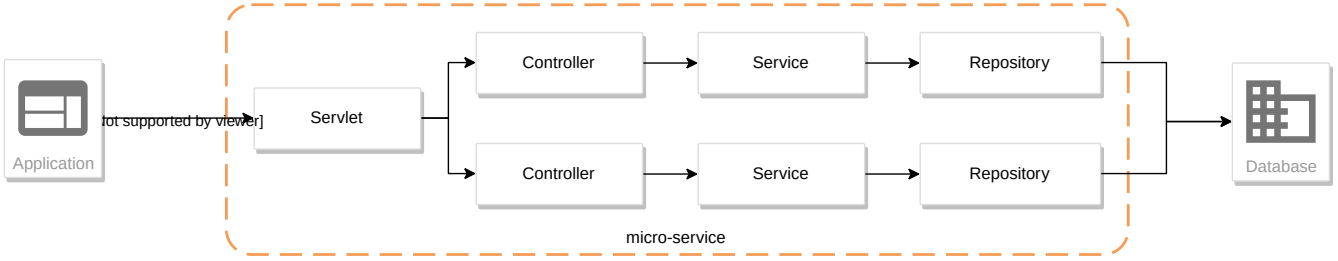
ALOM - TP Persistence

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 contrôleur implémente une méthode Java par route HTTP, récupère les paramètres, et appelle le service (rôle de routage)
- le service implémente le métier de notre microservice
- le repository représente les accès aux données (avec potentiellement une base de données)

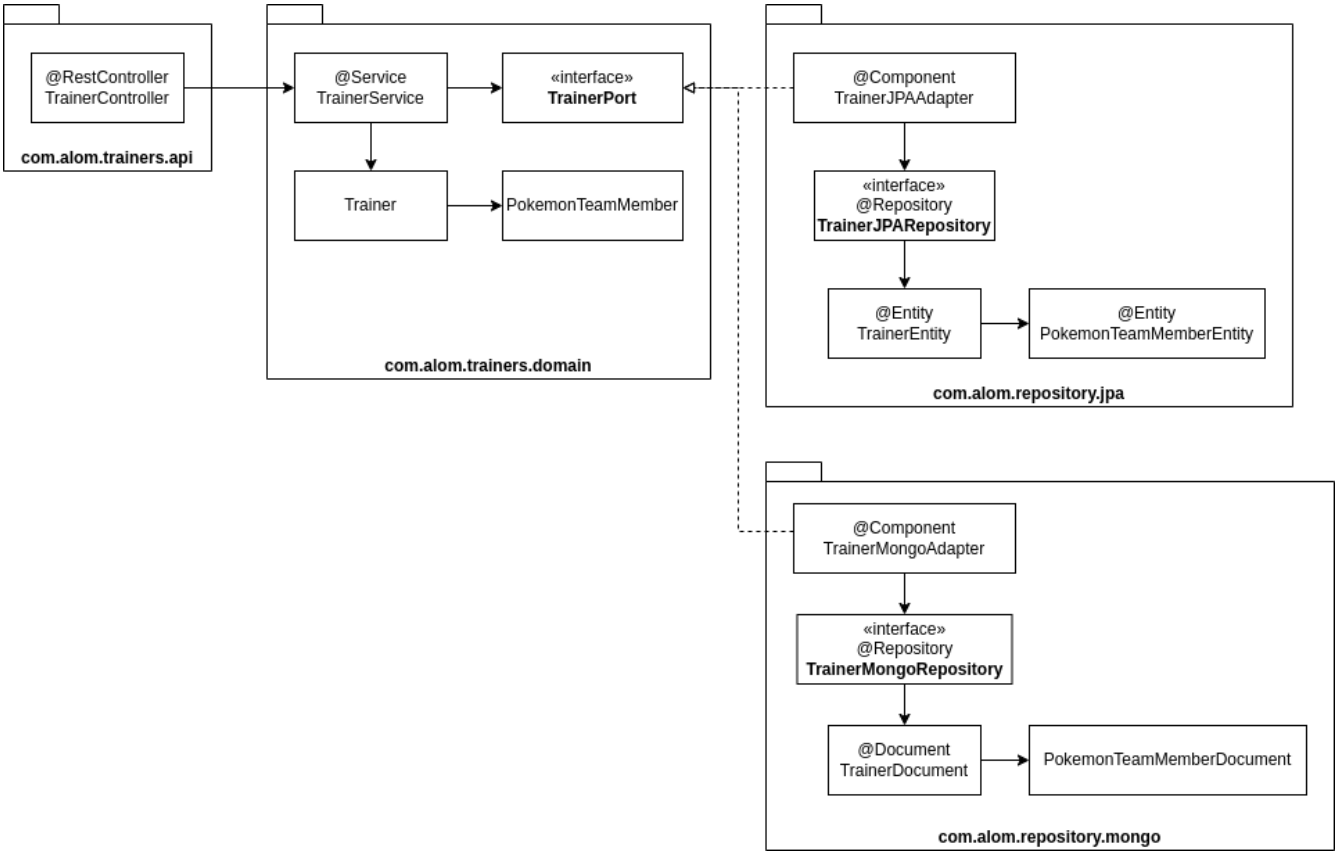
Et pour s'amuser un peu, nous allons réaliser un micro-service qui nous renvoie des données sur les dresseurs de Pokemon !

