Université de Montpellier

Faculté des sciences

Master Informatique parcours IPS

Programmation Avancés HMIN327

HMIN327

MonumTour: Application de gestion de monuments liés à des célébrités

---

Etudiant:

- KACIOUI Arezki

Enseignants :

- DUROUX Patrice
- MOUGENOT Isabelle

Année universitaire : 2020/2021

Sommaire

I. Introduction 4
II. Analyse du projet 4
A. Besoins fonctionnels 4
III. Outils et concéption 5
A. Technologies utilisées 5
B. Conception 6
C. Architecture de l'application 8
IV. Travail réalisé 9
A. Mise en place du projet 9
B. Création des entités et mapping objet relationnel 10
C. Création des Repository 11
D. Création des services 12
E. Création des Controllers 13
F. Création des templates 14
G. Spring sécurity 15
V. Conclusion et perspectives 16

Tables des figures

Figure 1 : Logo des technologies utilisées 5
Figure 2 : Diagramme de classes 6
Figure 3 : Diagramme de use case 7
Figure 4 : Spring Boot flow architecture 8
Figure 5 : Architecture et arborescence du projet MonumTour 9
Figure 6 : Entité Celebrite 10
Figure 7 : fichier application.properties 11
Figure 8 : CelebriteRepository 11
Figure 9 : interface CelebriteService 12
Figure 10 : CelebriteService 12
Figure 11 : CelebriteController 13
Figure 12 : Exemple d'utilisation de Thymleaf 14
Figure 13 : Annotation @Secured de Spring Security dans un Controller 15
Figure 14 : exemple d'utilisation du dialecte sec de Spring Security dans une template 15
Figure 15 : L'administrateur (ROLE_SUPER_ADMIN) a accès a tout. 16
Figure 16 : Le voyagiste (ROLE_ADMIN) a accès à la gestion des entités hormis les utilisateurs. 16
Figure 17 : Le touriste (ROLE_USER) a accès à la consultation 16

I. Introduction

Dans ce projet, on souhaite concevoir et développer une application web permettant la gestion de monuments associés à des célébrités. L'application permettra de s'inscrire, d'avoir un rôle et de réaliser des taches spécifiques selon son rôle.

L'application doit respecter un cahier des charges donné, notamment le modèle de données fournis et une architecture MVC.

II. Analyse du projet

A. Besoins fonctionnels

- L'application doit offrir une possibilité de connexion avec différents types d'utilisateurs:
  - Un administrateur
  - Un voyagiste
  - Un touriste
- Selon le type d'utilisateur, celui-ci aura accès à des fonctionnalités spécifiques:
  - L'administrateur peut effectuer des consultations, des ajouts, des mises à jour et des suppressions sur l'ensemble des entités, y compris les utilisateurs.
  - Le voyagiste peut effectuer des consultations, des ajouts, des mises à jour et des suppressions sur l'ensemble des entités, hormis les utilisateurs.
  - Le touriste peut effectuer des consultations.
  - L'utilisateur anonyme aura accès à la barre de navigation et l'inscription.
- Effectuer des prétraitements tels que le calcul de distance entre monuments ou l'affichage sur une carte.
- Respecter le modèle de données qui est donnée.

III. Outils et conception

La première étape de ce projet a été le choix des technologies à utiliser ainsi qu'une partie conception.

A. Technologies utilisées

Afin de réaliser ce projet, un panel de technologies à été utilisé, on cite ici les plus importantes

- ⇨ MySQL : un système de gestion de bases de données relationnelles SQL open source développé et supporté par Oracle.
- ⇨ JAVA : langage de programmation orienté objet
- ⇨ SpringBoot : Framwork JAVA permettant de créer rapidement et facilement des applications.
- ⇨ Hibernate : Framwork open source gérant la persistance des objets en base de données relationnelle (ORM)
- ⇨ Thymleaf : « moteur de template, sous licence Apache 2.0, écrit en Java pouvant générer du XML/XHTML/HTML5. Thymeleaf peut être utilisé dans un environnement web (utilisant l'API Servlet) ou non web. Son but principal est d'être utilisé dans un environnement web pour la génération de vue pour les applications web basées sur le modèle MVC »[1].
- ⇨ Bootstrap : Boites à outils contenant des codes HTML et CSS utile à la création de design pour des applications web.
- ⇨ Javascript : « langage de programmation de scripts principalement employé dans les pages web interactives et à ce titre est une partie essentielle des applications web »[2].

