ALOM - TP 5 - GUI - MVC & Templating

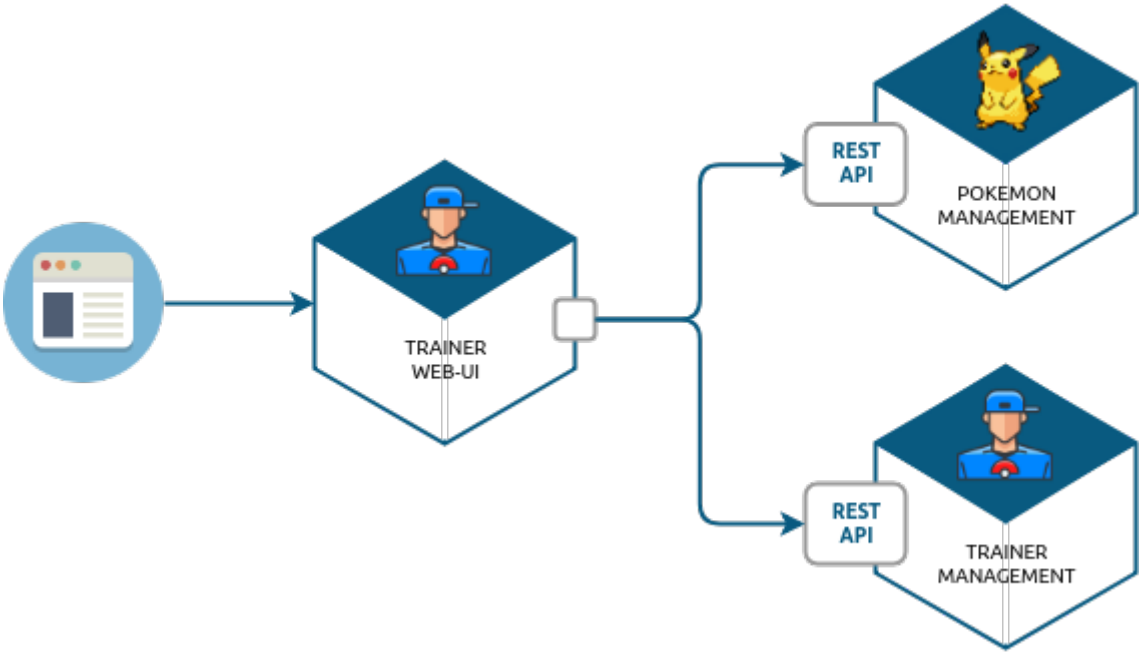
Table of Contents

1. Présentation et objectifs

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

Nous allons aujourd'hui développer la WEB-UI de gestion des dresseurs de Pokemon.

Ce micro-service se connectera au micro service de pokemon management et trainer management !



On ressemble de plus en plus à une architecture micro-service, comme celle d'UBER !

Nous allons développer :

1. deux repositories, qui accèdent aux données de trainer management et pokemon management
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. créer des templates pour afficher nos données

Nous repartons de zéro pour ce TP !

2. Github

Cliquez sur le lien suivant pour créer votre repository git: GitLab classroom

Clonez ensuite votre repository git sur votre poste !

N'oubliez pas ! Vous n'avez pas besoin de forker ce repostiory pour travailler, il

vous appartient !

3. Architecture

Pour préparer les développements, on va également tout de suite créer quelques packages Java qui vont matérialiser notre architecture applicative.

Cette architecture est maintenant habituelle pour vous ! C'est l'architecture que l'on retrouve sur de nombreux projets

Créer les packages suivants:

- `fr.univ_lille.alom.game_ui.views` : va contenir les controlleurs MVC de notre application
- `fr.univ_lille.alom.game_ui.config` : va contenir la configuration de notre application
- `fr.univ_lille.alom.game_ui.pokemonTypes` : va contenir les classes liées aux pokemons (bo et services)
- `fr.univ_lille.alom.game_ui.trainers` : va contenir les classes liées aux dresseurs (bo et services)

Notre GUI va manipuler des concepts de plusieurs domaines métier (Trainer et Pokemon). Nous organisons notre application pour refléter ces domaines.

Notre projet est prêt !

4. La première vue !

4.1. Le controlleur index

Nous allons développer un `Controlleur` simple qui servira notre page d'index !