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 contrôleur spring pour gérer nos requêtes HTTP / REST
5. charger quelques données

Nous repartons de zéro pour ce TP !

Voici le schéma du code qu'il faut écrire :



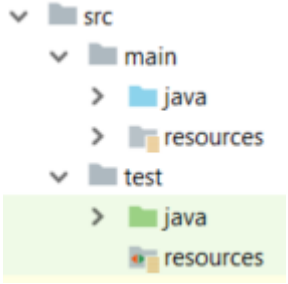
2. GitLab

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 !

À partir de ce TP, votre repository nouvellement créé contiendra au moins 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

```
<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>3.3.4</version> ②
    </parent>

    <properties>
        <java.version>21</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>

     <build> ⑤
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

① Modifiez votre `artifactId`

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

③ en java 21...

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. Les classes du domaine

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 `PokemonTeamMember`

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

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

① Notre classe de dresseur de Pokemon

② Son nom

③ La liste de ses pokemons

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

Vous pouvez utiliser des records à cette étape !

src/main/java/fr/univ_lille/alom/trainers/domain/PokemonTeamMember.java

```
1 public class PokemonTeamMember {
2
3     private int pokemonTypeId; ①
4
5     private int level; ②
```

```
6
7       public PokemonTeamMember() {
8       }
9
10      public PokemonTeamMember(int pokemonTypeId, int level) {
11          this.pokemonTypeId = pokemonTypeId;
12          this.level = level;
13      }
14
15      [...] ④
16  }
```

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

② le niveau de notre Pokemon !

Ajouter l'interface du TrainerPort !

src/main/java/fr/univ_lille/alom/trainers/domain/TrainerPort.java

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

Attention, ici, nous ne développerons pas l'implémentation du port, mais juste une interface qui servira par la suite. Il faudra lui ajouter quelques méthodes.

5. Le service métier

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

5.1. Le test unitaire

src/test/java/fr/univ_lille/alom/trainers/domain/TrainerServiceImplTest.java

```
1  class TrainerServiceImplTest {
2
3      @Test
4      void getAllTrainers_shouldCallThePort() {
5          var trainerPort = mock(TrainerPort.class);
6          var trainerService = new TrainerServiceImpl(trainerPort);
7
8          trainerService.getAllTrainers();
9
10         verify(trainerPort).findAll();
11     }
12
13     @Test
```

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

5.2. L'implémentation

L'interface Java

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

```
1  // TODO
2  class TrainerServiceImpl implements TrainerService { ①
3
4      private TrainerPort trainerPort;
5
6      public TrainerServiceImpl(TrainerPort trainerPort) {
7          this.trainerPort = trainerPort;
8      }
9
10     @Override
```

