



EJB PROJET CINEMA

Naveau Simon, Travaillé Loïc

Structure / articulation du projet :

Tout d'abord nous avons téléchargé et configuré le serveur GlassFish 5.0.

Nom	Modifié le	Type	Taille
asadmin	08/09/2017 07:27	Fichier	3 Ko
asadmin.bat	08/09/2017 07:27	Fichier de comma...	3 Ko

Il faut commencer par démarrer le service de base de données avec le script **asadmin** de GlassFish. Dans une invite de commande il suffit d'utiliser la commande « *asadmin start-database* » et le service est lancé.

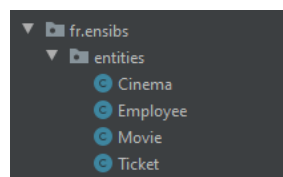
```
<properties>
  <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.EmbeddedDriver"/>
  <property name="javax.persistence.jdbc.url" value="jdbc:derby:cinema_db;create=true"/>
  <property name="eclipselink.ddl-generation" value="create-tables" />
  <property name="eclipselink.ddl-generation.output-mode" value="database" />
</properties>
```

Dans le **persistence.xml** nous ajoutons les propriétés liant la base à l'application. Cela va nous permettre de stocker nos informations. Dans la 3^{ème} propriété si on met la valeur « *drop-and-create-tables* » la base sera vidée à chaque redémarrage de l'application ce qui dans notre cas n'est pas souhaitable.

```
<persistence-unit name="cinema" transaction-type="JTA">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>

  <class>fr.ensibs.entities.Cinema</class>
  <class>fr.ensibs.entities.Employee</class>
  <class>fr.ensibs.entities.Movie</class>
  <class>fr.ensibs.entities.Ticket</class>
```

Dans ce même fichier nous déclarons tous les types d'entités qui pourront être stockés.



Nous avons donc créé une classe avec le tag **@Entity** pour chaque entité à stocker en base.

```
@Entity
public class Cinema implements Serializable {

    @Id
    @GeneratedValue
    private Long idCinema;

    private String name;

    private String address;

    private int postalCode;
```

Dans ces classes nous déclarons les colonnes de la table (attributs de classe) avec leurs getters et setters. Et nous ajoutons les restrictions de jointure pour lier nos tables avec les **@OneToMany**, **@ManyToOne** et **@ManyToMany**.

```

@OneToMany(
    mappedBy = "cinema",
    cascade = CascadeType.ALL,
    orphanRemoval = true,
    fetch = FetchType.EAGER
)
private List<Employee> employees = new ArrayList<>();

```

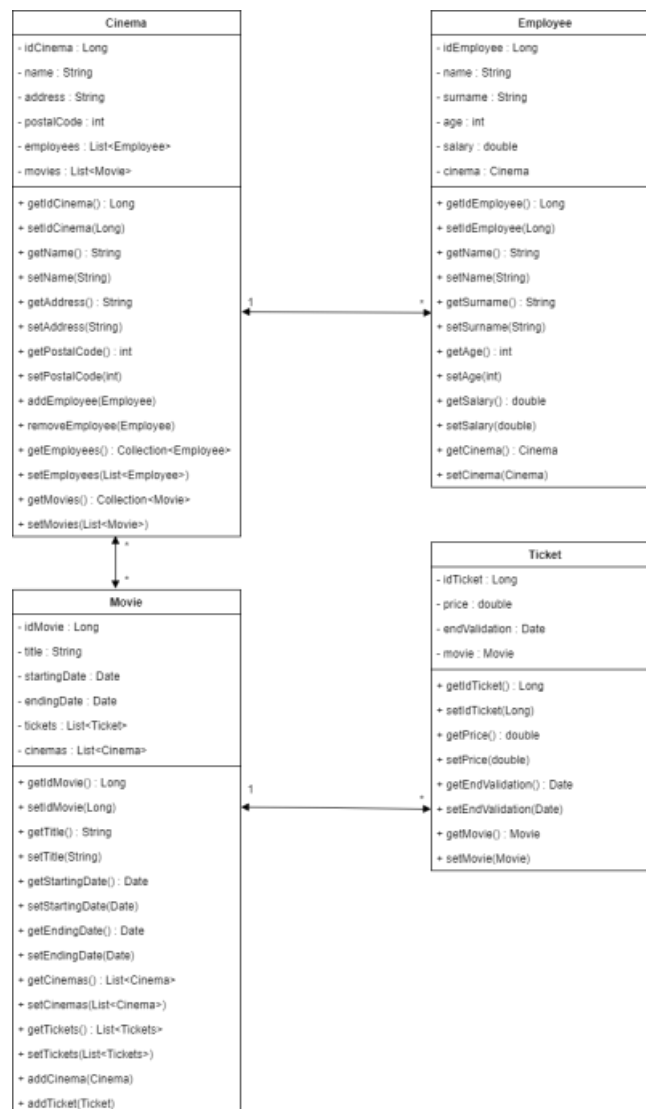
```

@ManyToOne(fetch = FetchType.EAGER)
private Cinema cinema;