4.1.1. Le test unitaire

Implémentez le test unitaire suivant :

fr.univ_lille.alom.game_ui.views.IndexControllerTest.java

```
1  package fr.univ_lille.alom.game_ui.views;
2
3  import org.junit.jupiter.api.Test;
4  import org.springframework.stereotype.Controller;
5  import org.springframework.web.bind.annotation.GetMapping;
6
7  import static org.junit.jupiter.api.Assertions.*;
8
9  class IndexControllerTest {
10
11     @Test
```

```
12      void controllerShouldBeAnnotated(){
13          assertNotNull(IndexController.class.getAnnotation(Controller.class)); ①
14      }
15
16      @Test
17      void index_shouldReturnTheNameOfTheIndexTemplate() {
18          var indexController = new IndexController();
19          var viewName = indexController.index();
20
21          assertEquals("index", viewName); ②
22      }
23
24      @Test
25      void index_shouldBeAnnotated() throws NoSuchMethodException {
26          var indexMethod = IndexController.class.getMethod("index");
27          var getMapping = indexMethod.getAnnotation(GetMapping.class);
28
29          assertNotNull(getMapping);
30          assertArrayEquals(new String[]{"/"}, getMapping.value()); ③
31      }
32  }
```

① notre controller doit être annoté `@Controller` (à ne pas confondre avec `@RestController`)

② si le retour de la méthode du controlleur est une chaîne de caractères, cette chaîne sera utilisée pour trouver la vue à afficher

③ on écoute les requêtes arrivant à `/`

4.1.2. L'implémentation

Implémentez la classe IndexController !

fr.univ_lille.alom.game_ui.views.IndexController.java

```
1  // TODO
2  public class IndexController {
3
4      // TODO
5      public String index(){
6          return ""; // TODO
7      }
8
9  }
```

4.2. Ajout du moteur de template

Nous allons utiliser le moteur de template `Mustache`.

Pour ce faire, ajoutez la dépendance suivante dans votre pom.xml

pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-mustache</artifactId>
</dependency>
```

Par défaut, les templates `Mustache` :

- sont positionnés dans un répertoire du classpath `/templates` (donc dans `src/main/resources/templates`, puisque Maven ajoute `src/main/resources` au classpath).
- sont des fichiers nommés `.mustache`

Les propriétés disponibles sont détaillées dans la documentation Spring

Nous allons modifier le suffixe des fichiers de template, pour être `.html`.

Créez le fichier `src/main/resources/application.properties` et ajoutez-y les propriétés suivantes.

src/main/resources/application.properties

```
①
spring.mustache.prefix=classpath:/templates/
②
spring.mustache.suffix=.html
③
server.port=9000
```

① On garde ici la valeur par défaut.

② On modifie la propriété pour prendre en compte les fichiers `.html` au lieu de `.mustache`

③ On en profite pour demander à Spring d'écouter sur le port 9000 !

4.3. Ajout du template !

Nous pouvons enfin ajouter notre template de page d'accueil !

La page d'accueil va contenir 2 liens, un lien pour se créer un compte, et un lien pour s'authentifier.

Créer le fichier `src/main/resources/templates/index.html`

src/main/resources/templates/index.html

```
<!doctype html> ①
<html lang="en">
<head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
```

```
    <title>Pokemon Manager</title>

    <!-- Bootstrap CSS --> ②
    <link href
="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" rel
="stylesheet" integrity="sha384-
T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEV/Dwwykc2MPK8M2HN" crossorigin
="anonymous">
</head>
<body>
<div class="container">
    <h1 class="pt-md-5 pb-md-5">Pokemon Manager</h1> ③

    <div>
        <a href="/login" class="btn btn-primary" role="button">
            Login
        </a>
        <a href="/register" class="btn btn-secondary" role="button">
            Register
        </a>
    </div>
</div>

</body>
</html>
```

① On crée une page HTML

② En important les CSS de Bootstrap par exemple

③ On affiche un titre !

4.4. L'application

Démarrez votre application et allez consulter le résultat sur http://localhost:9000 !

5. Création de compte

Nous allons maintenant créer un formulaire de saisie permettant à un utilisateur de se créer un compte.

5.1. Servir des ressources statiques

Par défaut, Spring est capable de servir des ressources statiques.

Pour ce faire, il suffit de les placer au bon endroit !

Télécharger l'image chen.png et placez-la dans le répertoire `src/main/resources/static/images` ou `src/main/resources/public/images`

Une image pokemon-logo.png est aussi disponible pour votre page d'accueil.

Le positionnement des ressources statiques est paramétrable à l'aide de l'application.properties :

application.properties

