UNIVERSITÉ
DE MONTPELLIER

RAPPORT PROJET PROGRAMMATION AVANCEE

Développement d'une petite application web avec JAVA EE

ALIOUACHE Mohammed

Informatique pour les sciences

2017/2018

INTRODUCTION:

Java EE permet d'ajouter à java et java SE de départ de nombreuses bibliothèques qui vont donner accès à de nombreuses fonctionnalités puissantes, utiles et faciles à utiliser. L'objectif majeur de java entreprise édition et le développement d'applications web robustes distribuées, déployées et exécutées sur un serveur d'applications(Openclassrooms).

Les développeurs font souvent face au fait de trouver le meilleur moyen d'optimiser l'architecture d'un projet et sa façon de tourner. Le framework Spring par exemple fait en sorte d'améliorer le développement d'une application sur différents niveaux (persistance, sécurité…). Il existe aussi le framework Struts.

Maven, géré par Apache software foundation, est un outil de gestion de projets java et surtout l'Entreprise Edition. Son objectif est d'assurer le bon ordre de production d'un projet. Il procure la possibilité d'injecter des dépendances très facilement en utilisant un pom.xml qui rassembles les librairies utilisées dans le projet.

AVANT PROPOS:

Durant le cours de programmation avancée IPS-M2, chaque étudiant doit produire une petite application web en respectant les contraintes données. L'application permettra de créer une base de données contentant des départements avec ses lieux et chaque lieu aura des monuments dont on associera des célébrités. Les données doivent être présentées sous formes de pages web visualisables et faciles à gérer. Une petite remarque est le fait que je n'ai pas fait quelques fonction sur tous les pojos, par exemple, la validation des formulaires n'est disponible que dans le formulaire d'inscription des membres. Donc pour le faire sur le formulaire d'ajout de département, il faudra juste suivre la même procédure.

CONFIGURATION GENERALE:

Le modèle est constitué de différents pojos (classes classiques de java), ces derniers sont mappés à l'aide des annotations de Hibernate, le mapping va définir la méthode de l'enregistrement des instances dans la base de données de manière automatique en se servant de l'Entity Manager. La base de données utilisée dans le projet est Mysql. Les vues sont des Jsp qui génèrent des pages html en utilisant les données du modèle. La partie contrôleur est gérée par Spring qui mappera les url est exécutera la fonction correspondante. La partie authentification et sécurité (interception) est prise en charge avec Struts. Toutes les dépendances sont mentionnées dans le pom.xml du projet qui télécharge automatiquement depuis internet ce qu'il faut pour le bon fonctionnement des librairies. Il faut ajouter les dépendances de Maven dans les propriétés du projet.

PARTIE MODELE:

Voici à quoi ressemble un pojo de notre projet:

User.java

```
14  import javax.persistence.PrePersist;
15
16  @Entity
17  @Table(name="USER",uniqueConstraints = @UniqueConstraint(columnNames = "email"))
18  public class User implements Serializable{
19      /**
20       *
21       */
22      private static final long serialVersionUID = 1L;
23      @GeneratedValue(strategy = GenerationType.AUTO)
24      private int idUser;
25      @NotEmpty(message="Veuillez saisir votre nom")
26      @Column(name="nom", length=16, nullable=false)
27      private String nom;
28      @Column(name="prenom", length=16, nullable=false)
29      @NotEmpty(message="Veuillez saisir votre prénom")
30      private String prenom;
31      @Column(name="sexe", length=1)
32      private String sexe;
33      @Id
34      @Column(name="email", length=32, nullable=false)
35      @NotEmpty(message="Veuillez saisir une adresse mail")
36      @Pattern(regexp = "^[a-zA-Z]+[a-zA-Z0-9\\._-]*[a-zA-Z0-9]@[a-zA-Z0-9][a-zA-Z0-9\\._-]*
37      private String email;
38      @NotEmpty(message="Veuillez mettre un mot de passe")
39      @Column(name="password", length=32, nullable=false)
40      @Size(min = 8,message="Le mot de passe doit avoir au minimum 8 charactères")
41      private String password;
42      @Column(name="nationalite", length=16)
43      private String nationalite;
```

Différentes annotations nous sont offerte et nous permettent de manipuler le pojo comme on le souhaite (sous quel nom l'attribut va être enregistré dans la base de données, les contraintes associées à cet attribut, la longueur minimale ou maximale, …). Par exemple, l'annotation **@Pattern** nous offre la possibilité d'utiliser les expressions régulières pour vérifier la syntaxe des attributs avant de les ajouter dans la base de données, il existe aussi l'annotation **@Email** qui vérifie que l'attribut est sous forme d'une email.

