Nom & prénom : Fahd KORAICHE
Classe : 4IIR G2
Site : CENTRE

Rapport Java JEE

Objet* : *Projet cinéma avec Spring JEE et AngularJS

❖ Code source :

https://github.com/Koraiche/EMSI-4IIRG2CC-S8-JEE-/tree/main/Projet%20cinema

❖ Présentation :

- 1 - Technologies :

1.1 - Spring JEE :

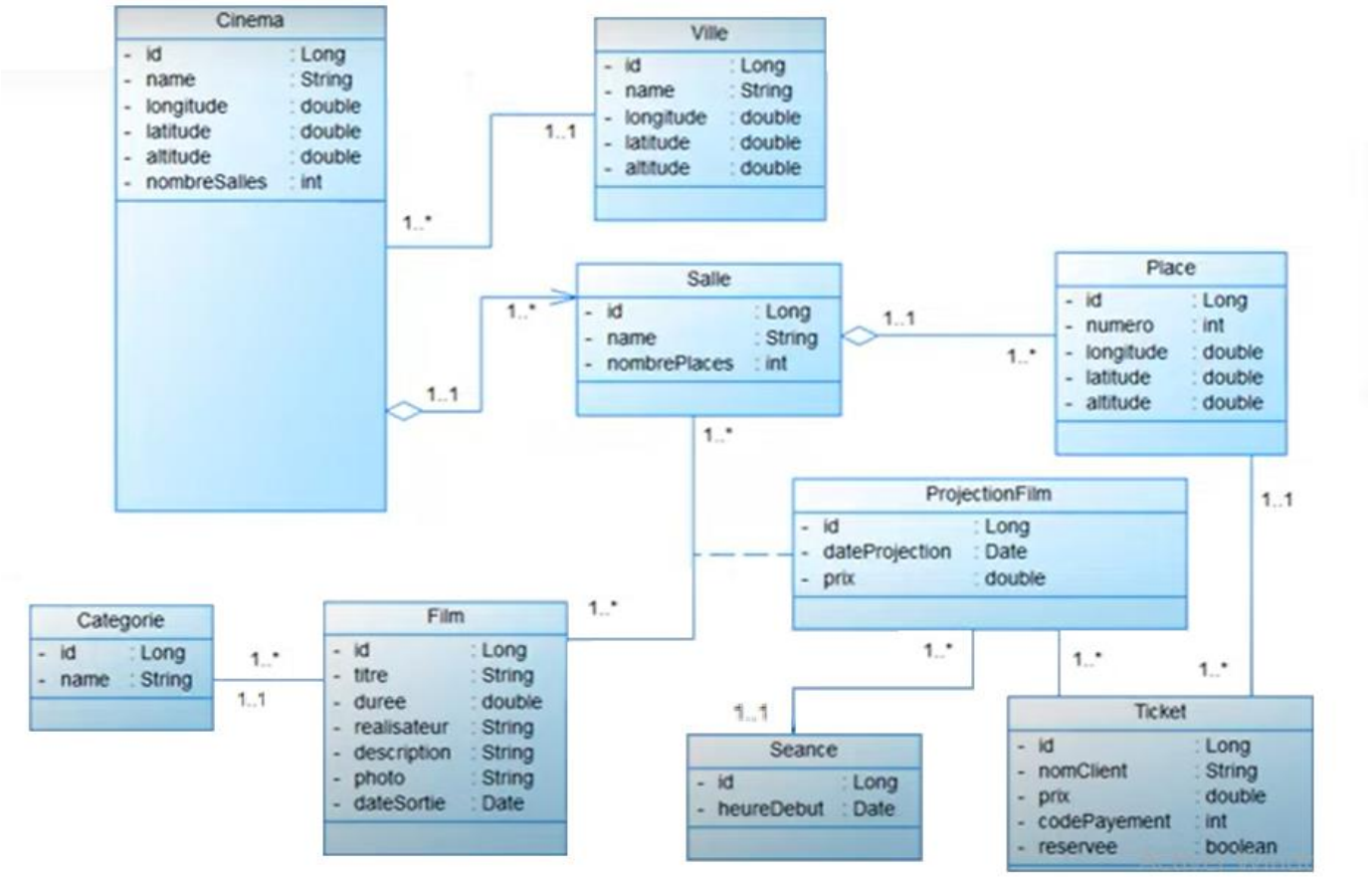
C'est le cadre de développement d'applications pour JavaEE. Il s'agit d'une plate-forme Java open source qui prend en charge Java pour le développement d'une application Java robuste de manière très fluide et facile.