```
# Path pattern used for static resources. ①
spring.mvc.static-path-pattern=/**
# Locations of static resources. ②
spring.web.resources.static-locations=classpath:/META-
INF/resources/,classpath:/resources/,classpath:/static/,classpath:/public/
```

① Ce paramétrage indique que l'ensemble des requêtes entrantes peut être une ressource statique !

② Et on indique à spring dans quels répertoires il doit chercher les ressources !

5.2. Ajouter un formulaire

Créez une page `register.html`, contenant un formulaire :

register.html

```
<div class="row">
    <img  src="/images/chen.png" class="col-md-2"/> ①

    <div class="row col">

        <div style="white-space: pre-line"> ②
            Hello there!
            Welcome to the world of Pokémon!
            My name is Oak! People call me the Pokémon Prof!
            This world is inhabited by creatures called Pokémon!
            For some people, Pokémon are pets. Other use them for fights.
            Myself… I study Pokémon as a profession. First, what is your name?
        </div>

        <form action="/register" method="post"> ③
            <div class="form-group">
                <label for="trainerName">Trainer name</label>
                <input type="text" class="form-control" id="trainerName" name
="trainerName" aria-describedby="trainerHelp" placeholder="Enter your name">
                <small id="trainerHelp" class="form-text text-muted">This will be
your name in the game !</small>
            </div>
            <button type="submit" class="btn btn-primary">Submit</button>
        </form>
    </div>

</div>
```

① Nous ajoutons notre ressource statique.

② Le discours d'introduction original du Professeur Chen dans Pokémon Bleu et Rouge !

③ Un formulaire de création de dresseur !

Notez comme la ressource statique est référencée par `/images/chen.png`, et qu'elle est positionnée dans le répertoire `src/main/resources/static/images/chen.png`. Spring utilise le répertoire paramétré comme base de recherche, les sous-répertoires sont parcourus également !

5.3. Les impacts sur la couche controlleur

Créez un nouveau controlleur pour servir notre page `register`, et recevoir la requête `POST /register` de création.

5.3.1. Les tests unitaires

Ajouter les tests unitaires suivants :

fr.univ_lille.alom.game_ui.views.RegisterControllerTest.java

```
@Test
void registerNewTrainer_shouldReturnAModelAndView(){
    var registerController = new RegisterController();
    var modelAndView = registerController.registerNewTrainer("Blue");

    assertNotNull(modelAndView);
    assertEquals("registered", modelAndView.getViewName());
    assertEquals("Blue", modelAndView.getModel().get("name"));
}

@Test
void registerNewTrainer_shouldBeAnnotated() throws NoSuchMethodException {
    var registerMethod = RegisterController.class.getDeclaredMethod
("registerNewTrainer", String.class);
    var getMapping = registerMethod.getAnnotation(PostMapping.class);

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

5.3.2. L'implémentation

Implémenter le RegisterController.

fr.univ_lille.alom.game_ui.views.RegisterController.java

```
@Controller
public class RegisterController {
```

```
3
4       @GetMapping("/register")
5       String register(){
6           return "register";
7       }
8
9       // TODO
10      ModelAndView registerNewTrainer(String trainerName){
11          // TODO
12      }
13
14  }
```

5.3.3. Le nouveau template

Nous allons devoir également créer un nouveau template pour afficher le résultat.

Créez le template `registered.html`

src/main/resources/templates/registered.html

```
1  <!doctype html> ①
2  <html lang="en">
3  <head>
4      <!-- Required meta tags -->
5      <meta charset="utf-8">
6      <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-
   fit=no">
7      <title>Pokemon Manager</title>
8
9      <!-- Bootstrap CSS -->
10     <link href
   ="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" rel
   ="stylesheet" integrity="sha384-
   T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEV/Dwwykc2MPK8M2HN" crossorigin
   ="anonymous">
11 </head>
12 <body>
13 <div class="container">
14     <h1 class="pt-md-5 pb-md-5">Pokemon Manager - Welcome {{name}}</h1> ①
15
16     <div class="row">
17         <img  src="/images/chen.png" class="col-md-2"/>
18
19         <div class="row col-md-10">
20
21             <p style="white-space: pre-line">
22                 Right! So your name is {{name}}! ①
23
24                 {{name}}! ①
```

```
25
26                 Your very own Pokémon legend is about to unfold!
27                 A world of dreams and adventures with Pokémon awaits!
28                 Let's go!
29             </p>
30             <p>
31                 <a href="/profile" class="btn btn-primary" role="button">View you
   profile !</a>
32             </p>
33
34         </div>
35
36     </div>
37
38 </div>
39
40 </body>
41 </html>
```

① On utilise le champ `name` du model pour alimenter notre titre et notre texte !

