POO : API & Outils
TD et TP

Thibaut Smith
thibaut.smith@soprasteria.com
thibaut.smith-fisseau@univ-lemans.fr

19 novembre 2021

1 TP 2

1.1 TP DM noté 1/2 : Réalisation d'une application web avec Spring Boot

Ce TP est noté !

- Vous pouvez le terminer chez vous ;
- A rendre via : UMTICE (cours : POO API et outillages, clé : PAAP4) ;
- Date limite : 31 décembre.

Vous allez réaliser une application à l'aide de Java et des technologies suivantes :

- Spring `https://spring.io/`
- Spring Boot `https://spring.io/projects/spring-boot`
- JPA `https://www.tutorialspoint.com/jpa/index.htm`
- Hibernate `http://hibernate.org/`
- H2 `http://www.h2database.com/html/main.html`
- Spring Data JPA `https://spring.io/projects/spring-data-jpa`
- Thymeleaf `https://www.thymeleaf.org/`

Nous allons utiliser beaucoup d'API, n'hésitez pas à chercher sur internet des indications et de la documentation. Suivez les étapes tranquillement en vérifiant que l'application fonctionne toujours.

Étape 1

Commencez par générer votre application sur le site de Spring, avec Spring Initializr : `https://start.spring.io/`

Étape 2

Vous allez générer un projet Maven, avec Java, et Spring Boot 2.2.1. Dans le groupe, ajoutez le nom du groupe Maven du projet que vous souhaitez créer. Pour l'artifact, choisissez "tp5".

Étape 3

Ajoutez les dépendances suivantes :

- Web
- JPA
- Hibernate
- H2
- DevTools
- Thymeleaf

Étape 4

Générez votre projet.

Étape 5

Cherchez une description succincte de chaque dépendance ajoutée pour trouver à quoi sert-elle, et ajoutez-y ces informations dans un fichier “README.md” à la racine de votre projet.

Étape 6

Lancez votre IDE, configurez le proxy si nécessaire, et importez le nouveau projet (Vérifiez les aides plus bas dans le document si besoin).

Étape 7

Pour lancer une application Spring Boot, vous pouvez utiliser la commande maven suivante :

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

Créez une nouvelle “Run Configuration” de type Maven et dans le goal, utilisez la commande ci-dessus. Lancez la nouvelle configuration et regardez ce qui est inscrit dans le terminal pour voir si tout est OK.

Étape 8

Vous devriez avoir le message suivant dans le terminal :

```
Tomcat started on port(s): 8080 (http) with context path ''
```

Ouvrez la page `http://localhost:8080`.

Si vous avez un message d’erreur, cela signifie peut-être que le port 8080 est déjà utilisé. Vous pouvez ajouter cette configuration dans le fichier *application.properties* pour démarrer sur le port 9090 :

```
server.port=9090
```

Étape 9

Nous allons maintenant créer notre première page. Commencez par créer les packages “model”, et “controller” (Spring respecte le modèle MVC) dans le package racine, celui contenant la classe avec la méthode main. **Attention** : le contrôleur a besoin de la vue pour fonctionner (pensez-y lors de vos tests).

Étape 10

Dans le package “controller”, créez la classe HelloWorldController.java avec le contenu suivant :

```
@Controller
public class HelloWorldController {

    @GetMapping("/greeting")
    public String greeting(@RequestParam(name="nameGET",
       required=false, defaultValue="World") String
       nameGET, Model model) {
        model.addAttribute("nomTemplate", nameGET);
        return "greeting";
    }
}
```

`https://e-gitlab.univ-lemans.fr/snippets/2`

Étape 11

Dans le dossier “resources”, créez un dossier “templates” et dedans, le fichier “greeting.html” :

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Blog</title>
    <meta http-equiv="Content-Type" content="text/html;
       charset=UTF-8" />
</head>
<body>
    <p th:text="'Bonjour ' + ${nomTemplate} + ' !'" />
</body>
</html>
```

`https://e-gitlab.univ-lemans.fr/snippets/3`

Étape 12

Relancez votre application avec le launcher et allez sur la page `http://localhost:8080/greeting`. Tentez ensuite l'URL suivante : `http://localhost:8080/greeting?name=ENSIM`

Étape 13

Relisez le code du contrôleur, aidez-vous de documentation sur internet, et répondez aux questions suivantes :

1. Avec quelle partie du code avons-nous paramétré l'url d'appel /greeting ?
2. Avec quelle partie du code avons-nous choisi le fichier HTML à afficher ?
3. Comment envoyons-nous le nom à qui nous disons bonjour avec le second lien ?

Ajoutez les réponses dans le README.

Étape 14

Nous allons maintenant activer la base de données H2. Dans le fichier "application.properties", ajoutez les lignes suivantes :