Voici la configuration simple de hibernate qui permet de prendre en charge la persistance:

persistence.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>

<persistence version="2.0"
xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
        <persistence-unit name="persistence-tag" transaction-type="RESOURCE_LOCAL">
                <provider>org.hibernate.ejb.HibernatePersistence</provider>
                <properties>
                <property name="hibernate.hbm2ddl.auto" value="update"/>
                <property name="hibernate.show_sql" value="true" />
                </properties>
        </persistence-unit>
</persistence>
```

Et cette configuration est appelée dans la configuration de Spring comme suit, Et aussi il faut passer les identifiants de la base de données, le nom de la base de donnée et le jdbc.Driver (Mysql, Oracle ...):

spring.xml

```xml
    <!--  -->
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataS
    <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
    <property name="url" value="jdbc:mysql://localhost:3306/javaEE"></property>
    <property name="username" value="root" />
    <property name="password" value="mercilam" />
</bean>
    <!--  -->
<bean id="persistenceUnitManager" class="org.springframework.orm.jpa.persistenceuni
    <property name="persistenceXmlLocations">
        <list>
            <value>classpath*:configs/persistence.xml</value>
        </list>
    </property>
    <property name="defaultDataSource" ref="dataSource"></property>
</bean>
```

PARTIE CONTOLLEUR:

Le contrôleur est en charge de mapper les actions à partir de leurs URL et exécuter une fonction correspondante. Pour mettre en place un contrôleur dans Spring, il faut au minimum définir une classe Controller qui va être annoté avec **@Controller** importé de org.springframework.stereotype .

Il faudra aussi configurer quelques points :

- La balise <mvc:ressources/> permet de définir un mapping pour les ressources : les css, js, image …
- Dans la classe org.springframework.web.servlet.view.**InternalResourceViewResolver,** il faudra configurer la manière d'interprétation des return des contrôleurs.
- La balise <context:component-scan/> définie le package ou se trouve les contrôleurs.
- Voici la configuration associée en code :

controllers-config.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:beans="http://www.springfr
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc http://www.springframewc
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/
        http://www.springframework.org/schema/context http://www.springframework.org/schem
    xmlns:mvc="http://www.springframework.org/schema/mvc">

    <annotation-driven/>

    <mvc:resources mapping="/img/**" location="/images/" />

    <!--pour guider le return des controllers vers les bons fichiers jsp -->
    <beans:bean
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <beans:property name="prefix" value="/pages/" />
        <beans:property name="suffix" value=".jsp" />
    </beans:bean>
    <context:component-scan base-package="com.projetjee.controllers" />
</beans:beans>
```

Une fois cette configuration faite, il faudra l'inclure dans le fichier web.xml qui se trouve dans le WAB-INF de webapp/ de la manière suivante :

web.xml

```xml
    <servlet>
        <servlet-name>controllerServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/controllers-config.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>controllerServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
```

Le contrôleur Spring propose plusieurs fonctionnalités pour couvrir les besoin coté web :

- Le contrôleur instancie l'interface du Dao et avec l'annotation **@AutoWired** il permet de remplir l'instance avec les bons paramètres pour prévenir les NullPointerException.
- l'annotation **@RequestMapping** permet de dire que la fonction qui suit est celle qui s'exécute une fois son attribut « value » est passé en url. Son attribut « params » déclare les paramètres attendus passés à l'url mappée, ses paramètres pourront être récupérés à l'aide de l'annotation **@RequestParam** dans les arguments de la fonction.
- Les méthodes exécutées retournent des chaines de caractères qui à l'aide de la configuration précédente retrouve la bonne page jsp correspondante.
- La classe Model et grâce a sa méthode addModelAttribute nous permet de passer des objets à nos page jsp, le contenu de ces objets peut être afficher dans la page web grâce à

l'interpolation en mettant l'objet dans **${monObjet}.** Il permet aussi de mettre en place un binding entre les champs html avec une instance d'un pojos en utilisant des taglibs

- à l'aide de **@Valid** et la classe org.springframework.validation.**BindingResult** on peut vérifier les champs remplis dans les formulaires qu'ils correspondent bien aux spécifications mentionnées dans les pojos et aucune contrainte n'est violée avant de passer à la partie sauvegarde dans la base de données
- plusieurs contrôleurs peuvent être déclarés dans le package si une organisation s'impose.