6. Ajouter du layouting

6.1. Le header

Nous allons utiliser l'inclusion de templates pour éviter de copier/coller notre header de page sur l'ensemble de notre application !

Créez un répertoire `layout` dans `src/main/resources/templates`. Ce répertoire va nous permettre de gérer les templates liés à la mise en page de notre application.

Dans le répertoire `layout`, créez un fichier que l'on appellera `header.html` :

header.html

```
1 <!doctype html> ①
2 <html lang="en">
3 <head>
4     <!-- Required meta tags -->
5     <meta charset="utf-8">
6     <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-
   fit=no">
7     <title>Pokemon Manager</title>
8
9     <!-- Bootstrap CSS -->
10    <link href
   ="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" rel
   ="stylesheet" integrity="sha384-
   T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEV/Dwwykc2MPK8M2HN" crossorigin
```

```
    ="anonymous">
11 </head>
```

6.2. L'inclusion

L'utilisation de notre header dans un template se fait alors avec une inclusion `Mustache`.

Modifiez vos templates pour utiliser l'inclusion :

index.html

```
1 {{> /layout/header}}
2
3 <body>
4     [...]
5 </body>
```

7. Se connecter aux micro-services

Nous allons maintenant appeler le micro-service pokemon-type-api, que nous avons écrit lors du TP 3 !.

Pour ce faire, nous allons commencer par déployer ce TP sur Clever Cloud !

7.1. Déploiement chez Clever-Cloud !

Reprenez la procédure de déploiement du TP précédent pour déployer votre `pokemon-type-api`.

7.2. Le business object

La classe du business object va être la copie de la classe du micro-service que l'on va consommer.

Nous avons donc besoin ici de deux record (que vous pouvez copier/coller depuis votre TP 3 !) :

- PokemonType: représentation d'un type de Pokemon
- Sprites: représentation des images du Pokemon (avant et arrière)

fr.univ_lille.alom.game_ui.pokemonTypes.PokemonType

```
1 public record PokemonType {}
```

fr.univ_lille.alom.game_ui.pokemonTypes.Sprites

```
1 public record Sprites {}
```

7.3. Le service

7.3.1. L'interface PokemonService

Écrire l'interface de service suivante :

fr.univ_lille.alom.game_ui.pokemonTypes.PokemonTypeService

```
1 public interface PokemonTypeService {
2
3     List<PokemonType> listPokemonsTypes();
4
5 }
```

7.3.2. La configuration du RestTemplate

Par défaut, Spring n'instancie pas de RestTemplate.

Il nous faut donc en instancier un, et l'ajouter à l' `application context` afin de le rendre disponible en injection de dépendances.

Pour ce faire, nous allons développer une simple classe de configuration :

fr.univ_lille.alom.game_ui.config.RestConfiguration.java

```
1 @Configuration ①
2 public class RestConfiguration {
3
4     @Bean ②
5     RestTemplate restTemplate(){
6         return new RestTemplate(); ③
7     }
8
9 }
```

① L'annotation `@Configuration` enregistre notre classe `RestConfiguration` dans l'application context (comme `@Component`, ou `@Service`)

② L'annotation `@Bean` permet d'annoter une méthode, dont le résultat sera enregistré comme un bean dans l' `application context` de spring.

7.3.3. Le test d'intégration

Implémentez le test d'intégration suivant :

