# Java Enterprise Edition - TP - Intégration des compétences

Jonathan Mabit
*Université d'Angers*
mbtjonathan@gmail.com

Niels Petersen
*Université d'Angers*
ptrsn.niels@gmail.com

## I. Introduction

TP réalisé dans le cadre du cours de Java Enterprise Edition. Le but de ce TP étant de mettre en oeuvre les compétences acquises durant les différentes séances. Ces compétences portent sur les différents moyens afin de créer une application web en JEE tel que JPA, JAX, JSP et les servlets.

## II. Spécialité fonctionnelle

Le choix d'application étant libre, nous avons décidé de créer une application permettant de créer des playlists de musique.

— Une musique possède un id, un nom,un genre. Elle peut être faites par un seul artiste et peut être dans une ou plusieurs playlist. Plusieurs musiques peuvent avoir le même nom. Les musiques peuvent être likée par les utilisateurs.

— Le nom des artistes est unique. Chaque artiste a fait zéro ou plusieurs musiques et possède une description.

— Les playlists ont un nom unique, sont composées de musiques et ont une description.

— Un utilisateur peut liker les musiques, il possède un pseudo et un mot de passe servant à se connecter.

Voici donc les fonctionnalités disponibles aux utilisateurs ; ils peuvent créer des artistes et ajouter des musiques à ces artistes, ils peuvent également créer et gérer des playlists en ajoutant des musiques. Ils est également possible de rechercher des musiques, artistes, et playlists. Et enfin, il est possible de liker des musiques.

Nous avons commencé par créer un github afin de pouvoir travailler plus facilement en collaboration. Nous avons commencé notre projet en réfléchissant aux tables de notre base de données et en l'implémentant. Nous avons utilisé pour cela Xampp, mysql et phpmyadmin. De plus, nos projets s'appuient sur de nombreuses librairies (JPA, JAX-RS, ...), ils nous faut donc ajouter leurs fichiers jar dans les dossiers WEB-INF/lib de nos projets. Par ailleurs, notre application se découpe en deux projets, le premier comportant toute la partie back-end c'est à dire, l'API, la couche dao, et les entitées JPA. Le deuxième, sera un dynamic web project, et contiendra la partie front-end de notre application, c'est à dire les servlet et fichiers jsp.

## III. BDD - SQL

### A. Diagramme de la BDD

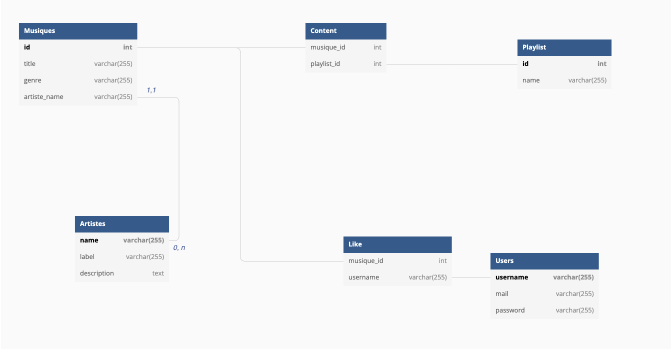


FIGURE 1. Diagramme BDD

Il est important de noter que nous avons limité les likes aux musiques afin de limiter les interractions n-m dans la base de données. Cela nous permet de simplifier la mise en place du projet. En effet avec le temps impartit, il aurait été compliqué d'ajouter la possibilité de liker les playlists et les artistes. De plus, la mécanique serait la même que les likes aux musiques, cela présente donc peu d'intérêt.

## IV. JPA - Java Persistence API

*Cette section et les suivantes concerne le projet SPOTAPI.*

Le JPA permet d'établir un lien entre notre projet java et la base de données. Plus précisément, il s'agit d'un ORM [1], qui permet de faire le lien entre des classes de notre projet et des tables de la bdd. Dans notre projet, il s'agit des classes présentes dans le package entities.

1. Object Relationnal Mapping

### A. *Fichier xml - configuration*

Dans un premier temps, il nous faut créer un fichier persistence.xml dans lequel on va définir des paramètres permettant la communication à la base de données. Voici quelques lignes importantes :

— Cette ligne définit le nom de l'unité utilisée

```
<persistence-unit
↪ name="UniteSpoty">
```

Une unité est un ensemble de classes liés à une table qui seront gérés par les entity manager. De plus une unité contient des paramètres qui lui sont propres.