```
# Enabling H2 Console
spring.h2.console.enabled=true
```

Relancez l'application et allez sur l'URL : `http://localhost:8080/h2-console`, et appuyez sur "Connect". Si vous avez une erreur, essayer l'url suivante :

```
jdbc:h2:mem:testdb
```

Étape 15

Vous êtes sur l'interface de votre base de données **in-memory** !

Étape 16

Nous allons rajouter notre première classe du modèle MVC pour notre site : la classe *Address*. Dans le package "model" créé tout à l'heure, vous pouvez ajouter la classe Address suivante :

```
@Entity
public class Address {
    @Id
    @GeneratedValue
    private Long id;
    private Date creation;
```

```
    private String content;
}
```

`https://e-gitlab.univ-lemans.fr/snippets/4`

Rajoutez les méthodes *get* et *set* de toutes les propriétés via le bon raccourci de votre IDE.

Étape 17

Relancez-votre application, retournez sur la console de H2 : `http://localhost:8080/h2-console`. Avez-vous remarqué une différence ? Ajoutez la réponse dans le README.

Étape 18

Expliquez l'apparition de la nouvelle table en vous aidant de vos cours sur Hibernate, et de la dépendance Hibernate de Spring. Ajoutez la réponse dans le README.

Étape 19

Nous allons utiliser Spring pour ajouter des éléments dans la base de données. Pour ce faire, créer un fichier “data.sql” à côté du fichier “application.properties” et ajoutez-y le contenu suivant :

```
INSERT INTO address (id, creation, content) VALUES (1,
   CURRENT_TIMESTAMP(), '57 boulevard demorieux');
INSERT INTO address (id, creation, content) VALUES (2,
   CURRENT_TIMESTAMP(), '51 allee du gamay, 34080
   montpellier');
```

`https://e-gitlab.univ-lemans.fr/snippets/5`

Étape 20

Relancez l'application, retournez sur la console H2 : `http://localhost:8080/h2-console`. Faites une requête de type SELECT sur la table Adress. Voyez-vous tout le contenu de data.sql ? Ajoutez la réponse dans le README.

Étape 21

Nous allons maintenant créer une interface de type *Repository* pour accéder aux adresses. Une classe *Repository* permet d'accéder aux données de la base de données. Á côté de Adresse, créez une interface nommée “AdresseRepository” avec le code suivant :

```
@Repository
public interface AddressRepository extends
   CrudRepository<Address, Long> {

}
```

`https://e-gitlab.univ-lemans.fr/snippets/6`

Étape 22

Nous allons créer un autre contrôleur pour donner un accès aux adresses via l'url `http://localhost:8080/adresses`. Créez la classe **AddressController.java** dans le package des contrôleurs.

```
@Controller
public class AddressController {

    @Autowired
    AddressRepository addressRepository;

    @GetMapping("/adresses")
    public String showAddresses(Model model) {
        model.addAttribute("allAdresses",
           addressRepository.findAll());
        return "addresses";
    }
}
```

`https://e-gitlab.univ-lemans.fr/snippets/7`

Étape 23

Pouvez-vous trouver à quoi sert l'annotation **@Autowired** du code précédent sur internet ? Ajoutez la réponse dans le README.

Étape 24

Créez de plus le fichier “addresses.html” dans le dossier “templates” de tout à l’heure :

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Adresses</title>
    <meta http-equiv="Content-Type" content="text/html;
       charset=UTF-8" />
</head>
<body>

<h1>Les differentes adresses</h1>

<ul th:each="address: ${allAddresses}">
    <li th:text="${address.content}" />
</ul>

</body>
</html>
```

Étape 25

Vous êtes maintenant prêt et autonome pour la suite !

Étape 26

Et si vous rajoutiez le nom de la personne qui a fait la recherche d’une adresse ?

1. Ajouter l’auteur dans l’entité “Address.java”,
2. Dans le fichier data.sql, ajoutez la donnée,
3. Côté vue, utilisez la nouvelle donnée et affichez-la.

Étape 27

Pouvez-vous rajouter une navbar pour naviguer entre chaque pages ?

Étape 28

Utilisez les *Thymeleaf fragments* pour pouvoir déplacer la déclaration de la navbar à un seul endroit, et l’inclure dans chaque vues.

Étape 29

Regardez sur internet comment ajouter Bootstrap à votre projet.

Étape 30

Expliquez la méthode que vous avez utilisé pour ajouter Bootstrap dans le README.

2 TP 4

2.1 TP DM noté 2/2 : Utilisation d'API web avec Spring Boot

Ce TP est noté !

- Vous pouvez le terminer chez vous ;
- A rendre via : UMTICE (cours : POO API et outillages, clé : PAAP4) ;
- Date limite : 31 décembre.