fr.univ_lille.alom.game_ui.pokemonTypes.PokemonTypeServiceImplTest

```
1 package fr.univ_lille.alom.game_ui.pokemonTypes;
2
3 import org.junit.jupiter.api.Test;
```

```
4  import org.springframework.beans.factory.annotation.Autowired;
5  import
   org.springframework.boot.test.autoconfigure.web.client.AutoConfigureWebClient;
6  import org.springframework.boot.test.autoconfigure.web.client.RestClientTest;
7  import org.springframework.core.io.ClassPathResource;
8  import org.springframework.http.MediaType;
9  import org.springframework.stereotype.Service;
10 import org.springframework.test.context.TestPropertySource;
11 import org.springframework.test.web.client.MockRestServiceServer;
12 import org.springframework.web.client.RestTemplate;
13
14 import static org.assertj.core.api.Assertions.assertThat;
15 import static org.junit.jupiter.api.Assertions.assertNotNull;
16 import static
   org.springframework.test.web.client.match.MockRestRequestMatchers.requestTo;
17 import static
   org.springframework.test.web.client.response.MockRestResponseCreators.withSuccess;
18
19 @RestClientTest(PokemonTypeServiceImpl.class)
20 @AutoConfigureWebClient(registerRestTemplate = true)
21 @TestPropertySource(properties = "pokemonType.service.url=http://localhost:8080")
22 class PokemonTypeServiceIntegrationTest {
23
24     @Autowired
25     PokemonTypeService pokemonTypeService;
26
27     @Autowired
28     MockRestServiceServer server;
29
30     @Autowired
31     PokemonTypeService service;
32
33     @Autowired
34     RestTemplate restTemplate;
35
36     @Test
37     void serviceAndTemplateShouldNotBeNull(){
38         assertNotNull(service);
39         assertNotNull(restTemplate);
40     }
41
42     @Test
43     void listPokemonsTypes_shouldCallTheRemoteService() {
44         // given
45         var response = """
46                        [
47                            {
48                                "id": 151,
49                                "name": "mew",
50                                "types": ["psychic"]
51                            }
```

```
                        ]
                        """;
        server.expect(requestTo("http://localhost:8080/pokemon-types/"))
                .andRespond(withSuccess(response, MediaType.APPLICATION_JSON));

        var pokemons = pokemonTypeService.listPokemonsTypes();
        assertThat(pokemons).hasSize(1);
    }

    @Test
    void pokemonServiceImpl_shouldBeAnnotatedWithService(){
        assertNotNull(PokemonTypeServiceImpl.class.getAnnotation(Service.class));
    }

    @Test
    void setRestTemplate_shouldBeAnnotatedWithAutowired() throws
NoSuchMethodException {
        var setRestTemplateMethod = PokemonTypeServiceImpl.class.getDeclaredMethod
("setRestTemplate", RestTemplate.class);
        assertNotNull(setRestTemplateMethod.getAnnotation(Autowired.class));
    }

}
```

7.3.4. L'implémentation

Pour exécuter les appels au micro-service de gestion des pokemons, nous allons utiliser le `RestTemplate` de Spring. Le `RestTemplate` de Spring fournit des méthodes simples pour exécuter des requêtes HTTP. La librairie `jackson-databind` est utilisée pour transformer le résultat reçu (en JSON), vers notre classe de BO.

- la javadoc du RestTemplate ici
- la documentation de spring qui explique le fonctionnement et l'usage du `RestTemplate` ici

Implémentez la classe suivante :

fr.univ_lille.alom.game_ui.pokemonTypes.PokemonTypeServiceImpl

```
// TODO
public class PokemonTypeServiceImpl implements PokemonTypeService {

    public List<PokemonType> listPokemonsTypes() {
        // TODO
    }

    void setRestTemplate(RestTemplate restTemplate) {
        // TODO
    }
```

```
11
12     void setPokemonTypeServiceUrl(String pokemonServiceUrl) {
13         // TODO
14     }
15 }
```

7.3.5. L'injection des properties

Nous allons également utiliser l'injection de dépendance pour l'URL d'accès au service !

Les paramètres de configuration d'une application sont souvent injectés selon la méthode que nous allons voir !

Modifiez le fichier `application.properties` pour y ajouter une nouvelle propriété :

src/main/resources/application.properties

```
pokemonType.service.url=https://alom-pokemon-type-api.cleverapps.com  ①
```

① Nous utilisons un paramètre indiquant à quelle URL sera disponible notre micro-service de pokemons! Utilisez l'url à laquelle votre service est déployé, ou une URL en localhost.