— Ces lignes définissent les fichiers .class correspondant aux tables de la BDD ainsi que le package où les trouver

```
<class>entities.Musics</class>
<class>entities.Playlist</class>
<class>entities.Artists</class>
<class>entities.Users</class>
```

— Cette ligne définit l'IP, le port pour accéder à la BDD, elle définit également la table à laquelle sont liées les classes ci-dessus.

```
<property
↪ name="javax.persistence.jdbc.url"
↪ value="jdbc:mysql://localhost:
3306/spoty?serverTimezone=UTC" />
```

— Ces lignes donnent l'utilisateur et le mot de passe de celui ci permettant de se connecter à la BDD

```
<property
↪ name="javax.persistence.jdbc.user"
↪ value="root" />
<property
↪ name="javax.persistence.jdbc.
password" value="" />
```

— Il faut également définir quel driver l'unité doit utiliser pour accéder à la base de données, dans notre cas c'est une base de données mysql donc nous avons ceci :

```
<property
↪ name="javax.persistence.jdbc.
driver" value="com.mysql.jdbc.Driver"
↪ />
```

### B. *Entities*

Après avoir créer le fichier persistence.xml, il faut créer les classes permettant d'avoir les enregistrements des tables de la BDD en objet utilisables en java, en veillant à avoir les même noms de classe que ceux inscrit dans le fichier xml.

L'annotation permettant de lier la table artists de la BDD à la classe artists du projet :

```
@Entity @Table(name="artists")
```

Entity indique à JPA, que cette classe doit être lié et gérer par les entity manager. Table indique à quel table lié cette classe. Ces annotations doivent être mises avant le prototype de classe, de toutes les classes définies dans le fichier xml et nécessitant un lien avec la base de donnée. Il suffit de changer le nom de la table liée à la classe.

Il faut ensuite définir des variables pour chaque champ de la base de données, attention, chaque variable doit avoir exactement le même nom que les champs de la table. Chaque variable étant liée à une clé primaire doit être annoté avec l'annotation @Id.

```
@Id
private String name;
private String label;
private String description;
```

Cependant, si la clé primaire est auto incrémentée par la base de donnés, il est nécessaire d'ajouter une annotation supplémentaire, afin d'indiquer à JPA, que lors de la création d'une nouvelle entité, l'id doit être généré à partir du dernier id présent dans la base de données, comme ce fut le cas pour nos classes musics et playlists.

```
@Id
@GeneratedValue(strategy =
↪ GenerationType.IDENTITY)
private int id;
private String title;
private String genre;
```

Ensuite pour toutes les clés étrangères, il faut également préciser de quelle type de relation il s'agit (oneToMany, ManyToOne, ManyToMany), et ensuite nous avons choisi de lier les variables de la manière suivante : une des classes sera liée à la base de donnée, et l'autre classe sera liée à l'autre classe. Voilà ce qu'il se passe :

```
//classe Musics
    @ManyToOne
    @JoinColumn(name="artist")
    private Artists artist;
```

La classe musics va chercher l'enregistrement correspondant à la clé étrangère artist présente dans la table musics. Cela est fait grâce à la propriété name dans l'annotation joinColumn.

```
//classe Artists
    @OneToMany(mappedBy="artist",
    ↪ targetEntity=Musics.class)
    @JsonbTransient
    Set<Musics> music;
```

Ensuite, dans la classe artiste, nous avons besoins des données inverses, c'est à dire de toutes les musiques de l'artiste, pour faire ça, nous pouvons mettre deux propriétés dans l'annotation du type de relation, la propriété targetEntity, qui indique sur quelle classe ira chercher JPA et la propriété mappedBy qui indique à quel variable de cette classe. JPA se chargera ensuite de mettre dans notre variable musics toutes les musiques de l'artiste. Cela est rendu possible par le fait que notre entity manager gère toutes les entités, et peut donc aisément effectuer ce genre de comportement. Cependant, il est très important de noter, que la classe ayant l'annotation @JoinColumn sera celle qui gérera la modification dans la base de données. En effet, si nous reprenons, l'exemple ci-dessus, si nous modifions la variable music en ajoutant une musique au set et que nous faisons un update ensuite pour enregistrer la modification dans la base de données, celles-ci ne sera pas prise en compte par

la bdd, elles le sera seulement pour les entités local gérées par l'entity manager. Si nous voulons ajouter une musique, il nous faut donc en créer une, et lui définir directement son artiste.

