trainerService.getAllTrainers();
9
10        verify(trainerRepo).findAll();
```

```
11      }
12
13      @Test
14      void getTrainer_shouldCallTheRepository() {
15          var trainerRepo = mock(TrainerRepository.class);
16          var trainerService = new TrainerServiceImpl(trainerRepo);
17
18          trainerService.getTrainer("Ash");
19
20          verify(trainerRepo).findById("Ash");
21      }
22
23      @Test
24      void createTrainer_shouldCallTheRepository() {
25          var trainerRepo = mock(TrainerRepository.class);
26          var trainerService = new TrainerServiceImpl(trainerRepo);
27
28          var ash = new Trainer();
29          trainerService.createTrainer(ash);
30
31          verify(trainerRepo).save(ash);
32      }
33
34  }
```

5.2. L'implémentation

L'interface Java

src/main/java/fr/univ_lille/alom/trainers/TrainerService.java

```
1  public interface TrainerService {
2
3      Iterable<Trainer> getAllTrainers();
4      Trainer getTrainer(String name);
5      Trainer createTrainer(Trainer trainer);
6  }
```

et son implémentation

src/main/java/fr/univ_lille/alom/trainers/TrainerServiceImpl.java

```
1  // TODO
2  public class TrainerServiceImpl implements TrainerService { ①
3
4      private TrainerRepository trainerRepository;
5
6      public TrainerServiceImpl(TrainerRepository trainerRepository) {
7          this.trainerRepository = trainerRepository;
```

```
    }

    @Override
    public Iterable<Trainer> getAllTrainers() {
        // TODO
    }

    @Override
    public Trainer getTrainer(String name) {
        // TODO
    }

    @Override
    public Trainer createTrainer(Trainer trainer) {
        // TODO
    }
}
```

① à implémenter !

Comme nous n'avons pas la main sur l'implémentation du repository (spring le crée dynamiquement), l'utilisation de l'injection de dépendances devient primordiale !

6. Le controlleur

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

6.1. Le test unitaire

Le controlleur est simple et s'inpire de ce que nous avons fait au TP précédent.

src/test/java/fr/univ_lille/alom/trainers/TrainerControllerTest.java

```
class TrainerControllerTest {

    @Mock
    private TrainerService trainerService;

    @InjectMocks
    private TrainerController trainerController;

    @BeforeEach
    void setup(){
        MockitoAnnotations.initMocks(this);
    }

    @Test
    void getAllTrainers_shouldCallTheService() {
```

```
        trainerController.getAllTrainers();

        verify(trainerService).getAllTrainers();
    }

    @Test
    void getTrainer_shouldCallTheService() {
        trainerController.getTrainer("Ash");

        verify(trainerService).getTrainer("Ash");
    }
}
```

6.2. L'implémentation

Compléter l'implémentation du controller :

src/main/java/fr/univ_lille/alom/trainers/TrainerController.java

```
public class TrainerController {

    private final TrainerService trainerService;

    TrainerController(TrainerService trainerService){
        this.trainerService = trainerService;
    }

    Iterable<Trainer> getAllTrainers(){
        // TODO ①
    }

    Trainer getTrainer(String name){
        // TODO ①
    }

}
```

① Implémentez !

6.3. L'ajout des annotations Spring

Ajoutez les méthodes de test suivantes dans la classe `TrainerControllerTest` :

TrainerControllerTest.java

```
@Test
void trainerController_shouldBeAnnotated(){
    var controllerAnnotation =
            TrainerController.class.getAnnotation(RestController.class);
```

```
5     assertNotNull(controllerAnnotation);
6
7     var requestMappingAnnotation =
8             TrainerController.class.getAnnotation(RequestMapping.class);
9     assertArrayEquals(new String[]{"/trainers"}, requestMappingAnnotation.value());
10 }
11
12 @Test
13 void getAllTrainers_shouldBeAnnotated() throws NoSuchMethodException {
14     var getAllTrainers =
15             TrainerController.class.getDeclaredMethod("getAllTrainers");
16     var getMapping = getAllTrainers.getAnnotation(GetMapping.class);
17
18     assertNotNull(getMapping);
19     assertArrayEquals(new String[]{"/"}, getMapping.value());
20 }
21
22 @Test
23 void getTrainer_shouldBeAnnotated() throws NoSuchMethodException {
24     var getTrainer =
25             TrainerController.class.getDeclaredMethod("getTrainer", String.class);
26     var getMapping = getTrainer.getAnnotation(GetMapping.class);
27
28     var pathVariableAnnotation = getTrainer.getParameters()[0].
   getAnnotation(PathVariable.class);
29
30     assertNotNull(getMapping);
31     assertArrayEquals(new String[]{"/{name}"}, getMapping.value());
32
33     assertNotNull(pathVariableAnnotation);
34 }
```

Modifiez votre classe `TrainerController` pour faire passer les tests !

6.4. L'exécution de notre projet !

Pour exécuter notre projet, nous devons simplement lancer la classe `TrainerApiApplication` écrite plus haut.

Mais avant cela, modifions quelques propriétés de spring !

6.4.1. Personnalisation de Spring-Boot

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