Ajoutez le test unitaire suivant au PokemonServiceIntegrationTest

```
1 @Test
2 void setPokemonServiceUrl_shouldBeAnnotatedWithValue() throws NoSuchMethodException
  {
3     var setter = PokemonTypeServiceImpl.class.getDeclaredMethod
  ("setPokemonTypeServiceUrl", String.class);
4     var valueAnnotation = setter.getAnnotation(Value.class); ①
5     assertNotNull(valueAnnotation);
6     assertEquals("${pokemonType.service.url}", valueAnnotation.value()); ②
7 }
```

① On utilise une annotation `@Value` pour faire l'injection de dépendances de properties

② Une expression `${}` (spring-expression-language) est utilisée pour calculer la valeur à injecter

Un guide intéressant sur l'injection de valeurs avec l'annotation @Value ici

7.4. Le controlleur

Nous allons maintenant écrire le controlleur PokemonTypeController !

7.4.1. Le test unitaire

Implémentez le test unitaire suivant :

fr.univ_lille.alom.game_ui.views.PokemonTypeControllerTest.java

```java
package fr.univ_lille.alom.game_ui.views;

import fr.univ_lille.alom.game_ui.pokemonTypes.PokemonType;
import fr.univ_lille.alom.game_ui.pokemonTypes.PokemonTypeService;
import org.junit.jupiter.api.Test;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

import java.util.List;

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

class PokemonTypeControllerTest {
    @Test
    void controllerShouldBeAnnotated(){
        assertNotNull(PokemonTypeController.class.getAnnotation(Controller.class));
    }

    @Test
    void pokemons_shouldReturnAModelAndView() {
        var pokemonTypeService = mock(PokemonTypeService.class);

        var pikachu = new PokemonType("pikachu", 25);

        when(pokemonTypeService.listPokemonsTypes()).thenReturn(List.of(pikachu));

        var pokemonTypeController = new PokemonTypeController();
        pokemonTypeController.setPokemonTypeService(pokemonTypeService);
        var modelAndView = pokemonTypeController.pokedex();

        assertEquals("pokedex", modelAndView.getViewName());
        var pokemons = (List<PokemonType>)modelAndView.getModel().get
("pokemonTypes");
        assertEquals(1, pokemons.size());
        verify(pokemonTypeService).listPokemonsTypes();
    }

    @Test
    void pokemons_shouldBeAnnotated() throws NoSuchMethodException {
        var pokemonsMethod = PokemonTypeController.class.getDeclaredMethod
("pokedex");
        var getMapping = pokemonsMethod.getAnnotation(GetMapping.class);

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


```

```
48 }
```

7.4.2. L'implémentation

Implémentez le controlleur :

fr.univ_lille.alom.game_ui.views.PokemonTypeController.java

```
1 // TODO
2 public class PokemonTypeController {
3
4     // TODO
5     public ModelAndView pokedex(){
6         // TODO
7     }
8
9 }
```

7.4.3. Le template

Nous allons créer une petite page qui va afficher pour chaque type de pokémon son nom, son image, ainsi que ses statistiques

Créer le template suivant :

src/main/resources/templates/pokedex.html

```
1 {{> layout/header}}
2
3 <body>
4
5     <div class="container">
6         <h1 class="pt-md-5 pb-md-5">Pokedex</h1>
7
8         <div class="card-deck">
9             {{#pokemonTypes}} ①
10            <div class="col-md-3">
11                <div class="card shadow-sm mb-3">
12                    <div class="card-header">
13                        ②
14                        <h4 class="my-0 font-weight-normal">{{name}} <span class
  ="badge text-bg-secondary">Id {{}} </span></h4>③
15                    </div>
16                    <img class="card-img-top" src="{{}}" alt="Pokemon"/> ③
17
18                    <div class="card-body">
19                        <span class="badge text-bg-primary">Type : {{}}</span> ③
20                    </div>
21                </div>
22            </div>
```

```
23              {{/pokemonTypes}}
24          </div>
25
26      </div>
27
28  </body>
29  </html>
```

① Voici comment on itère sur une liste !

② On affiche quelques valeurs

③ à compléter

Attention, si le template n'est pas correct, la vue ne s'affichera pas quand elle est requêtée, et des exceptions peuvent apparaître dans la console, en particulier des `NullPointerException` ou `StringIndexOutOfBoundsException`.

8. Pour aller plus loin

1. Affichez sur le Pokedex les types de chaque Pokemon (plante, électrique…)
2. Affichez sur le Pokedex les images "vues de derrière"
3. Développez une page web qui affiche la liste des dresseurs de Pokemons (accessible à `/trainers`)
4. Développez une page qui affiche le détail d'un dresseur de Pokemon (accessible à `/trainers/{name}`) :
    - son nom
    - son équipe