1.2 – AngularJS :

Angular est un framework d'application Web open source basé sur TypeScript dirigé par l'équipe **Angular** de Google et par une communauté d'individus et d'entreprises.

Nom & prénom : Fahd KORAICHE
Classe : 4IIR G2
Site : CENTRE

- 2 - Conception :



- 3 - Représentation BACKEND :

3.1 – Partie Couche DAO :

Dans cette première partie on a déclaré deux package pour representer notre couche DAO ma.emsi.cinema.dao et ma.emsi.cinema.entities, dont lesquels on a implémenté toutes nos classes ainsi que les repositories qui les conviennes.

Les* repos ma.emsi.cinema.dao *:

```
@RepositoryRestResource @CrossOrigin("*")
public interface CategoryRepository extends JpaRepository<Categorie, Long> {

}
```

```
@RepositoryRestResource @CrossOrigin("*")
public interface CinemaRepository extends JpaRepository<Cinema, Long> {

}
```

Nom & prénom : Fahd KORAICHE
Classe : 4IIR G2
Site : CENTRE

```java
@RepositoryRestResource @CrossOrigin("*")
public interface FilmRepository extends JpaRepository<Film, Long> {

}
```

```java
@RepositoryRestResource @CrossOrigin("*")
public interface PlaceRepository extends JpaRepository<Place, Long> {

}
```

```java
@RepositoryRestResource @CrossOrigin("*")
public interface SalleRepository extends JpaRepository<Salle, Long> {

}
```

```java
@RepositoryRestResource @CrossOrigin("*")
public interface ProjectionRepository extends JpaRepository<Projection, Long> {

}
```

```java
@RepositoryRestResource @CrossOrigin("*")
public interface SeanceRepository extends JpaRepository<Seance, Long> {

}
```

```java
@RepositoryRestResource @CrossOrigin("*")
public interface TicketRepository extends JpaRepository<Ticket, Long> {

}
```

```java
@RepositoryRestResource @CrossOrigin("*")
public interface VilleRepository extends JpaRepository<Ville, Long> {

}
```

Les classes ma.emsi.cinema.entities :

```java
@Entity @Data @NoArgsConstructor @ToString @AllArgsConstructor
public class Categorie {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(length=75)
    private String name;
    @OneToMany(mappedBy = "categorie")
    @JsonProperty(access = Access.WRITE_ONLY)
    private Collection<Film> films;

}
```

Nom & prénom : Fahd KORAICHE
Classe : 4IIR G2

Site : CENTRE

```java
@Entity @Data @NoArgsConstructor @ToString @AllArgsConstructor
public class Cinema implements Serializable{
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private double longitude, latitude, altitude;
    private int nombreSalles;
    @OneToMany(mappedBy = "cinema")
    private Collection<Salle> salles;
    @ManyToOne
    private Ville ville;
}
```

```java
@Entity @Data @NoArgsConstructor @ToString @AllArgsConstructor
public class Film {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String titre;
    private String description;
    private String realisateur;
    private Date dateSortie;
    private double duree;
    private String photo;
    @OneToMany(mappedBy = "film")
    @JsonProperty(access = Access.WRITE_ONLY)
    private Collection<Projection> projections;
    @ManyToOne
    private Categorie categorie;
}
```

```java
@Entity @Data @NoArgsConstructor @ToString @AllArgsConstructor
public class Place {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private int numero;
    private double longitude, latitude, altitude;
    @ManyToOne
    private Salle salle;
    @OneToMany(mappedBy = "place")
    @JsonProperty(access = Access.WRITE_ONLY)
    private Collection<Ticket> tickets;
}
```

```java
@Entity @Data @NoArgsConstructor @ToString @AllArgsConstructor
public class Projection {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Date dateProjection;
    private double prix;
    @ManyToOne
    @JsonProperty(access = Access.WRITE_ONLY)
    private Salle salle;
    @ManyToOne
    private Film film;
    @OneToMany(mappedBy = "projection")
    @JsonProperty(access = Access.WRITE_ONLY)
    private Collection<Ticket> tickets;
    @ManyToOne
    private Seance seance;
```