Figure 1 : Logo des technologies utilisées

[1] https://fr.wikipedia.org/wiki/Thymeleaf
[2] https://fr.wikipedia.org/wiki/JavaScript

B. Conception

A la suite du choix des technologies, la conception d'un digramme de use case a été fait en respectant le diagramme de classes données et les besoins fonctionnels de l'application.

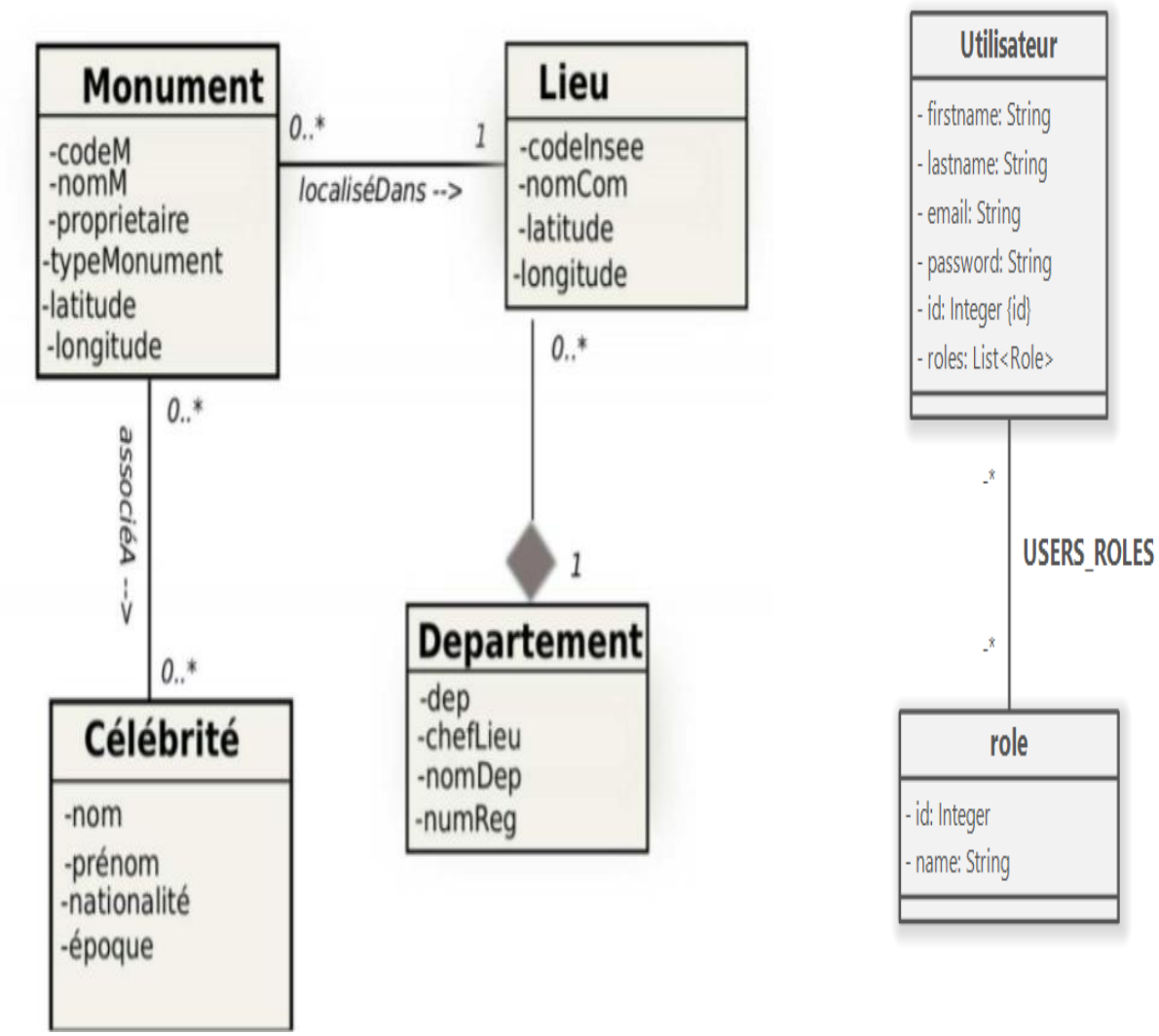

Figure 2 : Diagramme de classes

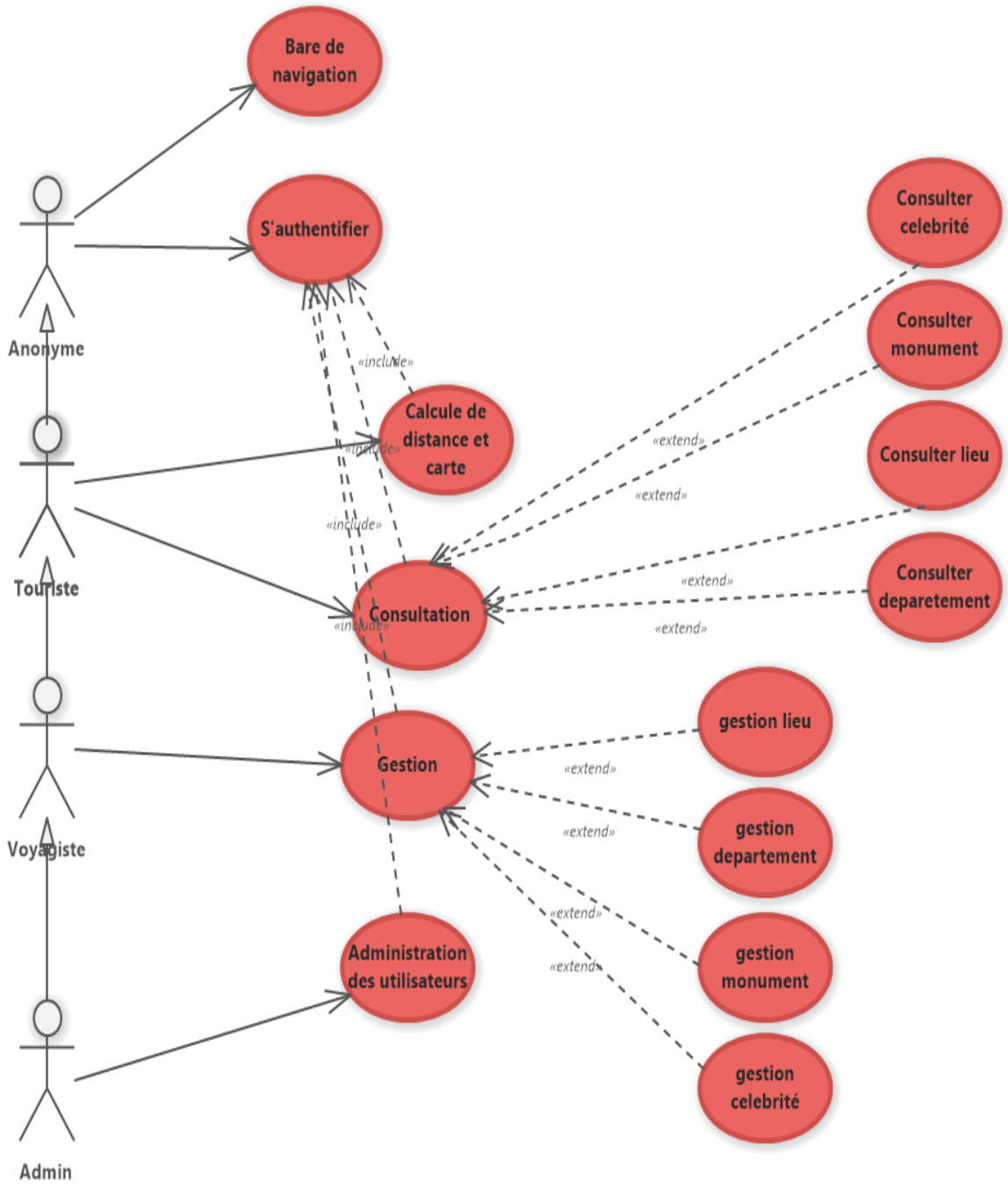

Figure 3 : Diagramme de use case

C. Architecture de l'application

Pour l'architecture du projet j'ai opté pour une architecture Spring Boot décrite dans la Figure 4.

L'application est composée de différentes briques interagissant entre elles.

Le coté **client**, représenté par les Templates Thymeleaf, permet l'interaction avec l'utilisateur.

Les requêtes de celui-ci sont ensuite transférées au **Controller** via le protocole http.

Le **Controller** traite les requêtes reçues en faisant appel aux différents **services** qui y sont injectés si nécessaire.

La couche **services** (Métier) est composée de deux sous-couches, des interfaces structurant la logique et des implémentations implémentant la logique. Cette couche fait appel au **Repository** qui étendent JPA afin d'interagir avec la base de données. Les models permettent de définir et de structurer les entités de l'application ainsi que la mapping objet relationnel avec les annotations Hibernate.