PARTIE VUE :

La partie vue est gérée par les pages jsp qui elles aussi proposent de nombreuses fonctionnalités permettant de venir au besoin coté web. En important des taglibs Struts ou Spring par exemple, on peut étendre les balises html en trouvant de nouveaux attributs.

- par Exemple pour compléter la partie vue afin de binder des variables, on trouvera le nouveau attribut de la balise personnalisée de Spring <Spring:form **ModelAttribute=...**> qui pointe vers un nom d'une instance de pojo déjà définie dans le contrôleur et ajouté dans le modèle avant d'être passé dans la jsp. Ainsi, en utilisant la balise <Spring:input **path=...**> on peut définir un chemin vers les attributs de l'objet du modelAttribute.
- il nous est aussi proposé d'utiliser les conditions et les boucles dans les jsp, ce qui nous permet de faire d'intéressantes manœuvres en toute simplicité

Importer le taglib de la jstl

<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="core"%>

Ensuite faire une condition

<core:**if** test=...>

…

…Ici le contenu à afficher si la condition dans le test est vérifiée

…

</core:**if**>

Ou faire une boucle par exemple :

<c:**forEach** items="${AllMonuments}" var="unMonument">

…

…Ici contenu de la boucle

…

</core:**forEach**>

Une autre fonctionnalité très pratique dans le but d'organiser le code et éviter les répétitions. Il y'a possibilité de définir des Templates qui sont fixe et de les appeler dans nos pages, un simple

exemple, mettre en place un header et un footer et ensuite les inclure dans toutes les pages. Il suffit de faire ainsi :

home.jsp

```
<%@ include file="../includes/top.jsp" %>
    <div class="container">
        <h1>Welcome To JAVA EE Project IPS 2018</h1>
    </div>
<%@ include file="../includes/bottom.html" %>
```

En utilisant les includes les pages deviennent très simples et facile à modifier.

PARTIE SECURITE ET INTERCEPTION :

Malgré que Spring dispose d'un spring security facile à déployer et très efficace, j'ai préféré utiliser Struts pour intercepter les url et ainsi filtrer le passage aux pages. J'ai téléchargé l'exemple existant dans www.journaldev.com et je l'ai inclus dans le projet en l'adaptant à ce dernier. Voici ce qu'il faut au minimum pour faire marcher les interceptions de Struts :

- un fichier de configuration de struts est ajouter dans le dossier des ressources juste à côté de spring.xml on y'ajoute struts.xml, voici son contenu :

struts.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
    "http://struts.apache.org/dtds/struts-2.3.dtd">
<struts>
    <constant name="struts.convention.result.path" value="/"></constant>

    <package name="user" namespace="/" extends="struts-default">
        <interceptors>
            <interceptor name="authentication"
                class="com.journaldev.struts2.interceptors.AuthenticationInterc
            <interceptor-stack name="authStack">
                <interceptor-ref name="authentication"></interceptor-ref>
                <interceptor-ref name="defaultStack"></interceptor-ref>
            </interceptor-stack>
        </interceptors>

        <default-interceptor-ref name="authStack"></default-interceptor-ref>

        <global-results>
            <result name="login" type="redirect">index.jsp</result>
        </global-results>
```

- La partie interceptor utilise celle par défaut de struts
- le global-result permet de rediriger la page vers un résultat si le retour de la fonction ne correspond à aucun traitement

- l'exemple qui suit va utiliser l'option action de struts pour montrer l'action à exécuter en cas ou l'attribut name est appelé. Une petite remarque, struts fais toujours appel à la fonction intercept avant de continuer sur l'action