Nom & prénom : Fahd KORAICHE
Classe : 4IIR G2
Site : CENTRE

```java
@Entity @Data @NoArgsConstructor @ToString @AllArgsConstructor
public class Salle {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private int nombrePlace;
    @ManyToOne
    @JsonProperty(access = Access.WRITE_ONLY)
    private Cinema cinema;
    @OneToMany(mappedBy = "salle")
    @JsonProperty(access = Access.WRITE_ONLY)
    private Collection<Place> places;
    @OneToMany(mappedBy = "salle")
    @JsonProperty(access = Access.WRITE_ONLY)
    private Collection<Projection> projections;
}
```

```java
@Entity @Data @NoArgsConstructor @ToString @AllArgsConstructor
public class Seance {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Temporal(TemporalType.TIME)
    private Date heureDebut;
}
```

```java
@Entity @Data @NoArgsConstructor @ToString @AllArgsConstructor
public class Ticket {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nomClient;
    private double prix;
    @Column(unique=false, nullable=true)
    private Integer codePayement;
    private boolean reserve;
    @ManyToOne
    private Place place;
    @ManyToOne
    @JsonProperty(access = Access.WRITE_ONLY)
    private Projection projection;

}
```

```java
@Entity @Data @NoArgsConstructor @ToString @AllArgsConstructor
public class Ville {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private double longitude, latitude, altitude;
    @OneToMany(mappedBy = "ville")
    private Collection<Cinema> cinemas;
}
```

3.2 – Partie Couche DAO avec services :

Dans le cadre d'utilisation de notre application cinéma on aura besoin d'implémenter noter base de données pour pouvoir faire des tests ainsi que de visualiser la communication entre l'api Rest créer grâce Spring avec notre

Nom & prénom : Fahd KORAICHE
Classe : 4IIR G2
Site : CENTRE

application Frontend créée avec Angular. Pour faire cela, on va implémenter une classe service qui va permettre d'initialiser nos tables grâce à Spring JPA au moment du lancement du projet, pour cela on va garder notre base en mode Create pour ne pas réinsérer les mêmes lignes à chaque démarrage de test.

Lien avec la base (MySQL) :

```
spring.datasource.url = jdbc:mysql://localhost:3306/db_cinema?serverTimezone=UTC
spring.datasource.username = root
spring.datasource.password =
spring.jpa.show-sql = true
spring.jpa.hibernate.ddl-auto = update
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
server.port = 8082
```

Interface d'initialisation :

```
public interface ICinemaInitService {
    public void initVilles();
    public void initCinema();
    public void initSalles();
    public void initPlaces();
    public void initSeances();
    public void initFilms();
    public void initCategories();
    public void initProjections();
    public void initTickets();

}
```

Class d'initialisation : Cette classe possède comme attributs tous les repositories qu'on a créé bien sûr avec le principe d'injection des dépendances @Autowired, et aussi elle redéfinie toutes les méthodes de l'interface

```
@Override
public void initCinema() {
    villeRepository.findAll().forEach(v->{
        Stream.of("MegaRama","IMAX", "FOUNOUN", "CHAHRAZAD", "DAOULIZ")
        .forEach(nameCinema -> {
            Cinema cinema = new Cinema();
            cinema.setName(nameCinema);
            cinema.setNombreSalles(3+(int)(Math.random()*7));
            cinema.setVille(v);cinemaRepository.save(cinema);
        });
    });

}
```

ICinemaService puisque elle l'implémente !, chaque fonction redéfinie contient un code qui utilise le repository adéquat a la table visée puis en insérer quelques lignes qui vont s'avérer utile dans la partie test(voici un exemple d'initialisation de la table Cinéma). Dans cet exemple on a parcouru la liste des villes, et pour chaque ville, on a figé un Stream de String que pour chacun d'eux on va créer un objet cinema qui va recevoir le Stream autant que name puis la ville courante et il l'insère dans la base via le repository adéquat. De plus, cette classe contient l'annotation **@Service** qui marque les Beans, ainsi que l'annotation **@Transactional** qui permet de garantir le principe de transaction dans les bases de données pour résoudre le problème de persistance de données.