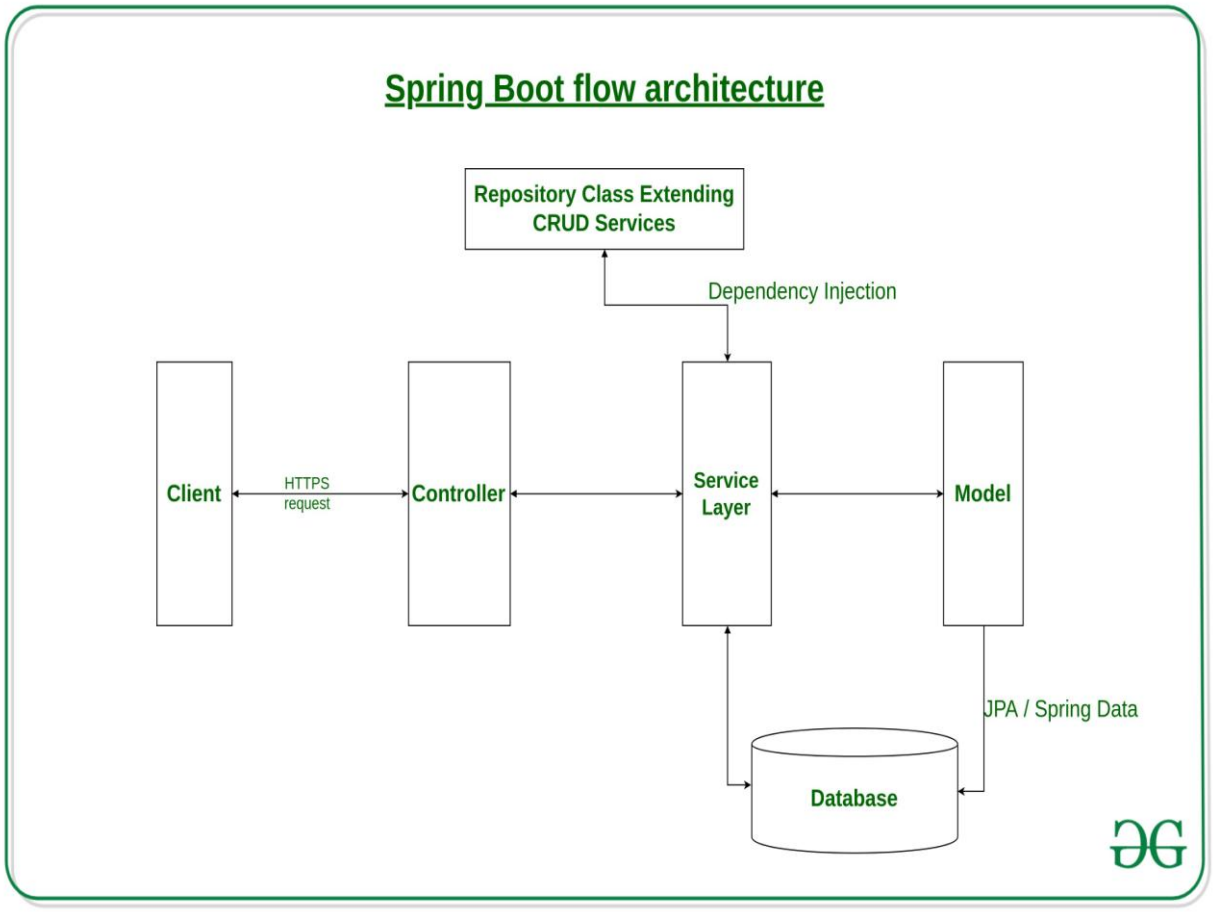

Figure 4 : Spring Boot flow architecture

IV. Travail réalisé

A. Mise en place du projet

Après avoir défini l'architecture de l'application, la partie de développement est entamée.

Tout d'abord, une mise en place de l'environnement technique est effectué (mise à jour des JDk's, installation des logiciels ; ...).

Par la suite, grâce à Spring initiliazer, j'ai généré le corps du projet contenant toutes les dépendances nécessaires à celui-ci :

- **Spring Boot DevTools :** Permet à l'application de redémarrer plus rapidement offrant une meilleure expérience de développement.
- **Spring Web :** Permet la conception d'application web avec un serveur Apache embarqué.
- **Thymeleaf :** Mouteur de template.
- **Spring Security :** Gestion des rôles, de l'authentification et la sécurité de l'application.
- **Spring Data JPA :** Permet la persistance des données, utilise Spring DATA et Hibernate.
- **MySQL Driver :** Connecteur à la base de données MySQL.