```
# on demande un niveau de logs TRACE a spring-web
logging.level.web=TRACE
# on modifie le port par defaut du tomcat !
server.port=8081
```

Le répertoire `src/main/resources` est ajouté au classpath Java par IntelliJ, lors de l'exécution, et par Maven lors de la construction de notre jar !

La liste des properties supportées est décrite dans la documentation de spring ici

6.4.2. Ajout de données au démarrage

Comme notre application ne contient aucune donnée au démarrage, nous allons en charger quelques-unes "en dur" pour commencer.

Ajoutez le code suivant dans la classe `TrainerApiApplication` :

src/main/java/fr/univ_lille/alom/trainers/TrainerApiApplication.java

```
@Bean ②
@Autowired ③
public CommandLineRunner demo(TrainerRepository repository) { ①
    return (args) -> { ④
        var ash = new Trainer("Ash");
        var pikachu = new Pokemon(25, 18);
        ash.setTeam(List.of(pikachu));

        var misty = new Trainer("Misty");
        var staryu = new Pokemon(120, 18);
        var starmie = new Pokemon(121, 21);
        misty.setTeam(List.of(staryu, starmie));

        // save a couple of trainers
        repository.save(ash); ⑤
        repository.save(misty);
    };
}
```

① On implémente un CommandLineRunner pour exécuter des commandes au démarrage de notre application

② On utilise l'annotation @Bean sur notre méthode, pour en déclarer le retour comme étant un bean spring !

③ On utilise l'injection de dépendance sur notre méthode !

④ CommandLineRunner est une @FunctionnalInterface, on en fait une expression lambda.

⑤ On initialise quelques données !

6.4.3. Exécution

Démarrez le main, et observez les logs (j'ai réduit la quantité de logs pour qu'elle s'affiche correctement ici) :

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

[main] [..] : Starting TrainerApi on jwittouck-N14xWU with PID 23154
(/home/jwittouck/workspaces/alom/alom-2020-2021/tp/trainer-api/target/classes started
by jwittouck in /home/jwittouck/workspaces/alom/alom-2020-2021)
[main] [..] : No active profile set, falling back to default profiles: default
[main] [..] : Bootstrapping Spring Data repositories in DEFAULT mode.
[main] [..] : Finished Spring Data repository scanning in 47ms. Found 1 repository
interfaces.
[main] [..] : Bean
'org.springframework.transaction.annotation.ProxyTransactionManagementConfiguration'
of type
[org.springframework.transaction.annotation.ProxyTransactionManagementConfiguration$$E
nhancerBySpringCGLIB$$ff9e9081] is not eligible for getting processed by all
BeanPostProcessors (for example: not eligible for auto-proxying)
[main] [..] : Tomcat initialized with port(s): 8081 (http) ②
[main] [..] : Starting service [Tomcat] ②
[main] [..] : Starting Servlet engine: [Apache Tomcat/9.0.14]
[main] [..] : 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] [..] : Initializing Spring embedded WebApplicationContext
[main] [..] : Published root WebApplicationContext as ServletContext attribute with
name [org.springframework.web.context.WebApplicationContext.ROOT]
[main] [..] : Root WebApplicationContext: initialization completed in 1487 ms
[main] [..] : Added existing Servlet initializer bean 'dispatcherServletRegistration';
order=2147483647, resource=class path resource
[org/springframework/boot/autoconfigure/web/servlet/DispatcherServletAutoConfiguration
$DispatcherServletRegistrationConfiguration.class]
[main] [..] : Created Filter initializer for bean 'characterEncodingFilter'; order=-
2147483648, resource=class path resource
[org/springframework/boot/autoconfigure/web/servlet/HttpEncodingAutoConfiguration.clas
s]
[main] [..] : Created Filter initializer for bean 'hiddenHttpMethodFilter'; order=-