3.2 – Partie Web, Controllers and RestControllers:

Dans cette partie on a représenté le package ma.emsi.cinema.web contenant la classe CinemaRestController qui représente notre Contrôleur Rest.

```
@RestController
@CrossOrigin("*")
public class CinemaRestController {
    @Autowired
    private FilmRepository filmRepository;

    @Autowired
    private TicketRepository ticketRepository;

    @GetMapping(path = "/imageFilm/{id}", produces = MediaType.IMAGE_JPEG_VALUE)
    public byte[] image(@PathVariable(name="id") Long id) throws Exception{
        Film f = filmRepository.findById(id).get();
        String photoName = f.getPhoto();
        File file = new File(System.getProperty("user.home") + "/cinema/images/" + photoName);
        Path path = Paths.get(file.toURI());
        return Files.readAllBytes(path);
    }

    @PostMapping("/payerTickets")
    @Transactional
    public List<Ticket> payerTickets(@RequestBody TicketForm ticketForm){
        List<Ticket> listTickets = new ArrayList<>();
        ticketForm.getTickets().forEach(idTicket->{
            System.out.println(idTicket);
            Ticket ticket = ticketRepository.findById(idTicket).get();
            ticket.setNomClient(ticketForm.getNomClient());
            ticket.setReserve(true);
            ticket.setCodePayement(ticketForm.getCodePayement());
            ticketRepository.save(ticket);
            listTickets.add(ticket);
        });
        return listTickets;
    }
```

Nom & prénom : Fahd KORAICHE
Classe : 4IIR G2
Site : CENTRE

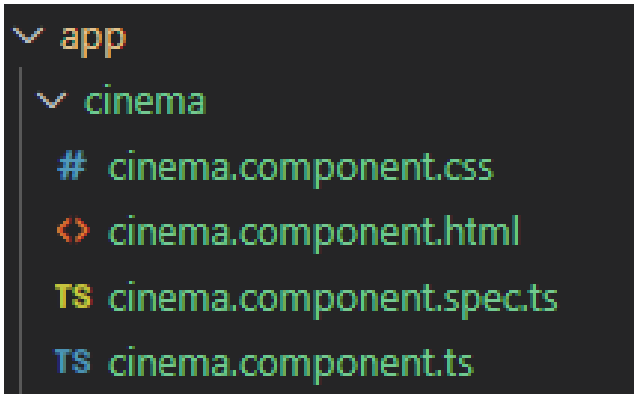
Ici au début de la déclaration on a l'annotation **@RestController** c'est une version spécialisée du contrôleur. elle inclut
les annotations ***@Controller*** et ***@ResponseBody*** et, par conséquent, simplifie l'implémentation du contrôleur: Chaque méthode de gestion des requêtes de la classe de contrôleur sérialise automatiquement les objets de retour dans ***HttpResponse*** . Puis, on a l'annotation @CrossOrigin("*"), elle sert a filtrer les sources qui peuvent accéder a notre api, dans notre cas la seule catégorie qui va accéder a notre api c'est NOTRE projet FrontEnd qu'on va voir par la suite, mais puisque on travaille toujours sur localhost et on même pas un path et un port fixe on a mis * pour autoriser toutes les requête entrantes a accéder.

Dans notre 1er requestHandler c'est la fonction image qui permet de retourner l'image correspondante a un film selon l'id du film en récupérer les photos depuis un dossier dans la machine serveur. Alors que dans notre deuxième handler c'est une méthode du type POST dont on va avoir besoin dans notre projet Frontend, c'est grâce à elle que notre projet va pouvoir communiquer avec l'api pour insérer les tickets dans la base et garantir la persistance de ces derniers.