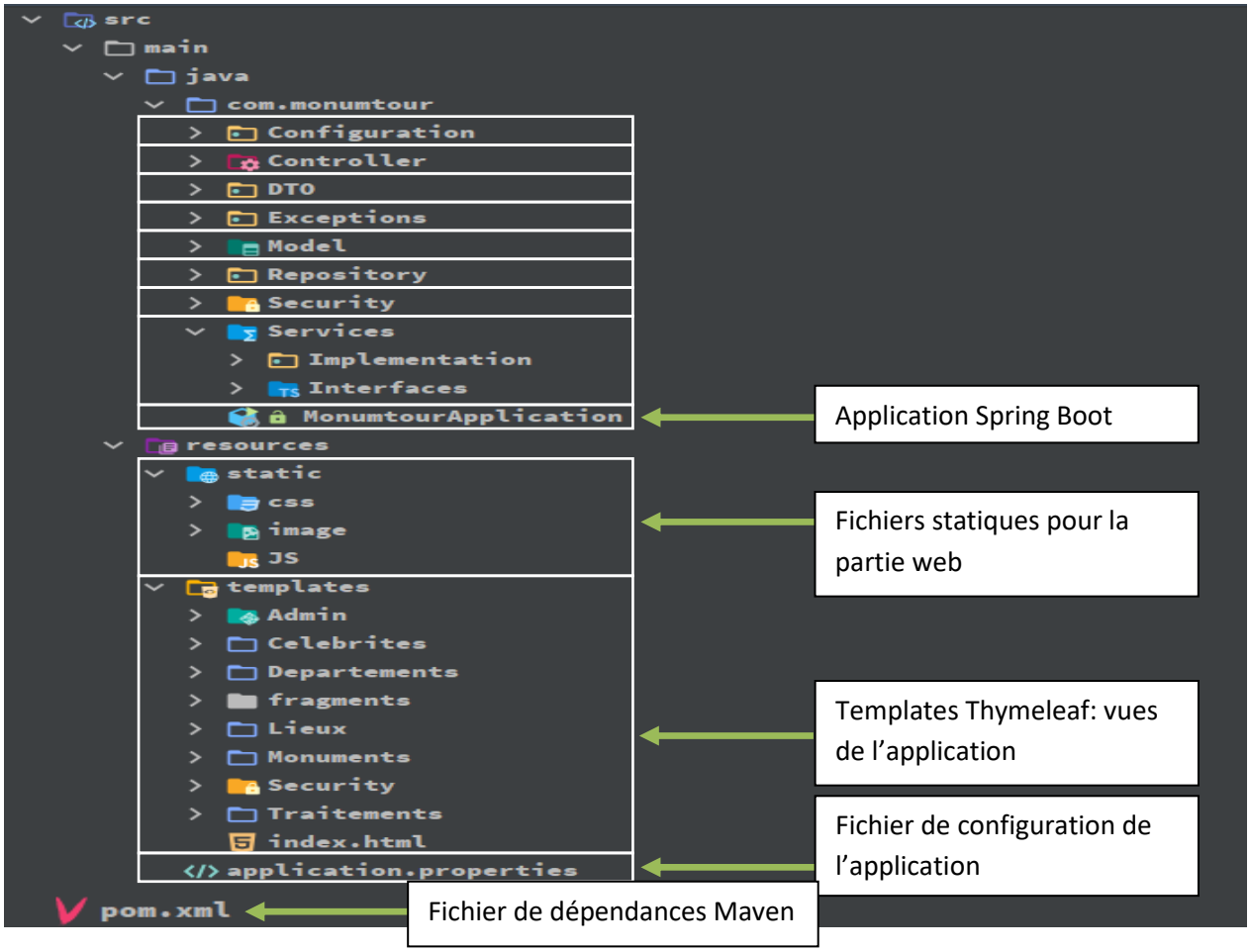

Figure 5 : Architecture et arborescence du projet MonumTour

Maven est le gestionnaire de projet choisi.

La Figure 5 montre l'architecture générale de mon projet.

A la racine du projet on retrouve un dossier **src** contenant l'ensemble des codes sources et le fichier **pom.xml,** fichier crucial dans le projet, permet la gestion des dépendances avec Maven.

Dans le dossier **src**, on retrouve deux dossiers, **src/main** et **src/test**. Le premier regroupe les codes sources de l'application et le second permet de faire des tests.

Le dossier **main**, contient deux sous-dossiers, **ressources** et **java.**

Ressources : on y retrouve les templates thymleaf, les fichiers statiques (CSS, images, Scripts JS) et le fichier de configuration application.properties.

Java : on y retrouve tout le code java organisé de sorte à respecter l'architecture Spring Boot.

Le fichier MonumTourApplication.java permet de lancer l'application.