```
11      public Iterable<Trainer> getAllTrainers() {
12          // TODO
13      }
14
15      @Override
16      public Trainer getTrainer(String name) {
17          // TODO
18      }
19
20      @Override
21      public Trainer createTrainer(Trainer trainer) {
22          // TODO
23      }
24  }
```

① à implémenter !

Nous utilisons l'injection de dépendances entre le service et le port, car nous allons implémenter 2 versions de ce port, une pour des bases de données SQL (JPA), et une pour des bases de données documents (MongoDb).

6. Le repository JPA

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, SQL, embarquée dans un premier temps.

Nous commençons les développements avec une base de données embarquée, puis nous testerons ensuite une base de données dans un container Docker.

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

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

Ajoutez les dépendances suivantes dans votre `pom.xml`

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

6.2. Les classes d'entité

Dans un package `fr.univ_lille.alom.trainers.jpa`, créez les classes d'entité suivantes :

- `TrainerEntity` : correspond à l'objet du domaine `Trainer`, et contient les mêmes attributs
- `PokemonTeamMemberEntity` : correspond à l'objet du domain `PokemonTeamMember`, et contient les mêmes attributs, plus un identifiant unique

Nous ne pouvons pas utiliser les `record` de Java pour représenter les TrainerEntity/PokemonTeamMemberEntity. 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 .

6.3. Les test unitaires

Implémentez les tests unitaires suivants :

src/test/java/fr/univ_lille/alom/trainers/jpa/TrainerEntityTest.java

```
1  package fr.univ_lille.alom.trainers.jpa;
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 TrainerEntityTest {
10
11     @Test
12     void trainerEntity_shouldBeAnEntity(){
13         assertNotNull(TrainerEntity.class.getAnnotation(Entity.class)); ①
14     }
15
16     @Test
17     void trainerEntityName_shouldBeAnId() throws NoSuchFieldException {
18         assertNotNull(TrainerEntity.class.getDeclaredField("name").getAnnotation(
   Id.class)); ②
19     }
20
21     @Test
22     void trainerEntityTeam_shouldBeAElementCollection() throws NoSuchFieldException
   {
23         assertNotNull(TrainerEntity.class.getDeclaredField("team").getAnnotation
   (OneToMany.class)); ③
24     }
25
```

```
26 }
```

① Notre classe `TrainerEntity` 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 `TrainerEntity` et `PokemonTeamMemberEntity` doit également être annotée. Ici, un `TrainerEntity` possède une collection de `PokemonTeamMemberEntity`.

src/test/java/fr/univ_lille/alom/trainers/jpa/PokemonTeamMemberEntityTest.java

```
1  class PokemonTeamMemberEntityTest {
2
3      @Test
4      void pokemonTeamMemberEntity_shouldBeAnEntity(){
5          assertNotNull(PokemonTeamMemberEntity.class.getAnnotation(Entity.class));
   ①
6      }
7
8      @Test
9      void pokemonTeamMemberEntity_shouldHaveAnId(){
10         assertNotNull(PokemonTeamMemberEntity.class.getDeclaredField("id"
   ).getAnnotation(Id.class)); ②
11     }
12
13 }
```

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

② Une entité JPA doit avoir un champ `@Id`.

6.4. L'interface du repository JPA

Créez une interface de repository JPA nommée `TrainerJpaRepository`.

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/

Ajoutez un test pour cette interface de repository :

src/test/java/fr/univ_lille/alom/trainers/jpa/TrainerEntityJpaRepositoryTest.java

```
1  package fr.univ_lille.alom.trainers.jpa;
2
3  import static org.junit.jupiter.api.Assertions.*;
4
5  @DataJpaTest ①
```

```java
class TrainerEntityJpaRepositoryTest {

    @Autowired ②
    private TrainerEntityJpaRepository repository;

    @Test
    void trainerJpaRepository_shouldExtendsCrudRepository() throws NoSuchMethodException {
        assertTrue(CrudRepository.class.isAssignableFrom(TrainerEntityJpaRepository.class)); ③
    }

    @Test
    void trainerJpaRepositoryShouldBeInstanciedBySpring(){
        assertNotNull(repository);
    }