- 4 - Représentation FRONTEND :

4.1 – Partie Angular 1 - Component:

Dans cette partie on commencer par créer un nouveau projet angular grâce a la ligne de commande, puis comme premiere etape on choisis de créer une seule et unique route '/cinema', on sait déjà que Angular favorise le concept de SinglePage Application mais dans notre cas c'est un signle page ET un single path, puisque on va rassembler toutes nos fonctionnalités dans le même component et est aussi unique dans notre projet, et c'est ca notre deuxième étape ! la création de notre component Cinema, qui se fait aussi grâce a la ligne de commande ***ng g c cinema (ng create component cinema).*** En ce qui conerne la path on doit modifier le fichier app.module.js



Nom & prénom : Fahd KORAICHE
Classe : 4IIR G2
Site : CENTRE

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { CinemaComponent } from './cinema/cinema.component';

const routes: Routes = [
  {
    path: 'cinema', component: CinemaComponent
  }
];
```

4.2 – Partie Angular 2 - Services:

Dans la deuxième partie on a introduit le principe des services et des souscriptions (subscribe). Pour pouvoir bénéficier de l'objet http qui va, par son tour, nous permettre de communiquer avec l'API qu'on a créée dans le projet Spring et profiter de l'outil JPA.

```
export class CinemaService {
  public host:string = "http://localhost:8082/";
  constructor(private http:HttpClient) { }
  public getVilles(){
    return this.http.get(this.host + "villes")
  }
  public getCinemas(v){
    return this.http.get(v._links.cinemas.href)
  }
  public getSalles(c){
    return this.http.get(c._links.salles.href)
  }
  public getProjections(salle){
    let url = salle._links.projections.href.replace("{?projection}","");
    return this.http.get(url + "?projection=p1");
  }
  public getTicketsPlaces(p){
    let url = p._links.tickets.href.replace("{?projection}","");
    return this.http.get(url + "?projection=ticketProj");
  }
  public payerTicket(dataForm){
    return this.http.post(this.host + "payerTickets",dataForm);
  }
```

Nom & prénom : Fahd KORAICHE
Classe : 4IIR G2
Site : CENTRE

Dans ce service on a un déclarer plusieurs méthode chacune d'eux utilise l'objet HttpClient. La première c'est getVilles(), elle ne prend aucun paramètre car on sait qu'on va récupérer toutes les villes existante sans filtrage, elle retourne le retour de la fonction http.get() vers la page html du component pour qu'il l'affiche. Apres, on getCinemas(), getSalles() ces deux prennent des paramètres, getCinemas() prend en paramètre l'objet JSON représentant la ville qu'on veut récupérer ses cinémas et l'utilise pour passer un paramètre adéquat a la fonction http.get(), tandis que la fonction getSalles() fait la même chose sauf que celle-ci prend un cinéma comme paramètre et non pas une ville !

On a aussi deux fonctions getProjections() et getTicketsPlaces() qui utilisent les projections ! C'est des classes qu'on a ajouté dans notre code back End pour simplifier et rendre plus clair le retour d'un service précis !

```
@Projection(name = "p1", types= {ma.emsi.cinema.entities.Projection.class})
public interface ProjectionProj {
    public Long getId();
    public double getPrix();
    public Date getDateProjection();
    public Salle getSalle();
    public Film getFilm();
    public Seance getSeance();
    public Collection<Ticket> getTickets();
}
```

```
@Projection(name = "ticketProj", types= {ma.emsi.cinema.entities.Ticket.class})
public interface TicketProjection {
    public Long getId();
    public String getNomClient();
    public double getPrix();
    public Integer getCodePayement();
    public boolean getReserve();
    public Place getPlace();
}
```

La majorité de nos sélections vont compter sur les ID alors que notre api cache ses derniers ! Pour cela on doit exposer les id des classes dont on va faire des sélections directes ! Pour faire ainsi il faut aller ver notre classe application et écrire …

```
@Override
public void run(String... args) throws Exception {
    restConfiguration.exposeIdsFor(Film.class, Salle.class, Ticket.class);
```