B. Création des entités et mapping objet relationnel

Les entités (Model) sont des classes Java implémentant l'interface **Serializable**. La création de ces entités respecte le digramme de classe donnée en Figure 2.

Des constructeurs vides et avec arguments, des getters et des setters sont ajoutés à chaque entité.

Afin de réaliser le mapping objet relationnel des annotations ont été utilisées.

```java
@Entity
public class Celebrite implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @NotNull
    private Long numCelebrite;
    private String nom;
    private String prenom;
    private String nationalite;
    private String epoque;
    @ManyToMany
    @JoinTable(name="AssocieA",joinColumns= @JoinColumn(name="codeCelebrites"),
            inverseJoinColumns=@JoinColumn(name="codeM"))
    private Collection<Monument> monuments;
```

Figure 6 : Entité Celebrite

Par exemple dans le cas de l'entité Celebrite, Figure 6, **@Entity** permet de définir Celebrite comme étant une entité et de créer aisni une table ayant pour nom celebrite, si non spécifié avec l'annotation **@Table(name= « nom»).**

@Id : l'attribut numCelebrite sera considiré comme une clé primaire.

@GenratedValue(strategy = GenerationType.AUTO) : incrémentation automatique de l'ID.

@NotNull : ne peut pas être null.

@ManyToMany : permet de créer une relation de type many to many avec une autre entité, ici monument, au niveau de la table associeA sur les colonnes codeCelebrites et codeM.

```
spring.datasource.url= jdbc:mysql://localhost:3306/arezki?createDatabaseIfNotExist=true&useSSL=false&serverTimezone=UTC
spring.datasource.username = root
spring.datasource.password =

spring.jpa.show-sql= true
spring.jpa.properties.hibernate.format_sql= true

# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto= update
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect
#Port d'écoute
server.port = 8080
#filtre spring qui fait que toutes les requetes passe par spring secu
spring.security.user.name = admin
spring.security.user.password = 123
```

Figure 7 : fichier application.properties

Au niveau du fichier **application.properties**, Figure 7 , le paramètre **spring.jpa.hibernate.ddl-auto** est configuré sur **update**, cela signifie que **Hibernate** mettra à jour les tables au niveau de la base de données si elles existent ou les créera dans le cas ou elles n'existent pas.

C. Création des Repository

Après la création des différentes entités, on a procédé à la création des Repository pour chaque entité. Le repository est une interface implémentant JPARepository. Par défaut, toutes les méthodes CRUD sont ainsi héritées à chaque repository. D'autres méthodes plus spécifique peuvent aussi être définies en utilisant des annotations **@Query** avec du JPQL. (Figure 8)

```
package com.monumtour.Repository;

import ...

public interface CelebriteRepository extends JpaRepository<Celebrite, Long> {


    @Query("select c from Celebrite c where c.nom like:nom")
    List<Celebrite> findCelebriteByNom(@Param("nom") String nom);
```

Figure 8 : CelebriteRepository

D. Création des services

Dans cette partie, le package services est séparé en deux sous-packages, interfaces et implémentations.

Dans Interfaces on défini l'ensemble des méthodes dont a besoin pour chaque entité.

Dans Implémentation, les interfaces sont implémentées, on utilise le repository, injecté, afin d'interagir avec la base de données. L'annotation **@Service** doit être ajouté avant la définition de la calasse.

```java
public interface ICelebriteService {
    //Crud operation
    //GET
    //by nom
    public List<Celebrite> findCelebriteByNom(String nom);
    //Celebrite par monument
    public Collection<Celebrite> getCelebriteParMonument(String codeMonument);
    //ALL
    public Collection<Celebrite> getAllCelebrites();
    //ADD
    public Celebrite addCelebrite(Celebrite c);
    //Ajouter Celebrite a un monument
    public void addCelebriteToMonument(Long numCelebrite, String codeM);
    //UPDATE
    Celebrite update(Celebrite c);
    Celebrite save(Celebrite c);

    //delete
    public void deleteCelebrite(Long numCelebrite);
    //PerPAge
    Page<Celebrite> getAllCelebritesParPage(int page, int size);
    Celebrite getCelebriteParId(Long id);
}
```

Figure 9 : interface CelebriteService

```java
@Service
public class CelebriteService implements ICelebriteService {

    @Autowired
    private CelebriteRepository celebriteRepository;
    @Autowired
    private MonumentRepository monumentRepository;
    @Override
    public List<Celebrite> findCelebriteByNom(String nom) { return celebriteRepository.findCelebriteByNom(nom); }

    @Override
    public Celebrite addCelebrite(Celebrite c) {
        celebriteRepository.save(c);
        return c;
    }
```

Figure 10 : CelebriteService

E. Création des Controllers

Un Controller est crée pour chaque entité.