Enfin, voici les différents types d'annotation correspondant aux commutations :

— Annotations définissant la commutation 1,n dans la BDD :

```
//classe Artists
@OneToMany(mappedBy="artist",
 ↪  targetEntity=Musics.class)
@JsonbTransient
Set<Musics> music;

{...}

//classe Musics
@ManyToOne
@JoinColumn(name="artist")
private Artists artist;
```

Ces annotations permettent de créer un tableau de musique dans la classe artiste et de lier un objet artiste à chaque musique.

— Annotations définissant la commutation n,n dans la BDD :

```
//classe Playlists
@ManyToMany
@JoinTable(name="content",
 ↪  joinColumns=@JoinColumn(name=
 ↪  "playlist_id"),
 ↪  inverseJoinColumns=
 ↪  @JoinColumn(name="music_id"))
Set<Musics> musics;
```

Tout d'abord, il faut définir, la table qui fait le lien dans les relations n,n (ici content), ensuite il faut définir, sur quelle colonnes s'effectuent les jointures, joinColumn faisant référence à l'entité actuelle, et inverseJoinColumns faisant référence aux entités enregistrées dans la variable.

### C. DAO

Nous avons créer une classe DAO par entités afin que cela soit plus simple d'organiser nos fonctions. De plus, chaque classe implémente une interface, ce qui permet de s'assurer que si nous souhaitons changer de système de persistance des données, nous ayons simplement à implémenter ces interfaces. Enfin, nous avons également créer une classe (DaoMysql), qui gère tous les autres dao, afin de n'avoir qu'un seul import à faire, lorsque nous souhaitons utiliser nos DAO. Pour utiliser JPA, il est nécessaire d'instancier un entity manager, de la manière suivante :

```
private EntityManager em;

public DaoArtists(EntityManagerFactory
 ↪  emf)
{
this.em = emf.createEntityManager();
}
```

L'entity manager factory étant instancié dans notre classe DaoMysql, afin de s'assurer que toutes les couches dao, est le même.

```
public class DaoMysql implements IDao
{
        private IDaoMusics daoMusics;
        private IDaoArtists daoArtists;
        private IDaoUsers daoUsers;
        private IDaoPlaylists
         ↪  daoPlaylists;

        public DaoMysql()
        {
                EntityManagerFactory emf;
                emf = Persistence.
                 ↪  createEntityManagerFactory(
                 ↪  "UniteSpoty");

                daoMusics = new
                 ↪  DaoMusics(emf);

            {...}

        }

        {...}
}
```

L' entity manager nous permet de faire des transactions pour récupérer, modifier, ajouter ou retirer des données de la BDD. Nous avons alors programmé les fonctions qui nous serons utiles pour l'application à l'aide de ses transactions. Prenons l'exemple de la partie DAO pour les musiques. Nous pouvons récupérer des données spécifiques à l'aide d'une query :

```
@Override
public List<Musics> getByGenre(String
 ↪  genre)
{
        Query q = em.createQuery("From
         ↪  Musics WHERE genre=:g",
         ↪  Musics.class);
        q.setParameter("g", genre);

        return q.getResultList();
}
```

Comme on peut le voir ci-dessus, il est également possible de faire des requêtes paramétrées. Point d'attention, le deuxième paramètre de la fonction définie le type attendue comme résultat de la requête, cela permet par la suite à JPA de faire le lien entre BDD et classes. Nous pouvons insérer des données :

```
@Override
public Musics create(Musics music)
{
```

```
        em.getTransaction().begin();
        em.persist(music);
        em.getTransaction().commit();

        return music;
}
```

Nous pouvons modifer un objet existant, comme ci-dessous :

```
@Override
public Musics update(Musics music)
{
        em.getTransaction().begin();
        em.merge(music);
        em.getTransaction().commit();

        return music;
}
```

Nous pouvons supprimer des objets de la bdd :

```
 @Override
public boolean delete(int musicId)
{
        Musics rs = getById(musicId);

        if(rs != null)
        {
                em.getTransaction().begin();
                em.remove(rs);
                em.getTransaction().commit();
        }

        return rs != null;
}
```

On s'aperçoit assez vite, que nous avons accès, finalement, aux mêmes types d'actions que en SQL, ce qui est parfaitement logique. Cependant, il est important de noter, que lorsque nous souhaitons apporter une modification à la base de données quelle qu'elle soit, il faut commencer une transaction avant de faire le changement. Il est également nécessaire de commit la transaction afin que celle-ci soit enregistrée.