```
<action name="login" class="com.journaldev.struts2.actions.LoginAction">
    <interceptor-ref name="defaultStack"></interceptor-ref>
    <result name="success">/home</result>
    <result name="input">login.jsp</result>
</action>
```

- la classe LoginAction a une fonction qui s'appelle execute () qui est notre action à faire, cette fonction retourne une chaine de caractères, et suivant le String retourné, on le mappe dans les options result de action dans le fichier de configuration.
- Exception dans le projet, la base de données est consulté directement depuis la fonction execute de la classe LoginAction au lieu d'appeler le DAO, car il n'était pas possible d'instancier le dao depuis cette classe (pas ajouté dans la configuration). Voici comment vérifier la bonne authentification des utilisateurs :

LoginAction.java

```
22      public String execute() throws Exception{
23          Class.forName("com.mysql.jdbc.Driver");
24          Connection con = DriverManager.getConnection(
25                  "jdbc:mysql://localhost/javaEE", "root", "mercilam");
26          try {
27
28              try {
29                  Statement st = con.createStatement();
30                  ResultSet value = st
31                          .executeQuery("SELECT * FROM USER WHERE email='"+user.getEmail()+"';");
32                  if (value.next()){
33                      String pass=value.getString("password");
34                      if (pass.equals(user.getPassword())){
35                          user.setUserName(value.getString("nom")+" "+value.getString("prenom"));
36                          user.setNationalite(value.getString("nationalite"));
37                          user.setNom(value.getString("nom"));
38                          user.setPrenom(value.getString("prenom"));
39                          user.setSexe(value.getString("sexe"));