    @Test
    void testSave(){ ④
        var ash = new TrainerEntity("Ash");

        repository.save(ash);

        var saved = repository.findById(ash.getName()).orElse(null);

        assertEquals("Ash", saved.getName());
    }

    @Test
    void testSaveWithPokemons(){ ⑤
        var misty = new TrainerEntity("Misty");
        var staryu = new PokemonTeamMemberEntity(120, 18);
        var starmie = new PokemonTeamMemberEntity(121, 21);
        misty.setTeam(List.of(staryu, starmie));

        repository.save(misty);

        var saved = repository.findById(misty.getName()).orElse(null);

        assertEquals("Misty", saved.getName());
        assertEquals(2, saved.getTeam().size());
    }

}
```

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 construit 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 !

6.5. 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 !

7. L'adapter JPA

Il ne manque pas quelque chose ?

Les interfaces `TrainerPort` et `TrainerEntityJpaRepository` sont différentes.

Implémentez dans le package `fr.univ_lille.alom.trainers.jpa` une classe `TrainerJpaAdapter` qui implémente `TrainerPort`. Cette classe devra :

- recevoir en injection de dépendance l'interface `TrainerEntityJpaRepository`
- être elligible à l'injection de dépendance, en étant annotée `@Component` par exemple
- implémenter les méthodes de `TrainerPort`
- transformer les instances de `Trainer` en `TrainerEntity` et inversement là où c'est nécessaire

la transformation peut aussi être faite dans une méthode ou une classe consacrée, soyez créatifs.

8. Le controlleur

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

8.1. Le test unitaire

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

src/test/java/fr/univ_lille/alom/trainers/api/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");
    }
}
```

8.2. L'implémentation

Compléter l'implémentation du controller :

src/main/java/fr/univ_lille/alom/trainers/api/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 !

8.3. 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);
    assertNotNull(controllerAnnotation);

    var requestMappingAnnotation =
            TrainerController.class.getAnnotation(RequestMapping.class);
    assertArrayEquals(new String[]{"/trainers"}, requestMappingAnnotation.value());
}

@Test
void getAllTrainers_shouldBeAnnotated() throws NoSuchMethodException {
    var getAllTrainers =
            TrainerController.class.getDeclaredMethod("getAllTrainers");
    var getMapping = getAllTrainers.getAnnotation(GetMapping.class);

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

@Test
void getTrainer_shouldBeAnnotated() throws NoSuchMethodException {
    var getTrainer =
            TrainerController.class.getDeclaredMethod("getTrainer", String.class);
    var getMapping = getTrainer.getAnnotation(GetMapping.class);

    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 !

8.4. 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 !

8.4.1. 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

```
1 # on demande un niveau de logs TRACE a spring-web
2 logging.level.web=TRACE
3 # on modifie le port par defaut du tomcat !
4 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

8.4.2. 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

```
1 @Bean ②
2 @Autowired ③
3 public CommandLineRunner demo(TrainerPort port) { ①
4     return (args) -> { ④
```

```
5         var ash = new Trainer("Ash");
6         var pikachu = new PokemonTeamMember(25, 18);
7         ash.setTeam(List.of(pikachu));
8
9         var misty = new Trainer("Misty");
10        var staryu = new PokemonTeamMember(120, 18);
11        var starmie = new PokemonTeamMember(121, 21);
12        misty.setTeam(List.of(staryu, starmie));
13
14        // save a couple of trainers
15        port.save(ash); ⑤
16        port.save(misty);
17    };
18 }
```

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 !

8.4.3. 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

8.5. 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 }
```

9. Utilisation d'une base de données PostgreSQL dans un container docker

Démarrez une base de données PG avec Docker :

```
docker container run \
  -p 6543:5432 \ ①
  -e POSTGRES_PASSWORD=mysecretpassword \ ②
  postgres ③
```

① On mappe le port 6543 de la machine locale vers le port 5432 du container

② on utilise un mot de passe sécurisé

③ on utilise l'image docker `postgres` officielle.

9.1. 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

```
<dependency>
    <groupId>com.h2database</groupId>
    <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://localhost:6543/postgres
spring.datasource.username=postgres
spring.datasource.password=mysecretpassword

# 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
```

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`.