Enfin, pour toute la couche dao, nous avons implémenté des tests unitaires. Veillez à bien vider la base de données avant de lancer les tests.

## V. JAX

JAX est est une spécification JEE permettant la création d'un service Web satisfaisant les contraintes d'une architecture REST. En d'autres mots, elles nous permet de créer des services implémentant les standards définis par REST.

### A. *Pré requis*

Comme pour le JPA, la librairie JAXRS nécessite un paramétrage à l'aide d'un fichier xml nommé ici web.xml. Il y a 2 lignes particulièrement importante dans ce fichier, la première :

```
<param-value>spotPack</param-value>
```

Ce paramètre défini le package dans lequel seront implémentés les services. Le deuxième paramètre important :

```
<url-pattern>/spotapi/*</url-pattern>
```

Ce paramètre défini le préfixe ajouté à tous les urls de nos services.

Nous avons fais le choix que nos services renvoie des fichiers JSON. Ainsi il est nécessaire que toutes nos classes définies dans le package entities, implémente l'interface Serializable. Cela signifie que les objets de ces classes pourront être convertis en JSON, afin que le transfert soit plus rapide via le réseau.

### B. *Classes service*

Maintenant, après avoir créé le package spotPack comme définit précédemment, il nous faut créer les classes de service, correspondant à une API. On créé quatre classes services à l'image des entités. Ces classes font le lien entre les fonctions de la couche dao correspondante avec des URL, nous permettant donc d'accéder au bdd, de l'extérieur de ce projet, et donc via des méthodes HTTP (dû au fait que nous utilisions le standard REST).

Pour implémenter un service, une fois la classe créée, définir avant le prototype, le path de ce service, cela se fait grâce à l'annotation path.

```
@Path("/artists")
public class ArtistsService
{
    {...}
}
```

Il faudra également mettre cette annotation devant chaque méthode de notre classe, par ailleurs chaque méthode correspond à un service.

```
@Path("/all")
@GET
@Produces(MediaType.APPLICATION_JSON)
public List<Artists> allArtists()
{
        return
         ↪  dao.getDaoArtists().getAll();
}
```

Il faut ensuite indiquer via une annotation le type de requête autorisé pour ce service. Le type de requête sont ceux implémentés dans HTTP, c'est à dire ; GET pour la récupération d'une ressource, PUT pour la modification, POST pour la création, et DELETE pour la suppression. Comme indiqué précédemment le retour de nos services sera un JSON, il faut indiquer à l'aide de l'annotation le type de la réponse. Il est également possible de passer des paramètres via l'url, par exemple si via un service nous souhaitons récupérer une ressource particulière, il est possible de le faire de la manière suivante :

```
@Path("/findByName/{id}")
@GET
@Produces(MediaType.APPLICATION_JSON)
```

```
public List<Artists>
 ↪  findArtistByName(@PathParam("id")
 ↪  String id)
{
        return dao.getDaoArtists().
         ↪  getByName(id);
}
```

Il faut spécifier dans l'url, que nous attendons un paramètre, cela se fait grâce au nom du paramètre entre crochets. Pour récupérer ce paramètre dans la fonction, on utilise une annotation PathParam comme ci-dessus. Les méthodes POST et PUT requièrent une entrée sous forme de json, pour cela il faut indiquer que le service consomme un JSON avec l'annotation correspondante.