```

Par exemple ci-dessus la configuration de la jointure entre le cinéma et ses employés. Nous avons ajouté certaines options pour activer la suppression automatique des fils lors de la suppression de l'objet parent.

Avec ces 4 classes nous avons donc créé la structure suivante :



Pour chaque bean **Entity** nous avons donc un bean **Session** qui va nous permettre de manager la base de données. Ces sessions implémentent à chaque fois une interface de type **@Remote** et une interface de type **@Local**. Nous y avons implémenté les types de requête dont nous avons besoin (création, recherche, suppression, ...). Il ne faut pas oublier d'ajouter le tag **@PersistenceContext** à l'EntityManager pour le lier par le nom avec la persistance de donnée dans le **persistence.xml**.

```

maService.java
package fr.ensibs.sessions;

import ...

@Stateless
public class CinemaService implements CinemaServiceLocal, CinemaServiceRemote {

    @PersistenceContext(unitName = "cinema")
    private EntityManager em;

    @Override
    public void createCinema(String name, String address, int postalCode) {
        Cinema cinema = new Cinema();
        cinema.setName(name);
        cinema.setAddress(address);
        cinema.setPostalCode(postalCode);
        em.persist(cinema);
    }
}

```

Dans certains cas de modifications des clés étrangères, il ne faut pas oublier de mettre à jour les 2 items en jeu, sinon les données sont en cache et donc pas effectives.

```

em.getEntityManagerFactory().getCache().evict(Employee.class, employee.getIdEmployee());

```

Gestion des cinémas :

Pour commencer nous avons des entités de type **Cinema** qui possèdent un id, un nom, une adresse, un code postal, une liste d'employés et une liste de films à l'affiche.

🏠 Cinemas Management

Add a cinema :

CREATE ➤

Cinema list :

Name	Address	Postal code	Employees	Movies	Actions
Simon Naveau	7 Allée de l'École Maître d'aban	56000	0	1	⋮ 🗑
j.h,hj	hj,hj,	9848	0	1	⋮ 🗑
%MM	mÁ¹	951	0	1	⋮ 🗑
Simon Naveau	651651561	56000	0	0	⋮ 🗑

Après la page d'accueil de l'application nous avons une page de gestion de cinéma, il est sur celle-ci possible d'ajouter des cinémas, les supprimer ou aller vers leur page de management. Cette page est une JSP associée, elle, à une servlet Java. Je vais maintenant décrire le fonctionnement global de cette page qui est le même pour les suivantes, que je ne vais donc pas détailler.

```

@WebServlet(name = "manageCinemas", urlPatterns = {"/manageCinemas"})
public class ManageCinemas extends HttpServlet {

    public static final String VIEW = "/manageCinemas.jsp";

    @EJB
    private CinemaServiceLocal cinemaService;

    /**
     * Handles the HTTP <code>POST</code> method.
     *
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException      if an I/O error occurs
     */
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        List<Cinema> cinemas = this.cinemaService.getAllCinemas();
        request.setAttribute("cinemas", cinemas);
        request.getRequestDispatcher(VIEW).forward(request, response);
    }
}