Le Controller est une classe Java qui utilise l'annotation **@Controller,** celle-ci permet de signifier à Spring que cette classe est un Controller. Le Controller est un composant essentiel dans le model MVC. Il permet la définition des différentes routes contenues dans l'application avec l'annotation **@RequestMapping** et le traitement des requêtes qui y sont émises. Et le renvoie de la vue appropriée.

Dans chaque Controller sont injectés les services nécessaires afin de pouvoir interagir avec les entités. Cette injection se fait avec l'annotation **@Autowired**.

```java
@Controller
public class CelebriteController {

    @Autowired
    private ICelebriteService celebriteService;
    @Autowired
    private IMonumentService monumentService;

    @Secured(value = { "ROLE_ADMIN", "ROLE_USER" })
    @RequestMapping(value = "/allCelebrite")
    public String allCelebrite(ModelMap modelMap){
        Collection<Celebrite> celebrites = celebriteService.getAllCelebrites();
        modelMap.addAttribute( attributeName: "celebrites", celebrites);
        return "Celebrites/allCelebrites";
    }
```

Figure 11 : CelebriteController

La Figure 11 représente le CelebriteController, Controller qui gère toutes les requêtes en lien avec les célébrités. Lorsque l'utilisateur effectue une requête de type GET sur la route http://monumTour.com/allCelebrites, le Controller avec sa méthode allCelebrite va récupérer toutes les célébrités en utilisant le celebriteService injecté, et les stocker dans une variable célébrités. Par la suite le Controller va renvoyer la Template Celebrites/allCelebrites.html ainsi que la liste de toutes les célébrités avec l'utilisation de l'objet modelMap et de sa méthode attribute().

F. Création des templates

Thymleaf a été utilisé afin de gérer les templates et de rendre leurs contenu dynamique.

Thymleaf utilise un dialecte spécifique permettant d'effectuer diverses opérations telles que l'itération ou l'ajout de condition.

Grace au model renvoyé par le Controller, Thymleaf peut adapter le contenu HTML.

Par exemple dans Figure 12 grâce à l'itération fourni par thymleaf, on peut itérer sur une liste de célébrités et afficher les informations voulus pour chaque célébrités au niveau d'un tableau.

Il permet aussi l'intégration de Spring security afin de gérer l'affichage selon le rôle.

```
        <th>Monuments associées</th>
        <th sec:authorize="hasRole('ROLE_ADMIN')">Edition</th>
        <th sec:authorize="hasRole('ROLE_ADMIN')">Suppression</th>
    </tr>
    <tr th:each="c:${celebrites}">
        <td th:text="${c.getNumCelebrite()}"></td>
        <td th:text="${c.getNom()}"></td>
        <td th:text="${c.prenom}"></td>
        <td th:text="${c.nationalite}"></td>
        <td th:text="${c.epoque}"></td>
        <td> <p th:each="m:${c.getMonuments()}" th:text="${m.getNomM()}"></p></td>
        <td sec:authorize="hasRole('ROLE_ADMIN')"><a class="btn btn-success"
                th:href="@{updateCelebrite(id=${c.numCelebrite})}">Editer</a></td>
        <td sec:authorize="hasRole('ROLE_ADMIN')"><a class="btn btn-danger"
                onclick="return confirm('Confirmer suppression ?')"
               th:href="@{deleteCelebrite(id=${c.numCelebrite})}">Supprimer</a>
        </td>
    </tr>
```

Figure 12 : Exemple d'utilisation de Thymleaf

Le style et le design de l'application a été fait à l'aide de CSS et de Bootstrap.

G. Spring sécurity

Spring Security permet de gérer les rôles, l'accès aux routes et l'authentification

La configuration de Spring Security se fait au niveau du fichier SecurityConfiguration.java contenu dans la package Security. Dans ce fichier on définit les stratégies de sécurité de l'application.