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Artists addArtist(Artists a)
{
        return
         ↪  dao.getDaoArtists().create(a);
}
```

Il y a tout de même quelques points de vigilance, si le paramètre dans l'url est une chaine de caractères, il est possible que cette chaine contienne des caractères spéciaux, pour cela il faudra encoder les paramètres dans le front lors de l'appel à l'api et il faut les décoder dans l'API. Comme ceci :

```
@Path("/findByName/{id}")
@GET
@Produces(MediaType.APPLICATION_JSON)
public List<Artists>
 ↪  findArtistByName(@PathParam("id")
 ↪  String id)
{
        try {
                id = java.net.URLDecoder.
                 ↪  decode(id,
                 ↪  "UTF-8");
        } catch
         ↪  (UnsupportedEncodingException
         ↪  e) {
                e.printStackTrace();
        }
        return dao.getDaoArtists().
         ↪  getByName(id);
}
```

Grâce à notre API, nous proposons différents services aux clients de celle-ci. Cependant, il nous faut réaliser ce client, qui sera le deuxième projet, l'interface utilisateur.

## VI. JSP - SERVLETS

*Cette section concerne le projet SPOTFRONT.*

### A. Servlet & HttpWrapper

Concernant, les servlets nous avons simplement appliqué ce que nous avions vu en TP. Petit rappel toutefois de ce qu'est une servlet et de son mode de fonctionnement : Une servlet est une classe java qui reçoit une requête en entrée, effectue des traitements et renvoie une réponse. Cela permet donc de créer une réponse dynamique. Les httpServlets implémentent deux fonctions, doGet et doPost, qui commeur leur nom l'indique permette de traiter les requêtes http de type GET et POST.

Toutefois, il y a un point en particulier qui mérite un peu d'explication. En cours, pour faire des requêtes, nous avions vu ce code ci-dessous : Cependant, ce code va être utilisé de

```
try {
  URL url = new URL("http://localhost:8080/TP5Ex3/api/etudiant/id/1");
  HttpURLConnection conn = (HttpURLConnection) url.openConnection();
  conn.setRequestMethod("GET");
  conn.setRequestProperty("Accept", "application/json");

  if (conn.getResponseCode() != 200) {
    throw new RuntimeException("Failed : HTTP error code : "
    + conn.getResponseCode());
  }

  BufferedReader br = new BufferedReader(
    new InputStreamReader((conn.getInputStream())));

  String output;
  System.out.println("Output from Server .... \n");
  while ((output = br.readLine()) != null) {
    System.out.println(output);
    ObjectMapper mapper = new ObjectMapper();
    Etudiant e = mapper.readValue(output, Etudiant.class);
    System.out.println(e.toString());
  }

}
catch (MalformedURLException e) {
  e.printStackTrace();
}
catch (IOException e) {
  e.printStackTrace();
}
```

manière récurrente dans notre front-end. Nous l'avons donc généralisé. Voici la classe que nous avons créer pour cela. Ci-

```
public class HttpWrapper
{
        public static <T> T getOneInstance(Class<T> classname, String stringUrl) throws Exception
        {
                T object = null;

                try
                {
                        URL url = new URL(stringUrl);
                        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
                        conn.setRequestMethod("GET");
                        conn.setRequestProperty("Accept", "application/json");

                        if (conn.getResponseCode() != 200 && conn.getResponseCode() != 204) {
                                throw new RuntimeException("Failed : HTTP error code : " + conn.getResponseCode());
                        }

                        BufferedReader br = new BufferedReader(
                                        new InputStreamReader((conn.getInputStream())));

                        String output;
                        while ((output = br.readLine()) != null)
                        {
                                ObjectMapper mapper = new ObjectMapper();
                                object = mapper.readValue(output, classname);
                        }

                }catch (MalformedURLException e)
                {
                        e.printStackTrace();
                }
                catch (IOException e)
                {
                        e.printStackTrace();
                }

                return object;
        }
```

dessus, la fonction getOneInstance, sera utilisé pour tous les appels à l'api retournant une seule instance d'un objet. Le T entre chevrons après le mot clé static défini un type. On peut ensuite utiliser ce type pour nos variables, ce qui nous permet de généraliser. On peut ainsi dire que l'on attend une classe en premier de type T et que notre fonction getOneInstance retourne un objet de type T.

Pour effectuer un post, il faut paramétrer différemment l'objet HttpURLConnection. Pour cela, il faut rajouter quel


type de contenu et l'encodage attendu dans le body.