```

Nous avons donc une servlet nommée **ManageCinemas** et accessible par l'URL renseignée dans `urlPatterns`, ici `/manageCinemas`. Cela veut dire qu'une tentative d'accès par navigateur à l'adresse <http://localhost:8080/CinemaProject/manageCinemas> va envoyer une requête GET vers la servlet et donc appeler la méthode `doGet` ci-dessus. Dans celle-ci nous récupérons la liste de tous les cinémas avec l'aide du bean session associé. Nous le plaçons ensuite, cette liste de cinéma, dans l'attribut `cinemas` qui va être « forward » à la page de l'attribut `VIEW` et donc ici `/manageCinemas.jsp`.

Ces JSP utilisent le framework CSS Materialize et comportent plusieurs formulaires dont nous allons parler dans quelques lignes. Mais tout d'abord revenons à la liste de cinémas qui est dans notre requête. Cet attribut est accessible dans notre JSP et nous allons itérer dessus pour afficher tous les cinémas de la base.

```

<c:forEach items="${cinemas}" var="cinema">
    <tr>
        <td>${cinema.name}</td>
        <td>${cinema.address}</td>
        <td>${cinema.postalCode}</td>
        <td>${cinema.employees.size()}</td>
        <td>${cinema.movies.size()}</td>
        <td>
            <form action="manageCinemas" method="post">
                <input type="hidden" name="idCinemaToManage" value="${cinema.idCinema}"/>
                <button class="btn-floating btn-large waves-effect waves-light blue darken-4"><i
                    class="material-icons">more</i></button>
            </form>
        </td>
        <td>
            <form action="manageCinemas" method="post">
                <input type="hidden" name="idCinemaToDelete" value="${cinema.idCinema}"/>
                <button class="btn-floating btn-large waves-effect waves-light red darken-4"><i
                    class="material-icons">delete</i></button>
            </form>
        </td>
    </tr>
</c:forEach>

```

Avec cette boucle `forEach` on itère sur l'attribut dans « items » de la première ligne. Après avec `${...}` on peut ajouter du code et accéder aux attributs des objets.