10000, resource=class path resource
[org/springframework/boot/autoconfigure/web/servlet/WebMvcAutoConfiguration.class]
[main] [..] : Created Filter initializer for bean 'formContentFilter'; order=-9900,
resource=class path resource
[org/springframework/boot/autoconfigure/web/servlet/WebMvcAutoConfiguration.class]
```

```
[main] [..] : Created Filter initializer for bean 'requestContextFilter'; order=-105,
resource=class path resource
[org/springframework/boot/autoconfigure/web/servlet/WebMvcAutoConfiguration$WebMvcAuto
ConfigurationAdapter.class]
[main] [..] : Mapping filters: characterEncodingFilter urls=[/*],
hiddenHttpMethodFilter urls=[/*], formContentFilter urls=[/*], requestContextFilter
urls=[/*]
[main] [..] : Mapping servlets: dispatcherServlet urls=[/]
[main] [..] : HikariPool-1 - Starting...
[main] [..] : HikariPool-1 - Start completed.
[main] [..] : HHH000204: Processing PersistenceUnitInfo [
    name: default
    ...]
[main] [..] : HHH000412: Hibernate Core {5.3.7.Final} ③
[main] [..] : HHH000206: hibernate.properties not found
[main] [..] : HCANN000001: Hibernate Commons Annotations {5.0.4.Final}
[main] [..] : HHH000400: Using dialect: org.hibernate.dialect.H2Dialect
[main] [..] : HHH000476: Executing import script
'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@1ef93e01'
[main] [..] : Initialized JPA EntityManagerFactory for persistence unit 'default'
[main] [..] : Mapped [/**/favicon.ico] onto ResourceHttpRequestHandler [class path
resource [META-INF/resources/], class path resource [resources/], class path resource
[static/], class path resource [public/], ServletContext resource [/], class path
resource []]
[main] [..] : Patterns [/**/favicon.ico] in 'faviconHandlerMapping'
[main] [..] : Initializing ExecutorService 'applicationTaskExecutor'
[main] [..] : ControllerAdvice beans: 0 @ModelAttribute, 0 @InitBinder, 1
RequestBodyAdvice, 1 ResponseBodyAdvice
[main] [..] : spring.jpa.open-in-view is enabled by default. Therefore, database
queries may be performed during view rendering. Explicitly configure spring.jpa.open-
in-view to disable this warning
[main] [..] :
    c.m.a.t.t.c.TrainerController: ④
    {GET /trainers/}: getAllTrainers()
    {GET /trainers/{name}}: getTrainer(String)
[main] [..] :
    o.s.b.a.w.s.e.BasicErrorController:
    { /error, produces [text/html]}: errorHtml(HttpServletRequest,HttpServletResponse)
    { /error}: error(HttpServletRequest)
[main] [..] : 4 mappings in 'requestMappingHandlerMapping'
[main] [..] : Detected 0 mappings in 'beanNameHandlerMapping'
[main] [..] : Mapped [/webjars/**] onto ResourceHttpRequestHandler ["classpath:/META-
INF/resources/webjars/"]
[main] [..] : Mapped [/**] onto ResourceHttpRequestHandler ["classpath:/META-
INF/resources/", "classpath:/resources/", "classpath:/static/", "classpath:/public/",
"/"]
[main] [..] : Patterns [/webjars/**, /**] in 'resourceHandlerMapping'
[main] [..] : ControllerAdvice beans: 0 @ExceptionHandler, 1 ResponseBodyAdvice
[main] [..] : Tomcat started on port(s): 8081 (http) with context path ''
[main] [..] : Started TrainerApi in 3.622 seconds (JVM running for 4.512)
```

① Wao!

② On voit que un Tomcat est démarré, comme la dernière fois. Mais cette fois-ci, il utilise bien le port `8081` comme demandé dans le fichier `application.properties`

③ Le nom `Hibernate` vous dit quelque chose? spring-data utilise hibernate comme implémentation de la norme JPA !

④ On voit également nos controlleurs !

On peut maintenant tester les URLs suivantes:

- http://localhost:8081/trainers/
- http://localhost:8081/trainers/Ash

6.5. Le test d'intégration

Comme pour le TP précédent, nous allons compléter nos développements avec un test d'intégration.