```
URL url = new URL(stringUrl);
HttpURLConnection conn =
 ↪  (HttpURLConnection)
 ↪  url.openConnection();
conn.setRequestMethod("POST");
conn.setRequestProperty("Content-Type",
 ↪  "application/json; charset=utf-8");
 ↪  //request format type
```

Il faut également paramétrer l'objet de telle sorte qu'il autorise l'écriture dans le body de la requête.

```
conn.setDoOutput(true);
```

Ensuite pour passer notre objet dans le body, il faut d'abord le sérialiser en JSON. Ensuite il faut transformer ce JSON en un tableau d'octets. Puis en utilisant un objet OutputStream et la méthode write, nous pouvons finalement écrire le contenu de la requête.

```
ObjectMapper mapper = new ObjectMapper();
String jsonInput = mapper.
 ↪  writeValueAsString(inputObject);

try(OutputStream os =
 ↪  conn.getOutputStream())
{
        byte[] input =
         ↪  jsonInput.getBytes("utf-8");
    os.write(input, 0, input.length);
}
```

Ensuite comme nous l'avions expliqué, lorsqu'un des paramètres d'url est une chaine de caractères, nous devons l'encoder comme suit :

```
artistId =
 ↪  java.net.URLEncoder.encode(artistId,
 ↪  "UTF-8");
HttpWrapper.deleteOneInstance(boolean.
 ↪  class,
 ↪
 ↪  "http://localhost/SPOTAPI/spotapi/artists/
delete/" + artistId);
```

Enfin, tous les urls dans notre projet sont appelés sur le port 80, car nous avons changé le port par défaut du serveur tomcat à 80. Pour tester les projets, pensez à bien définir votre port, sinon tous les appels à l'API ne fonctionneront pas.

### *B. WebFilter*

Les filters sont, tout comme les servlets, une classe java. Les filters s'appliquent sur des servlets particulières ou sur certains chemins d'url. Ainsi, au lieu que la requête passe directement à la servlet, elle peut passer par un ou plusieurs filtres. Ils ont de nombreux cas d'utilisations, ils peuvent bien entendu permettre d'authentifier un utilisateur pour rendre certaines ressources disponibles seulement pour certains utilisateurs, groupes ou autres. Mais ils permettent également d'établir la connexion à la base de donnée et de la fermer, la préparation de transaction à la bdd ou encore gérer l'encodage de la page affichée. Dans notre cas, nous avons utilisé les filtres, dans leur utilisation la plus basique, pour l'authentification.

```
@WebFilter(urlPatterns = "/session/*")
public class AuthenticationFilter extends
 ↪  HttpFilter implements Filter
{


    {...}

        /**
         * @see
 ↪  Filter#doFilter(ServletRequest,
 ↪  ServletResponse, FilterChain)
         */
        public void
         ↪  doFilter(HttpServletRequest
         ↪  request, HttpServletResponse
         ↪  response, FilterChain chain)
         ↪  throws IOException,
         ↪  ServletException
        {
                HttpSession session =
                 ↪  request.getSession();
```

```
                if(session.getAttribute(
                ↪  "user") !=
                ↪  null)
                {
                        chain.doFilter(
                        ↪  request,
                        ↪  
                        ↪  response);
                }else
                {
                        response.
                        ↪  sendRedirect(
                        ↪  "../login");
                }
        }

    {...}

}
```

Voici comment cela fonctionne, au moment de l'inscription ou de la connexion d'un utilisateur, on enregistre en variable de session, l'objet Users correspondant. Ici notre filtre s'applique pour toutes les ressources ayant un chemin d'accès url commençant par /session. Nous aurions pu spécifier d'appliquer le filtre à certaines servlets particulières comme ceci :

```
@WebFilter(servletNames={servlet1,
↪  servlet2})
public class AuthentiticationFilter
↪  extends HttpFilter implements Filter
{
    {...}
}
```

Pour revenir à notre servlet pour l'authentification, toutes les requêtes voulant atteindre une ressource ayant l'url commençant par "/session" passera par le filtre, et si la variable de session user est null sera redirigé pour se connecter, sinon il pourra accéder à la ressource. Dans le cas ou plusieurs filtres aurait été présent pour ce chemin, c'est là que devient très importante la chaine de filtres, et l'instruction doFilter, qui permet à la manière d'une chaine d'exécuter le filtre suivant et ainsi de suite.

### *C. JSP/JSTL*

Dans notre projet, nous avons principalement utilisé du JSTL. Voici les "fonctions" que nous avons utilisés : Dans un premier temps, il est possible de créer des variables au sein même d'une page jsp, avec une portée différente.