40                          user.setTypeUser(value.getString("type"));
41                          sessionAttributes.put("USER", user);
42                          return SUCCESS;
43                      }
44                      return INPUT;
45                  }
46                  else{
47                      return INPUT;
48                  }
49              } catch (SQLException ex) {
50                  System.out.println("SQL statement is not executed!" + ex);
51              }
52              con.close();
53          } catch (Exception e) {
54              e.printStackTrace();
```

- Dans la fonction intercept (), la session utilisateur est récupérée. Donc si aucun utilisateur n'est connecté, il sera automatiquement redirigé vers une page d'authentification. Si un utilisateur est déjà connecté, son rôle va être vérifié. Ici soit il sera connecté en tant que touriste ou bien en tant que voyagiste. Le voyagiste étant avec plus d'accès, si un touriste

essaye d'accéder à une page que seul un voyagiste peu consulter, il sera très rapidement informé qu'il n'a pas les droits. Le code suivant en montre comment est implémentée cette vérification :

AuthenticationInterceptor.java ×

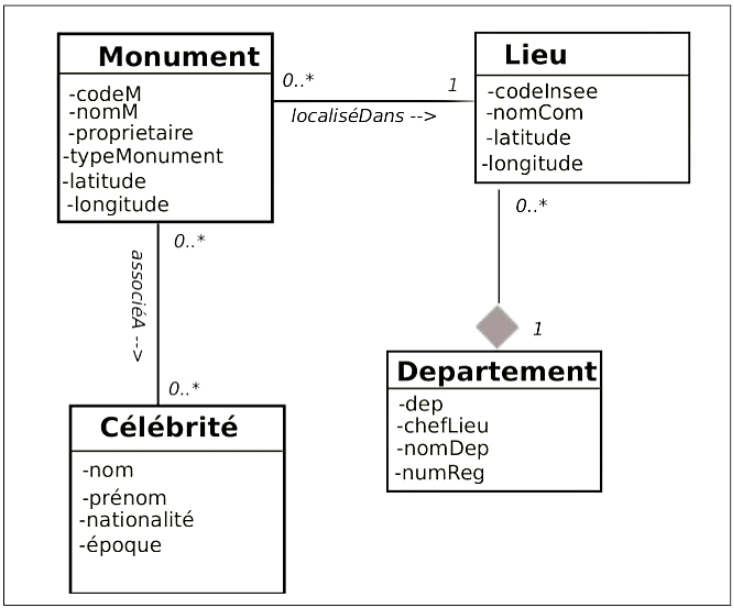
```
22
23      public String intercept(ActionInvocation actionInvocation)
24              throws Exception {
25          System.out.println("intercepting ...");
26          Map<String, Object> sessionAttributes = actionInvocation.getInvocationContext().getSessi
27
28          User user = (User) sessionAttributes.get("USER");
29          if(user == null){
30              System.out.println("No user connected!! redirecting...");
31              return Action.LOGIN;
32          }else{
33              System.out.println("user connected!! checking for authorisation...");