Créez le test suivant:

src/test/java/fr/univ_lille/alom/trainers/TrainerControllerIntegrationTest.java

```
1 @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
2 class TrainerControllerIntegrationTest {
3
4     @LocalServerPort
5     private int port;
6
7     @Autowired
8     private TestRestTemplate restTemplate;
9
10    @Autowired
11    private TrainerController controller;
12
13    @Test
14    void trainerController_shouldBeInstanciated(){
15        assertNotNull(controller);
16    }
17
18    @Test
19    void getTrainer_withNameAsh_shouldReturnAsh() {
20        var ash = this.restTemplate.getForObject("http://localhost:" + port +
  "/trainers/Ash", Trainer.class);
21        assertNotNull(ash);
22        assertEquals("Ash", ash.getName());
23        assertEquals(1, ash.getTeam().size());
24
25        assertEquals(25, ash.getTeam().get(0).getPokemonTypeId());
26        assertEquals(18, ash.getTeam().get(0).getLevel());
27    }
28
```

```
29      @Test
30      void getAllTrainers_shouldReturnAshAndMisty() {
31          var trainers = this.restTemplate.getForObject("http://localhost:" + port +
    "/trainers/", Trainer[].class);
32          assertNotNull(trainers);
33          assertEquals(2, trainers.length);
34
35          assertEquals("Ash", trainers[0].getName());
36          assertEquals("Misty", trainers[1].getName());
37      }
38  }
```

7. Utilisation d'une base de données managée sur le cloud public

Pour remplacer notre base de données embarquée, nous pouvons nous connecter sur une base de données réelle, que nous allons instancier sur un cloud public.

Pour ce faire, nous avons de nombreux clouds à disposition, avec des offres gratuites :

- clever-cloud :
  - clever-cloud (□□) propose des bases de données postgresql managées gratuites, pour une taille de 250Mo maximum, avec 5 connexions simultanées.
- AWS (□□): le cloud d'Amazon
  - Amazon propose des bases de données managées via son service RDS. Ce service est disponible gratuitement pendant 12 mois à compter de la date de création du compte, et dans la limite de 750 heures / mois (une carte bleue doit être saisie)
- GCP (□□): le cloud de Google
  - Google propose $300 de crédits offerts à l'inscription (une carte bleue doit être saisie)
- heroku (□□):
  - Heroku propose également des bases de données postgresql managées gratuites, dans la limite de 10 000 lignes, avec 10 connexions simultanées.

Pour ce TP, je prends l'exemple de clever-cloud, qui a aussi accepté de nous sponsoriser en nous offrant une organisation avec des crédits illimités □.

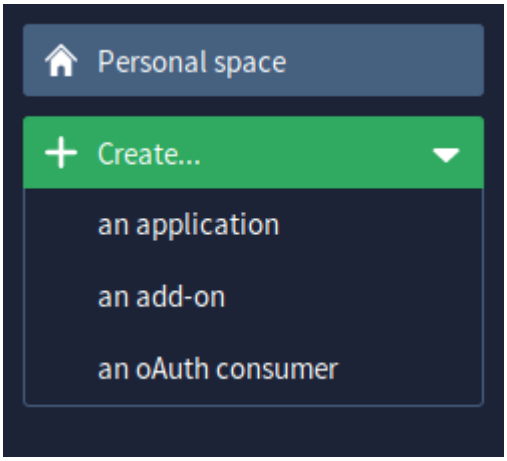
7.1. clever-cloud

Créez un compte sur https://www.clever-cloud.com, en utilisant votre adresse mail d'étudiant !

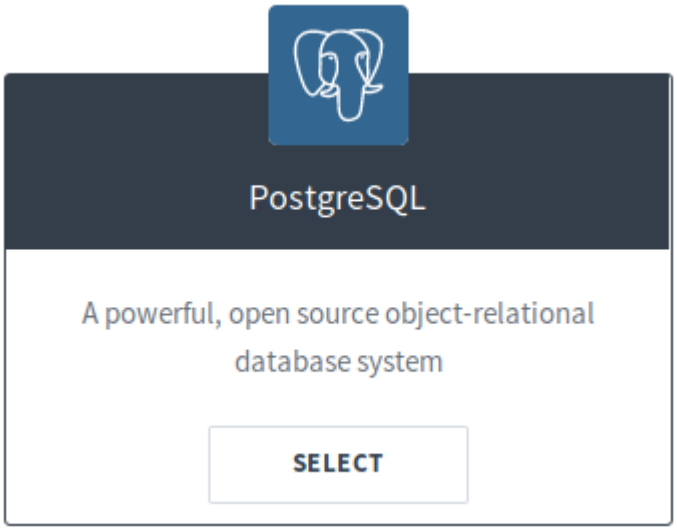
7.1.1. Instanciation de la base de données

Une fois votre compte créé, vous pouvez instancier une base de données en quelques clics !