On peut constater aussi la présence de formulaires. Pour pouvoir les lier à la servlet on ajoute dans leur attribut action le nom de la servlet qui va faire le traitement et post dans l'attribut method. Ainsi lors de la validation du formulaire nous pouvons rebasculer sur la servlet Java pour effectuer des traitements sur les données du formulaire.

```
*/
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws IOException {
    if (request.getParameter( "$: "cinemaName") != null) {
        try {
            String name = request.getParameter( "$: "cinemaName");
            String address = request.getParameter( "$: "cinemaAddress");
            String postalCode = request.getParameter( "$: "cinemaPostalCode");
            this.cinemaService.createCinema(name, address, Integer.parseInt(postalCode));
        } catch (Exception e) {
            e.printStackTrace();
        }
        response.sendRedirect( "$: "/CinemaProject/manageCinemas");
    }

    } else if (request.getParameter( "$: "idCinemaToManage") != null) {
        response.sendRedirect( "$: "/CinemaProject/manageCinema?id=" + request.getParameter( "$: "idCinemaToManage"));
    }

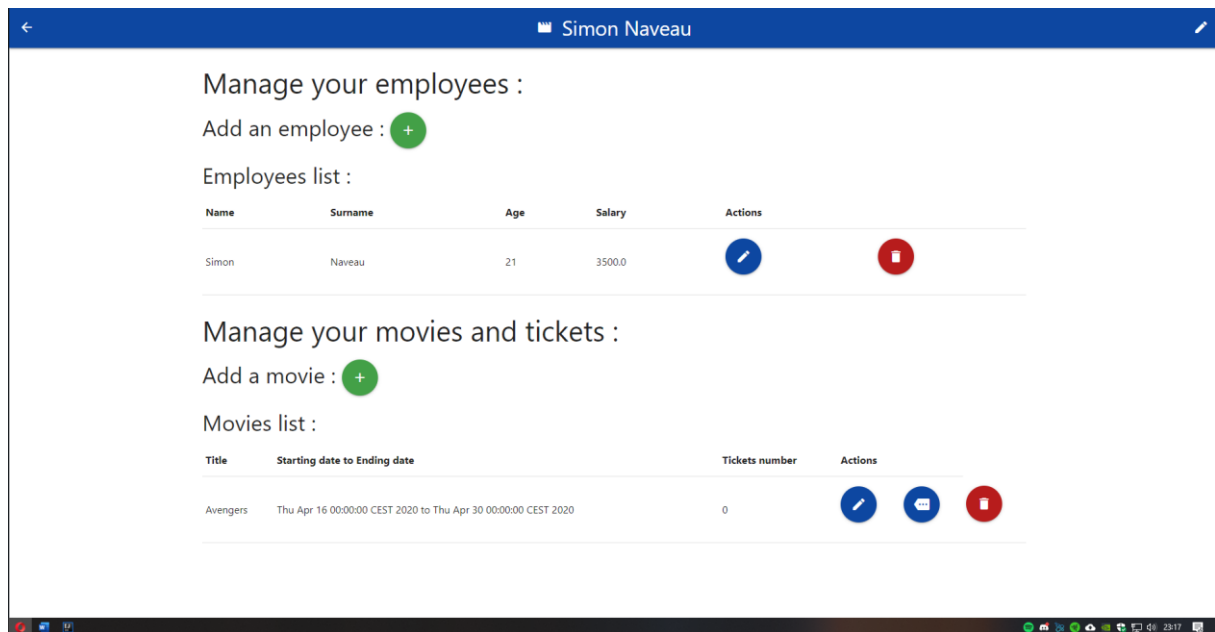
    } else if (request.getParameter( "$: "idCinemaToDelete") != null) {
        try {
            this.cinemaService.removeCinema(Long.parseLong(request.getParameter( "$: "idCinemaToDelete")));
        } catch (Exception e) {
            e.printStackTrace();
        }
        response.sendRedirect( "$: "/CinemaProject/manageCinemas");
    }
}
```

Tout ceci a lieu dans la méthode doPost de la servlet. Avec des conditions nous regardons quels paramètres sont dans la requête qui arrive (associé aux noms des inputs des forms). Suivant la présence de tel ou tel paramètre nous savons alors quelle action est demandée par l'utilisateur et nous effectuons alors le traitement adéquat. Prenons pour exemple la première condition :

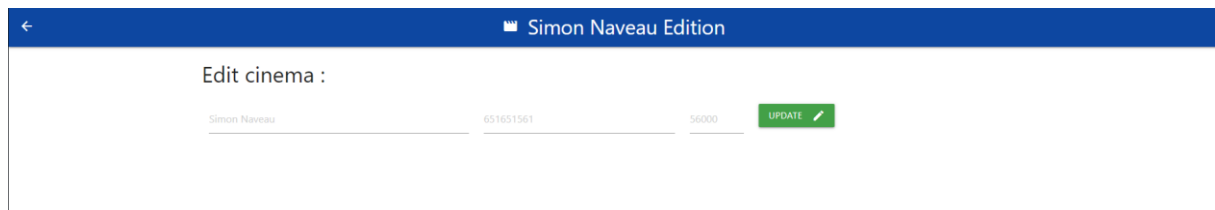
```
if (request.getParameter( "$: "cinemaName") != null) {
    try {
        String name = request.getParameter( "$: "cinemaName");
        String address = request.getParameter( "$: "cinemaAddress");
        String postalCode = request.getParameter( "$: "cinemaPostalCode");
        this.cinemaService.createCinema(name, address, Integer.parseInt(postalCode));
    } catch (Exception e) {
        e.printStackTrace();
    }
    response.sendRedirect( "$: "/CinemaProject/manageCinemas");
}
```