34          tour.add("unMonumentd");
35          tour.add("home");
36          tour.add("logout");
37          tour.add("mapMonuments");
38          tour.add("listMonuments");
39          String myInterceptedAction=actionInvocation.getProxy().getActionName();
40
41          if (user.getTypeUser().equals("touriste")){
42              if (!tour.contains(myInterceptedAction)){
43                  System.out.println("Access denied.");
44                  return "forbidden";
45              }
46          }
47          System.out.println("Access granted.");
48
49          Action action = (Action) actionInvocation.getAction();
50          if(action instanceof UserAware){
51              ((UserAware) action).setUser(user);
52          }
53          return actionInvocation.invoke();
54          }
55      }
```

- Aussi pour intercepter les url, il faut aussi ajouter struts au web.xml de la manière suivante :

web.xml ×

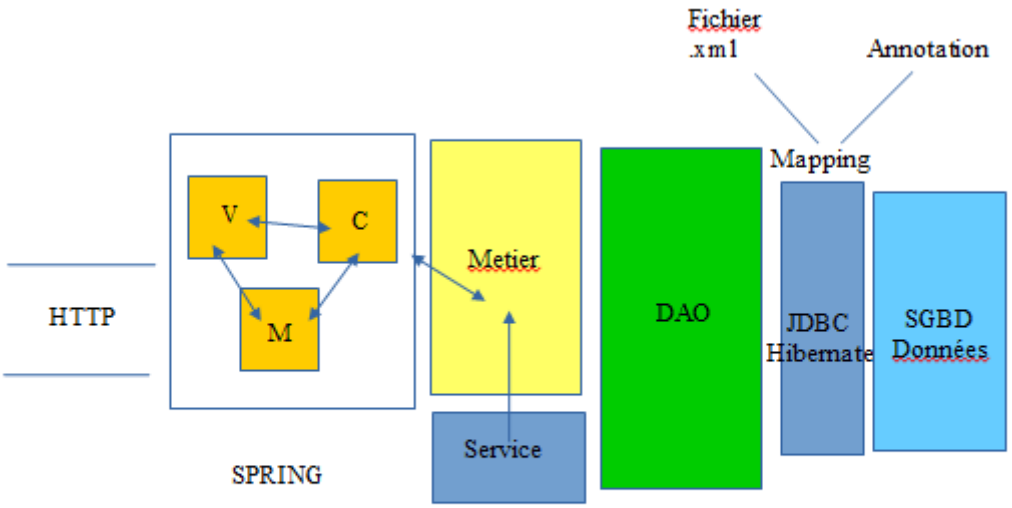
```
27      <!-- pour les interceptions de struts -->
28      <filter>
29          <filter-name>struts2</filter-name>
30          <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</f
31      </filter>
32
33      <filter-mapping>
34          <filter-name>struts2</filter-name>
35          <url-pattern>/*</url-pattern>
36      </filter-mapping>
37  </web-app>
```

Voici le diagramme UML imposé et respecté :

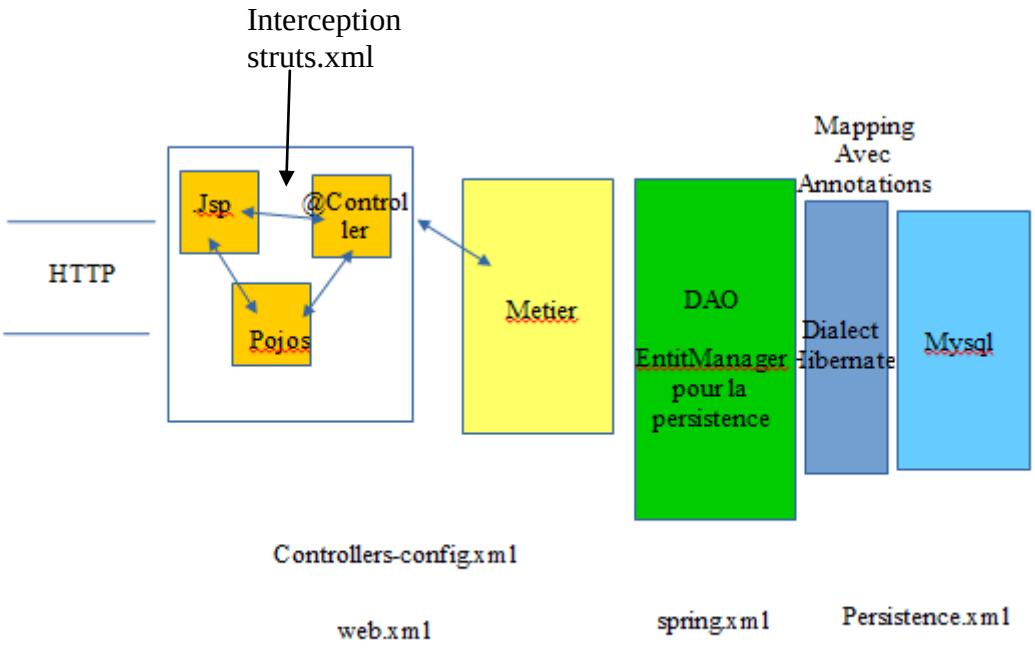


Donc, des pojos représentant les différentes classes ont été créées. Aussi pour l'association AssocieA qui est un pojo aussi.

L'architecture Maven-JEE-Spring à respecter est la suivante :



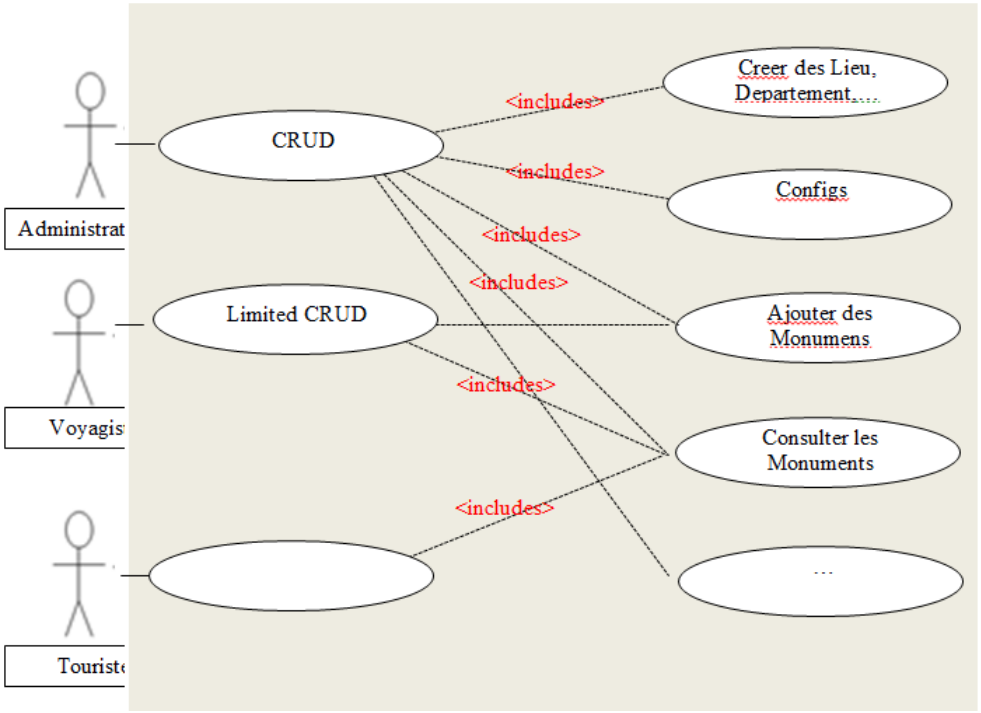
Je reprends le diagramme précédant pour décrire les options prise pour le projet.



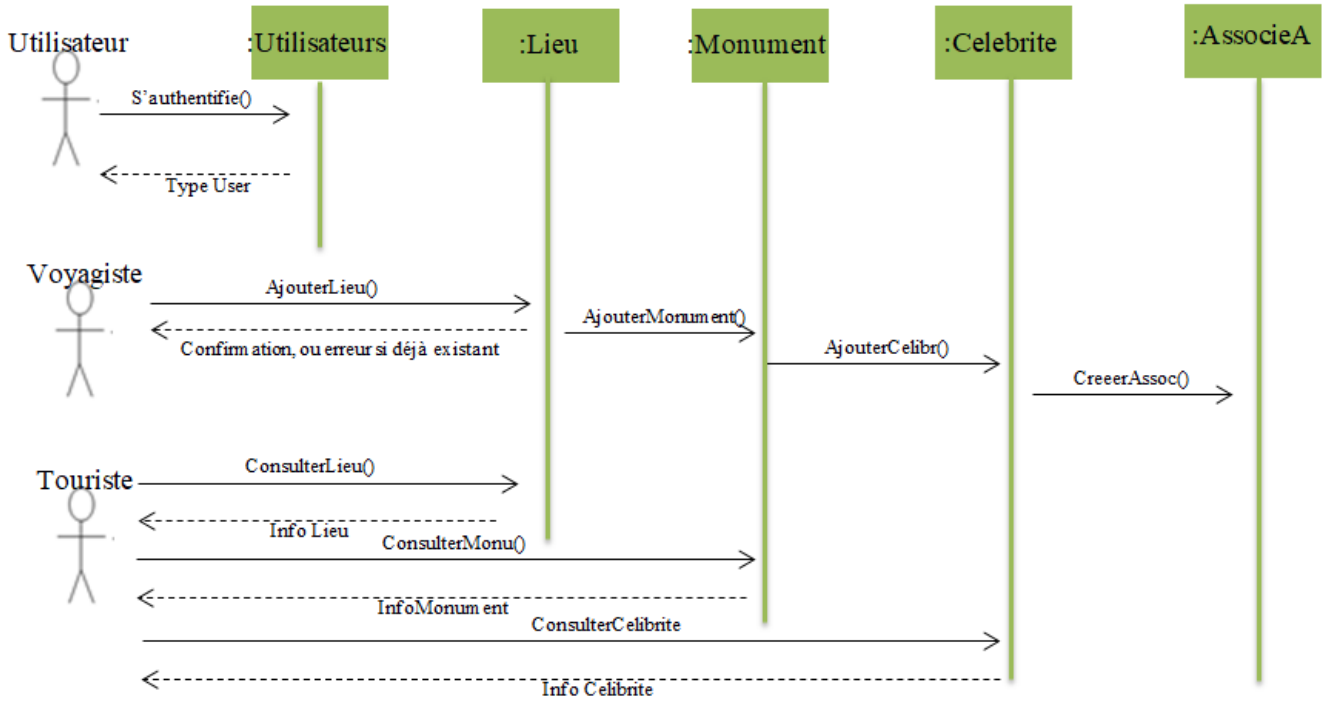
En partant de la droite vers la gauche en passant les couches:

- Dans la base de données, j'ai les tables décrites précédemment dans le diagramme uml. Les tables ne sont pas encore peuplées mais elles le sont au fur et à mesure de création de nouvelles pages JSP qui ajoutent des tuples dans la base de données.
- Dans le mapping j'ai utilisé Hibernate et les annotations
- Dans le dao j'utilise l'EntityManager pour consulter la base de données
- La couche Métier est comme une de passage ou de sécurité
- L'architecture MVC est gérée par Spring pour mettre en relation tous les composants et faciliter l'échange de données
- Les Pojos sont mappés
- Dans le CRUD j'ai réussi à faire un exemple de chacun.

Voici un exemple de diagramme de cas d'utilisation de l'application en vue :



La vue sur le navigateur va changer suivant la personne qui se connecte. Aussi on peut représenter le diagramme de séquences suivant :



Il y'a une présentation en relation avec le rapport pour voir le rendu du projet et aussi un git est disponible pour télécharger et essayer les fonctionnalités.

https://github.com/maliouache/JAVA-EE

CONCLUSION

Le projet a permet d’avoir une bonne base sur le JAVA Entreprise Edition qui est le plus utilisé par les grandes boites d’informatique. Vu que c’est nouveau, le début d’une nouvelle fonctionnalité est souvent lent mais une fois le concept appréhendé, ça a plutôt tendance à partir vite. L’architecture Maven et java ee nous aident à respecter l’architecture mvc au maximum, d’où la bonne organisation du code. Il existe plusieurs librairies qui nous offrent des fonctionnalités très pratique et faciles à implémenter. Si on prend un simple exemple, Hibernate nous donne une façon de mapper nos entities du model de manière simple et l’entity manager s’en charge de faire des requetes basiques pour nous, Ce qui facilite l’exploitation des bases de données.