Par la suite, on procède à la sécurisation de l'application au niveau des Controllers avec l'ajout de l'annotation **@Secured** et la spécification des rôles ayant accès à la route (Figure 13).

```
@Secured(value = { "ROLE_ADMIN", "ROLE_USER" })
```

Figure 13 : Annotation @Secured de Spring Security dans un Controller

Et au niveau des templates avec l'utilisation de dialecte sec de Spring Security.

```
    <div class="dropdown-menu" aria-labelledby="navbarDropdownDepartement">
        <a class="dropdown-item" href="#" th:href="@{/allDepartements}">Liste</a>
        <a sec:authorize="hasRole('ROLE_ADMIN')" class="dropdown-item" href="#" th:href="@{/addDepartement}">Ajout</a>
    </div>
</li>

<li class="nav-item" sec:authorize="isAnonymous()">
    <a class="nav-link " href="#" th:href="@{/registration}">S'inscrire</a>
</li>
<li class="nav-item" sec:authorize="isAnonymous()">
    <a class="nav-link " href="#" th:href="@{/login}">Se connecter</a>
</li>

<li class="nav-item" sec:authorize="isAuthenticated()" >
    <a class="nav-link" href="#" th:href=" @{/logout} " >Se déconnecter</a>
</li>
<li class="nav-item" sec:authorize="isAuthenticated()">
    <p class="nav-link" sec:authentication="name"></p>
</li>
<li class="nav-item" sec:authorize="hasRole('ROLE_SUPER_ADMIN')">
    <a class="nav-link" href="#" th:href="@{/admin}">Administration</a>
</li>
```

Figure 14 : exemple d'utilisation du dialecte sec de Spring Security dans une template

Dans les figures suivantes on voit l'effet de Spring security sur le rendu.

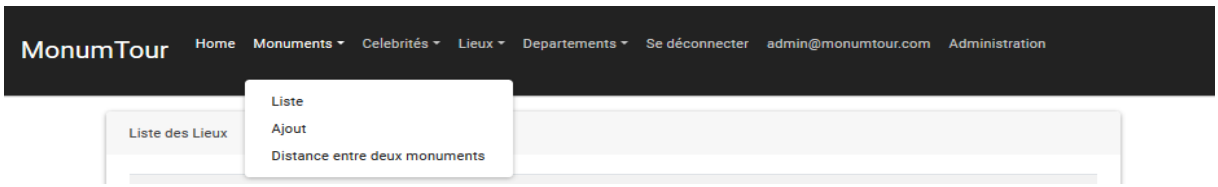

Figure 15 : L'administrateur (ROLE_SUPER_ADMIN) a accès a tout.

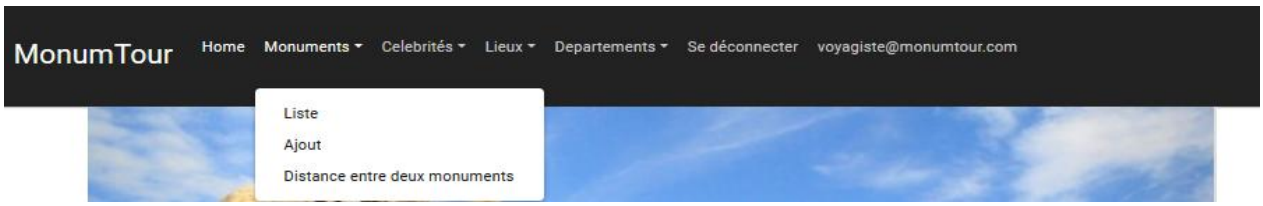

Figure 16 : Le voyagiste (ROLE_ADMIN) a accès à la gestion des entités hormis les utilisateurs.

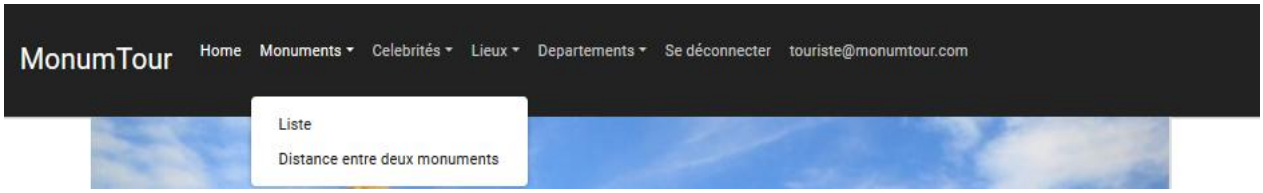

Figure 17 : Le touriste (ROLE_USER) a accès à la consultation

V. Conclusion et perspectives

Ce projet a été une expérience enrichissante tant au niveau technique que théorique, en effet, il m'a permis de m'initier à la conception d'application MVC avec Java et le framwork Spring Boot toute en respectant un cahier des charges de départ, de mieux comprendre la conception d'un modèle de données ainsi, le mapping objet relationnel et les différentes couches de ce type d'application.

En perspective, d'autres fonctionnalités peuvent être ajoutées à l'application telles que la proposition d'itinéraires par les voyagistes aux touristes, un affichage détaillé des monuments et des autres entités et un système de pagination.