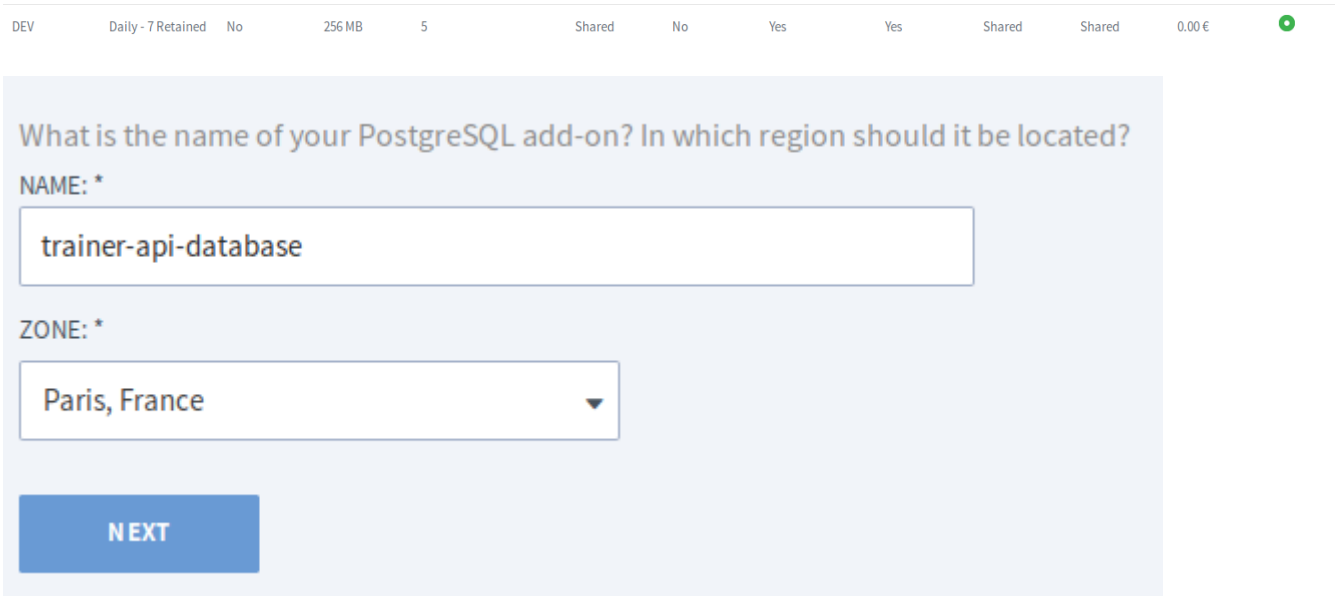
Dans la console, sélectionnez `Create > an add-on`.



Sélectionnez la base de données `postgresql`



Sélectionnez le plan `DEV`, qui est gratuit Donnez un nom à votre base de données, et sélectionnez la région `Paris` (un hébergement de notre base de données à Montréal créerait des temps de latence importants!)



Validez, et attendez quelques secondes! Votre base de données est prête!

Accédez au dashboard de votre base de données. Vous pourrez y trouver:

- Les informations de connexion à votre base de données
- Des menus permettant de réinitialiser votre base, re-généré de nouveaux identifiants de connexions, ou effectuer un backup.
- Vous pouvez également accéder à une interface "PGStudio" vous permettant de naviguer dans votre base de données.