Ce TP est dans la continuité du précédent, on va ajouter les fonctionnalités prévues dans le TD 5 :

- Utilisation de Spring Framework pour ajouter les fonctionnalités de base de données, utilisation d'API REST en ligne, génération de pages web ;
- Utilisation de H2 pour créer une base de données *in-memory* ;
- Utilisation de Thymeleaf pour faire des pages web ;
- **Récupération d'une adresse via un formulaire pour ensuite appeler une API donnant les coordonnées GPS** ;
- **Utilisation d'une API web pour récupérer la météo aux coordonnées GPS précises**.

Étape 1

Nous allons incorporer une nouvelle page (à l'url "adresse") contenant le formulaire du TD 5, permettant de demander une adresse.

(a) Créez un contrôleur, et ajoutez un formulaire permettant d'insérer l'adresse dans son template.

Solution :

```
<form action="/meteo" method="post">
  <div>
    <label for="address">Please input your
       adress:</label>
    <input name="address" id="address">
  </div>
  <div>
    <button>Get the weather at the given
       address!</button>
  </div>
</form>
```

(b) Mettez à jour la navbar pour qu'un lien aille sur cette nouvelle page "adresse".

Étape 2

Nous allons maintenant travailler sur les interactions entre pages.

(a) Créez un nouveau contrôleur, et son template (vide pour l'instant) pour l'url "meteo".

(b) Cette page "meteo" doit **être la cible** du formulaire de la page "adresse" que vous venez de créer, et sur validation du formulaire, doit envoyer le contenu vers le controller météo.

Modifier les attributs du formulaire précédent pour rediriger vers "meteo".

(c) Dans le contrôleur de la météo, ajoutez une méthode permettant de recevoir des appels POST (indice : annotation *PostMapping*).

(d) Récupérez la donnée du formulaire entrée par l'utilisateur dans le template. Dans la méthode du contrôleur de la météo, trouvez le bon paramètre permettant d'indiquer à Spring de valoriser l'adresse insérée dans le formulaire (aidez-vous de internet et de la documentation Spring).

(e) Une fois que le code compile, vérifiez qu'il fonctionne en mettant un point d'arrêt et en lançant le programme en mode debug.

Étape 3

À partir de l'adresse récupérée de l'utilisateur, faites un appel HTTP GET vers l'API Adresse de Etalab.

(a) Lisez la documentation Etalab Adresse : `https://geo.api.gouv.fr/adresse`

(b) Faites des tests d'URL via le navigateur (l'API fonctionne en GET, donc testable facilement via le navigateur).

(c) Regardez la structure de la réponse donnée par Etalab Adresse, et concevez l'objet Java de réponse (cf. documentation Spring : `https://spring.io/guides/gs/consuming-rest/`).

N'oubliez pas le proxy à éventuellement configurer si vous êtes sur le réseau de l'université : dans la partie "Astuces" dans ce document, il y a une aide sur la configuration du proxy pour Java.

Étape 4

Pour vérifier que vous obtenez bien les informations dans votre application, essayez de l'afficher dans le template Thymeleaf du contrôleur météo, ou via point d'arrêt en mode debug.

Étape 5

Une fois que vous arrivez à obtenir les coordonnées GPS, vous pouvez maintenant appeler l'API de MeteoConcept. Lisez la documentation pour comprendre comment spécifier la clé API, comment indiquer les coordonnées GPS, et comment est structuré la réponse. N'hésitez pas à faire des tests via votre navigateur.

(a) Lisez la documentation Meteo-Concept : `https://api.meteo-concept.com/documentation`

(b) Créez un compte pour avoir une clé API.

(c) Faites des tests d'URL via le navigateur (l'API fonctionne en GET, donc testable facilement via le navigateur).

(d) Regardez la structure de la réponse donnée par Meteo-Concept, et concevez l'objet Java de réponse (cf. documentation Spring)

Étape 6

Mettez la réponse à chacune de ces questions dans le README de votre projet :

— Faut-il une clé API pour appeler MeteoConcept ?
— Quelle URL appeler ?
— Quelle méthode HTTP utiliser ?
— Comment passer les paramètres d'appels ?
— Où est l'information dont j'ai besoin dans la réponse :
  — Pour afficher la température du lieu visé par les coordonnées GPS
  — Pour afficher la prévision de météo du lieu visé par les coordonnées GPS

Étape 7

Pour rendre le devoir-maison :

(a) Dans le README.md du projet, ajoutez un lien vers votre projet GitHub.

(b) Poussez votre code sur GitHub.

(c) Faites un zip de votre code source (lancez un “mvn clean” pour enlever les fichiers temporaires).

(d) Uploadez le zip sur UMTICE, dans la partie devoir-maison.