10. L'adapter Mongodb

Maintenant que le travail pour JPA est terminé, on implémente la connexion à la base de données MongoDB !

10.1. L'ajout de la dépendance spring-boot-data-mongodb

Ajoutez la dépendance `spring-boot-starter-data-mongodb` dans votre `pom.xml`

10.2. Les classes de documents

Dans un package `fr.univ_lille.alom.trainers.mongo`, créez les classes de documents suivantes :

- `TrainerDocument` : correspond à l'objet du domaine `Trainer`, et contient les mêmes attributs
- `PokemonTeamMemberDocument` : correspond à l'objet du domain `PokemonTeamMember`, et contient les mêmes attributs

Ajoutez sur la classe `TrainerDocument` l'annotation `@Document` pour indiquer qu'il s'agit d'une classe de document MongoDb.

10.3. L'interface du repository MongoDB

Créez une interface de repository Mongo nommée `TrainerMongoRepository`, ajoutez les méthodes similaires à ce qui avait été fait avec le `TrainerEntityJpaRepository`.

10.4. L'adapter MongoDb

Comme ça a été fait pour faire le lien entre `TrainerPort` et `TrainerEntityJpaRepository`, implémentez dans le package `fr.univ_lille.alom.trainers.mongo` une classe `TrainerMongoAdapter` qui implémente `TrainerPort`. Cette classe devra :

- recevoir en injection de dépendance l'interface `TrainerMongoRepository`
- implémenter les méthodes de `TrainerPort`
- transformer les instances de `Trainer` en `TrainerDocument` et inversement là où c'est nécessaire

la transformation peut aussi être faite dans une méthode ou une classe consacrée, soyez créatifs.

11. Utilisation d'une base de données MongoDB dans un container docker

Démarrez une base de données PG avec Docker :

```
docker container run \
  -p 38128:27017 \ ①
  -e MONGO_INITDB_ROOT_USERNAME=root \ ②
  -e MONGO_INITDB_ROOT_PASSWORD=monpasswordsupersecret \ ②
  mongo ③
```

① On mappe le port 38128 de la machine locale vers le port 27017 du container

② on utilise un user et un mot de passe sécurisé

③ on utilise l'image docker `mongo` officielle.

12. Configuration de Spring Boot pour MongoDb

Configurez les properties MongoDb pour Spring Boot

application.properties

```
1 # utilisation de vos parametres de connexion ①
2 spring.data.mongodb.host=
3 spring.data.mongodb.port=
```

```
spring.data.mongodb.database=admin
spring.data.mongodb.username=
spring.data.mongodb.password=
```

Ajoutez dans le package `fr.univ_lille.alom.trainers.mongo` une classe `MongoDbConfiguration`, qui sera annotée `@Configuration` et `@EnableMongoRepositories`.

Démarrez votre application, et observez ce qu'il se passe (ça ne doit pas démarrer).

On utilisera plus tard (dans 2 ou 3 séances) des profils Spring pour pouvoir choisir quelle implémentation utiliser, JPA ou Mongo, en attendant, supprimez l'annotation `@Component` du `TrainerJpaAdapter` pour utiliser la version MongoDb. Vous devez également supprimer les properties liées à JPA.

Créez un fichier `application-jpa.properties` et déplacez-les properties JPA dedans :

application-jpa.properties

```
# utilisation de vos parametres de connexion ①
spring.datasource.url=jdbc:postgresql://localhost:6543/postgres
spring.datasource.username=postgres
spring.datasource.password=mysecretpassword

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

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

Ajoutez également ces paramètres d'annotation dans votre classe `TrainerApiApplication`:

TrainerApiApplication.java

```
@SpringBootApplication(exclude = {
    DataSourceAutoConfiguration.class,
    DataSourceTransactionManagerAutoConfiguration.class,
    HibernateJpaAutoConfiguration.class
})
public class TrainerApiApplication {}
```

Si vous avez tout bien implémenté, on est proche d'une architecture hexagonale !

13. 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
```