```
<c:set var="musics"
↪  value="${requestScope.musics}"
↪  scope="request" />
```

Le scope request indique ici, que la portée de la variable est la requête en cours seulement. Il y a ensuite, les boucles forEach :

```
<c:forEach var="m" items="${musics}">
<div class="small Card"
↪  style="flex-direction: column">
        <h4>${m.getTitle()}</h4>
        <form method="POST"
        ↪  action="artist">
        <input type="hidden" name="a"
        ↪  value="${requestScope.artist.getName()
        ↪  />
        <input type="hidden" name="music"
        ↪  value="${m.getId()}"/>
        <input type="submit"
        ↪  name="deleteMusic"
        ↪  value="Supprimer" />
        </form>
</div>
</c:forEach>
```

Tout le code contenu entre ces deux balises, sera répété par la boucle forEach, celle-ci effectue des iterations sur la variable renseignée dans items, et enregistre la valeur actuelle de l'itération dans la variable spécifiée par la propriété var. On peut ensuite appelé les méthode de l'objet, ici m est une musique, on peut donc appeler toutes les méthodes de l'objet Musics. Ce qui nous permet ainsi de récupérer les informations liées à l'objet. Pour afficher des données, il suffit de les mettre en crochet précédé du symbole $. Autre point important, il est possible de n'afficher que certaines parties du code en fonction d'une condition.

```
<c:if
↪  test="${erreurs.containsKey('username')}">
↪  ${erreurs.get('username')}</c:if>
```

La condition étant contenue dans la propriété test et le message affiché si celle-ci est validée se trouve entre les balises <c :if></c :if>. Finalement, nous avons fait un menu dans notre application, et pour éviter d'avoir à le copier dans tous les fichiers, ce qui manquerait de flexibilité, nous avons utilisé les include présent dans le jsp. L'utilisation étant simple, il suffit de spécifier le nom du fichier jsp que l'on souhaite inclure. Il faut bien entendu, porter son attention sur l'endroit où on met le include, car le code du fichier sera inséré à cette position.