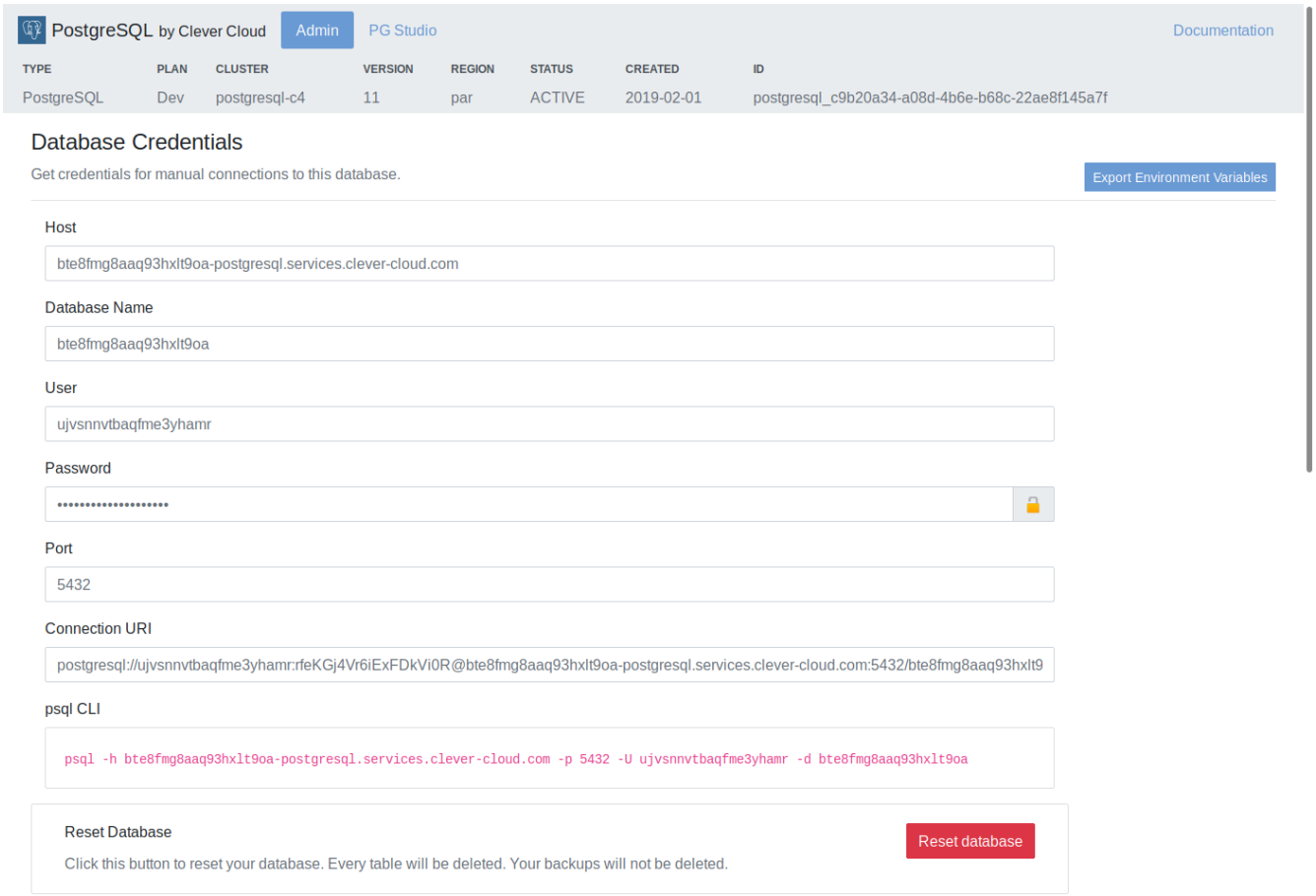


Figure 1. la page d'informations de votre base de données !

7.2. Configuration pour spring-boot

Nous allons utiliser votre base de données nouvellement créée pour votre application !

Modifiez votre `pom.xml` :

- Ajoutez une dépendance à `postgresql` (qui contiendra le driver JDBC postgresql)
- On positionne cette dépendance en scope `runtime`, car ce driver n'est nécessaire qu'à l'exécution

pom.xml

```
1 <dependency>
2     <groupId>com.h2database</groupId>
3     <artifactId>h2</artifactId>
```

```
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```

Modifiez votre fichier `application.properties` pour y renseigner les informations de connexion à votre base de données :

application.properties

```
# utilisation de vos parametres de connexion ①
spring.datasource.url=jdbc:postgresql://bae8fmg8aaq93hxlt9oa-postgresql.services.clever-cloud.com:5432/bae8fmg8aaq93hxlt9oa
spring.datasource.username=uavsnnvtbaqfme3yhamr
spring.datasource.password=rfeKGj4Vr6iExFDkVi0R

# personnalisation de hibernate ②
spring.jpa.hibernate.ddl-auto=update

# personnalisation du pool de connexions ③
spring.datasource.hikari.maximum-pool-size=1
```

① Renseignez les paramètre de connexion à votre base de donnée (remplacez les valeurs de mon exemple)

② L'utilisation du paramètre `spring.jpa.hibernate.ddl-auto` permet à hibernate de générer le schéma de base de données au démarrage de l'application.

③ par défault, spring-boot utilise le pool de connexion HikariCP pour gérer les connexions à la base de données. Comme le nombre de connexions est limité dans notre environnement, nous précisions que la taille maximale du pool est 1.

Dans le fichier `src/test/resources/application.properties`, forcez les tests à utiliser la base de données *h2* avec les properties suivantes : .src/test/resources/application.properties