Lors de l'appui sur le bouton CREATE de la page il y a 3 inputs d'envoyés dans la requête. Un se nomme « cinemaName », il est présent nous savons donc que c'est bien un ajout de cinéma qui est demandé. Après nous récupérons les 3 paramètres « name, address et postalCode » et nous appelons la méthode « createCinema » du bean session cinemaService qui gère les cinémas en base. Une fois cela fait, nous arrivons à la dernière ligne qui est une redirection vers cette même servlet (pour effectuer un rafraîchissement de la page). Il est aussi possible de passer des données dans l'URL et de les récupérer dans les méthodes doGet. Par exemple on passe l'id du cinéma à manager dans l'URL vers la page de management de cinéma. Comme ça à l'arrivée, la servlet manageCinema peut récupérer l'id et demander les informations de celui-ci à la base.

```
else if (request.getParameter( "$: "idCinemaToManage") != null) {
    response.sendRedirect( "$: "/CinemaProject/manageCinema?id=" + request.getParameter( "$: "idCinemaToManage"));
}
```



Sur la page de management d'un cinéma on a la liste de ses employés et des films qu'il propose. Il est possible de supprimer des éléments dans les listes avec les boutons rouges. On peut en ajouter avec les boutons verts (nous allons le voir dans les parties suivantes), et modifier les informations du cinéma avec le bouton d'édition présent dans le coin inférieur droit de la page. On arrive alors sur cette page :



Il n'est pas obligatoire de renseigner tous les champs pour modifier. Les champs non renseignés ne seront pas changés.

Gestion des employés :

Tout d'abord un employé ne peut travailler que dans un unique cinéma. C'est une relation OneToMany.

Manage your employees :

Add an employee : 

Employees list :

Name	Surname	Age	Salary	Actions
Simon	Naveau	21	3500.0	 

Dans cette section on peut voir la liste des employés, en supprimer ou en éditer (avec le bouton bleu).

Edit the employee :

Simon Naveau 21 3500.0 UPDATE

Comme pour le reste seul les champs renseignés seront modifiés.

On peut aussi comme dit précédemment en ajouter. Pour cela il faut juste compléter les champs de la page d'ajout accessible par le bouton vert.

Add an employee :

Name Surname Age Salary CREATE




Gestion des films :

Un cinéma à plusieurs films et l'inverse aussi. C'est une relation ManyToMany.

Manage your movies and tickets :

Add a movie : +

Movies list :

Title	Starting date to Ending date	Tickets number	Actions
Avengers	Thu Apr 16 00:00:00 CEST 2020 to Thu Apr 30 00:00:00 CEST 2020	0	  

Comme précédemment on peut voir, modifier et supprimer des éléments de la liste. Il est aussi possible d'en ajouter avec la page prévue à cet effet.

Add a movie :

Name jj/mm/aaaa jj/mm/aaaa CREATE

Search for an existing movie :

Name SEARCH

Avengers Thu Apr 16 00:00:00 CEST 2020 Thu Apr 30 00:00:00 CEST 2020 0 4
ADD +

Mais aussi de chercher dans ceux déjà existents avec le champ de recherche. Dans l'exemple ci-dessus nous avons recherché le film *Avengers* qui existe déjà dans un autre cinéma. On peut alors l'ajouter dans le nôtre aussi.

Gestion des tickets :



Il y a aussi ce bouton qui est en plus sur chaque élément de la liste de film. Celui-ci permet de gérer les tickets qui lui sont associés.

Avengers Management

Add tickets :

Price

jj/mm/aaaa

Number of tickets

CREATE >

Ticket list :

Ticket number	Price	End validation date	Actions
452	7.0	Wed Apr 15 00:00:00 CEST 2020	
453	7.0	Wed Apr 15 00:00:00 CEST 2020	
454	7.0	Wed Apr 15 00:00:00 CEST 2020	

On peut sur cette page ajouter des tickets dans la liste de tickets en bas de page. Il est toujours possible d'en supprimer.