```
<jsp:include page="nav.jsp" />
```

Dernier point, les variables définies dans le fichier qui contient l'instruction include, sont accessibles à l'intérieur du fichier inclus.

## VII. Résultat Final

Voici quelques captures d'écran du résultat final :

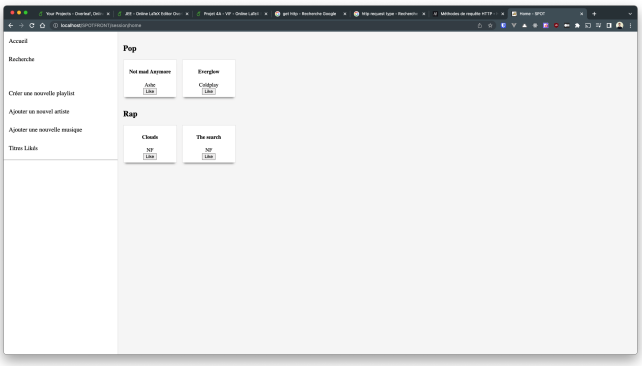


FIGURE 2. La page d'accueil

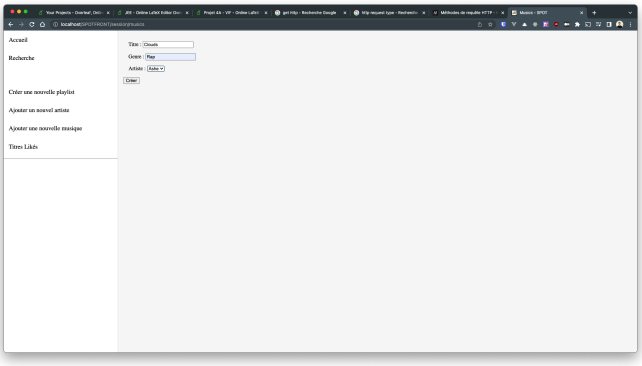


FIGURE 3. La page pour ajouter une musique

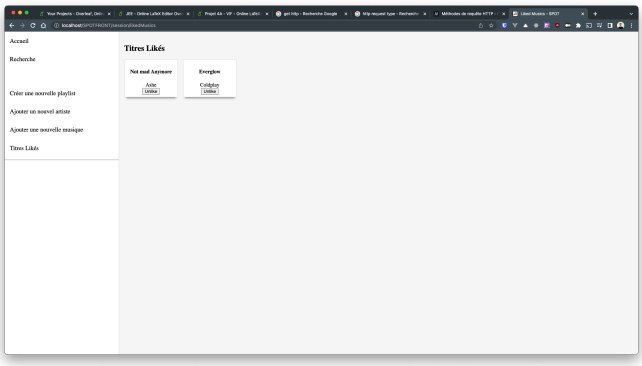


FIGURE 4. La page des musiques likés

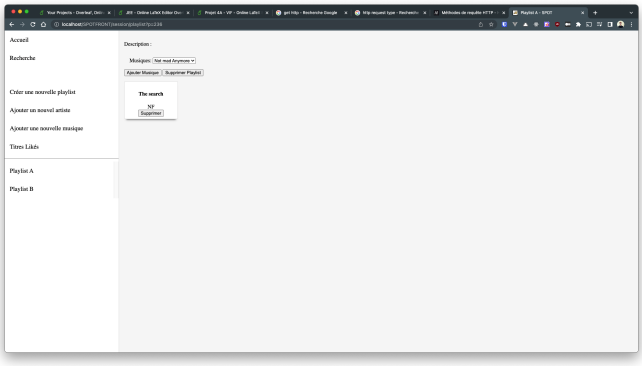


FIGURE 5. La page d'une playlist

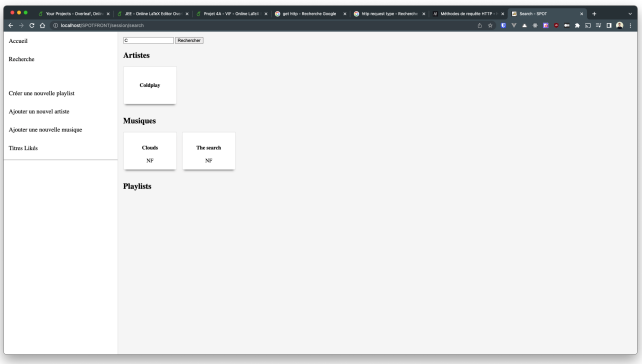


FIGURE 6. La recherche de musiques, artistes, playlists

## VIII. DISCUSSION

### A. Plus de rapidité

On observe que lors du premier appel à l'API, et donc par conséquent à la base de données, le temps de chargement est très long, pour établir la connexion, vérifier les identifiants et effectuer la requête. Ainsi comme nous l'avons vu, il pourrait être intéressant d'établir la connexion directement dans un filter, afin de gagner du temps lors du premier appel. Ce qui améliorerait l'expérience utilisateur.

### B. Lecteur Audio

Pour le moment, notre application s'utilise comme une bibliothèque de titres, artistes et de playlist, il serait intéressant d'ajouter un lecteur audio, pour pouvoir lire les fichiers audios des musiques que l'on ajoute.

### C. Archive War

Afin de faciliter le déploiement du projet, nous pourrions créer une archive WAR [2], qui est l'équivalent d'une archive jar, mais pour les projets web. Il suffirait ainsi de déployer cette archive War sur un serveur d'application.

## IX. CONCLUSION

Ce TP nous a permis de réfléchir à comment travailler de manière efficace sur un projet, lorsque nous sommes plusieurs développeurs et lorsque le projet comporte différentes parties (DAO, front-end, API). Nous avons pu apprendre ainsi le contexte et limite des technologies que nous avons utilisés. Nous avons également pu intégrer les différentes connaissances vu au cours des TP de logiciel.

2. Web Application Archive

## ANNEXE A
## PROJET GITHUB

Répertoire Github du projet : https://github.com/JonathanMbt/JEE-TP6

## ANNEXE B
## IMAGES ET BASE DE DONNÉES

Toutes les images utilisées dans ce rapport sont disponibles dans le dossier annexes/figures sur Github.