```
spring.datasource.url=jdbc:h2:mem:test
```

Attention, la *Connection URI* que clever-cloud vous affiche contient le login et le mot de passe d'accès à la base de données, et n'est pas une URL JDBC, ne la copiez pas! Re-construisez votre URL JDBC en prenant les champs *Host* et *Database Name*.

Pour rappel, la liste des propriétés acceptées par spring-boot peut se trouver dans leur documentation.

Le paramètre `spring.jpa.hibernate.ddl-auto` peut prendre les valeurs suivantes :

- create : le schéma est créé au démarrage de l'application, toutes les données existantes sont écrasées
- create-drop : le schéma est créé au démarrage de l'application, puis supprimé à son extinction (utile en développement)
- update : le schéma de la base de données est mis à jour si nécessaire, les données ne sont pas impactées
- validate : le schéma de la base de données est vérifié au démarrage

Dans IntelliJ, vous pouvez également vous connecter à votre base de données, utilisez le plugin `Database Tools & SQL`.

7.3. Déploiement chez Clever-Cloud !

Pour cette partie, je dois vous donner les droits d'accès à l'organisation. Appelez-moi pour que je puisse le faire avec vous !

7.3.1. Configuration de votre application

Clever-Cloud est capable d'exécuter tout type d'application. Nous allons lui indiquer quelle tâche maven appeler pour démarrer notre application.

Créez le fichier `maven.json` dans le répertoire `clevercloud` de votre TP, pour lui indiquer d'utiliser la tâche maven `spring-boot:run` :

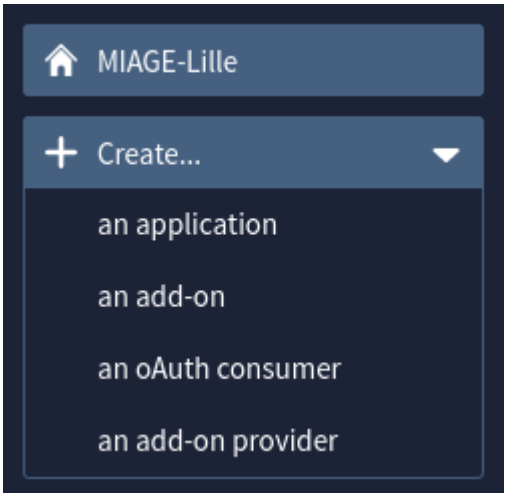
clevercloud/maven.json

```
{
    "deploy": {
        "goal": "spring-boot:run"
    }
}
```

7.3.2. Création de l'application

Sur le dashboard Clever-Cloud, dans l'organisation `Université de Lille`, cliquez sur `Create··· > an application`.

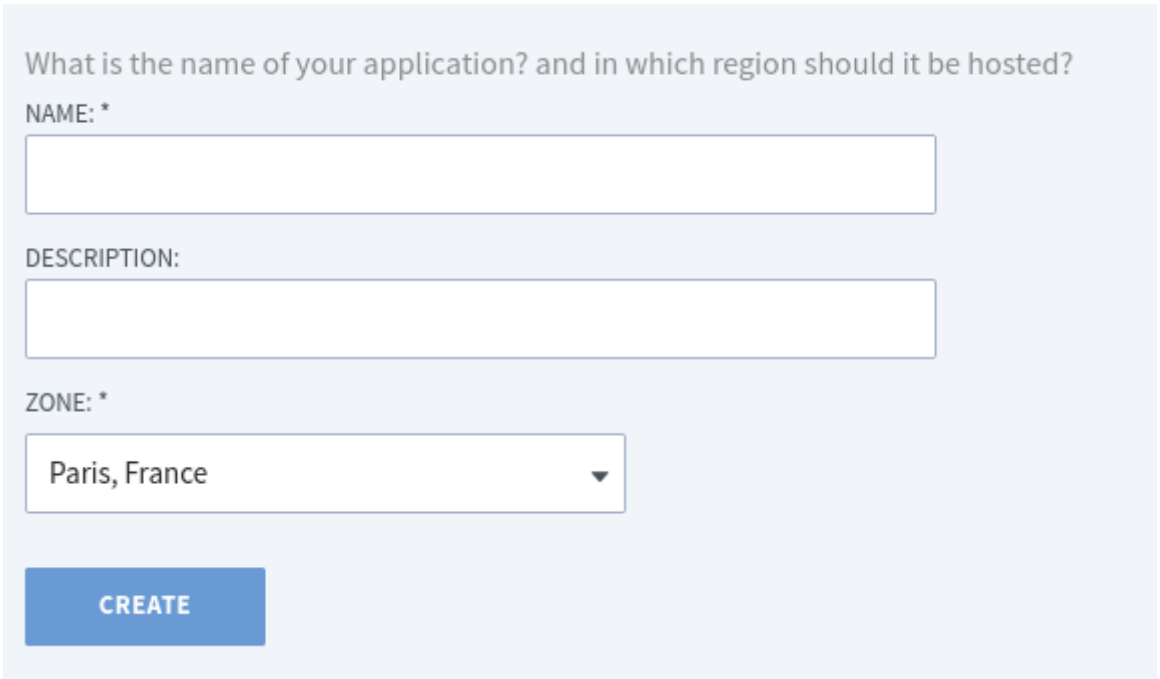
Attention, assurez-vous de déployer dans l'organisation `Université de Lille`, sinon des factures seront émises pour votre compte utilisateur !



De là, vous pouvez soit : * créer une application "Brand new". La suite de cette procédure utilise cette option. * créer une application depuis un repo Github (inutile dans notre cas).

Sélectionnez "Java + Maven"

Java + Maven



Nommez votre application comme votre repository, exemple : `trainer-api-julien.wittouck`. Cela vous permettra de retrouver facilement vos applications quand tout le monde aura créé la sienne !

Validez les écrans.

Une fois l'application créée, rendez-vous dans l'onglet *Environment variables* de votre application. Une variable existe déjà, nommée *CC_JAVA_VERSION*. Modifiez la variable pour y mettre la valeur **17**.

Ajoutez aussi une variable *SERVER_PORT* ayant pour valeur *8080*. Cette variable sera utile pour surcharger le port d'écoute de Tomcat qui est positionné dans le fichier properties.

Dans cet écran, nous pourrons à l'avenir positionner d'autres variables !

7.3.3. Intégration continue

Nous allons branche une intégration continue sur notre projet, pour déployer automatiquement !

Ajoutez la section suivante dans votre fichier `.gitlab-ci.yml` :

.gitlab-ci.yml

```
deploy:
  image:
    name: clevercloud/clever-tools
    entrypoint: ["/bin/sh", "-c"]
  stage: deploy
  script:
    - clever deploy --force
```

Ajoutez un fichier `.clever.json` à la racine de votre projet

.clever.json

```
{
  "apps": [
    {
      "app_id": "",
      "org_id": "orga_d02d9099-9664-47fd-8029-d90e36628e1d",
      "deploy_url": "https://push-n3-par-clevercloud-customers.services.clever-
cloud.com/",
      "name": "trainer-api",
      "alias": "trainer-api"
    }
  ]
}
```

Récupérez le `app_id` en allant dans l'onglet *Information* de votre application sur Clever Cloud.

Positionnez le `app_id` dans le champ de même nom, et à la fin de la `deploy_url` (derrière l'url https://push-n3-blabla/).

Générez un access token et un secret Clever Cloud pour que le pipeline GitLab puisse s'authentifier.

Rendez-vous à cette URL : https://console.clever-cloud.com/cli-oauth

Récupérez le Token et Secret affichés.

Rendez-vous dans votre projet GitLab, dans la section *Settings* / *CI/CD* / Variables.

Créez deux variables `CLEVER_TOKEN` et `CLEVER_SECRET`, de type *Variable*, avec les valeurs récupérées.

Si toutes les étapes sont correctes, chaque `git push` occasionnera un déploiement de votre application !

8. Pour aller plus loin

- Implémentez la création et la mise à jour d'un `Trainer` (route en POST/PUT) + Tests unitaires et tests d'intégration

```
POST /trainers/

{
  "name": "Bug Catcher",
  "team": [
    {"pokemonTypeId": 13, "level": 6},
    {"pokemonTypeId": 10, "level": 6}
  ]
}
```

- Implémentez la suppression d'un `Trainer` (route en DELETE) + Tests unitaires et tests d'intégration

```
DELETE /trainers/Bug%20Catcher
```