```
@Data
class TicketForm{
    private String nomClient;
    private int codePayement;
    private List<Long> tickets = new ArrayList<>();
}
```

Et éventuellement notre dernière fonction c'est payerTicket qui prend un objet JSON en paramètre représentant le nom du client + codePayment + tableau des IDs des tickets, cette fonction en coordonnant avec l'api elle va permettre d'insérer les tickets. Pour faire ainsi il nous faudra un objet Ticket contenant les mêmes champs dans la partie BackEnd pour qu'il puisse récupérer ses valeurs d'une manière abstraite ! Pour cella on a créé …

4.3 – Partie Angular 3 - Bootstrap:

Grace a la commande npm on doit installer les packages de bootstrap ainsi que jquery :

npm install bootstrap@3.3.7 --save
npm install jquery – save

NB : --save sert a modifier le fichier de configuration json pour enregistrer les installations des packages grâce a npm.

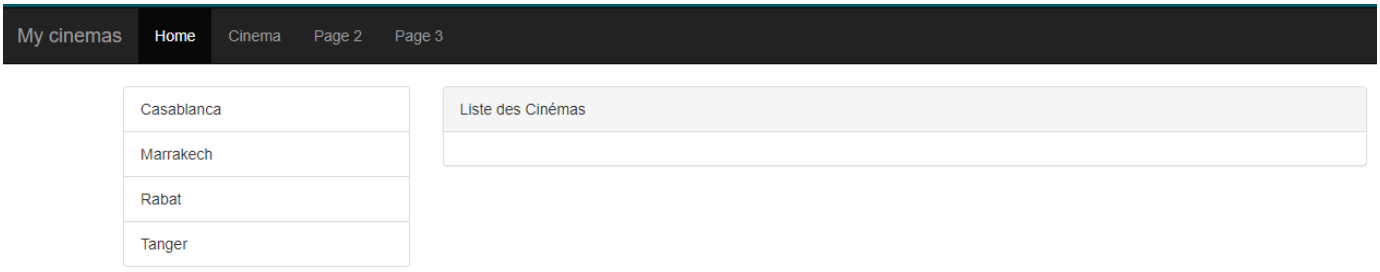
Puis on doit modifier notre fichier de configuration angular.json pour rendre les packages qu'on vient d'installer fonctionnels, on va le modifier ainsi :

```
"styles": [
  "src/styles.css",
  "node_modules/bootstrap/dist/css/bootstrap.min.css"
],
"scripts": [
  "node_modules/jquery/dist/jquery.min.js",
  "node_modules/bootstrap/dist/js/bootstrap.min.js"
]
```

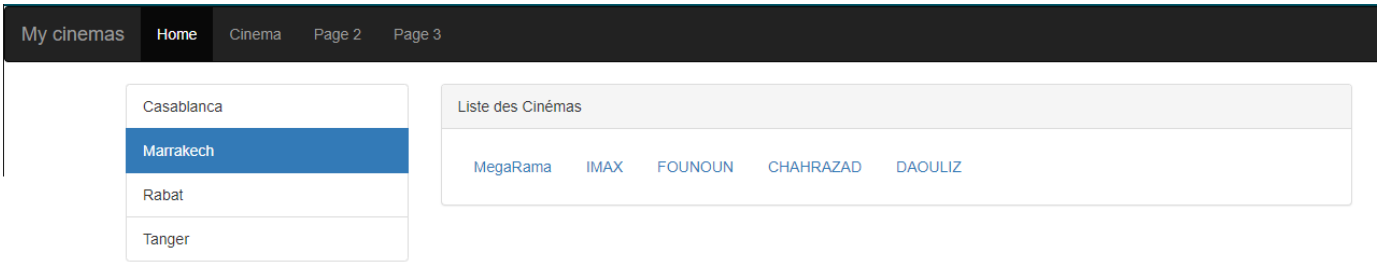
4.4 – Partie Angular 4 – Interface :

Comme j'ai cité précédemment l'application ne possède qu'une seule page (un seule component), cette page fonctionne selon des évènement onClick :

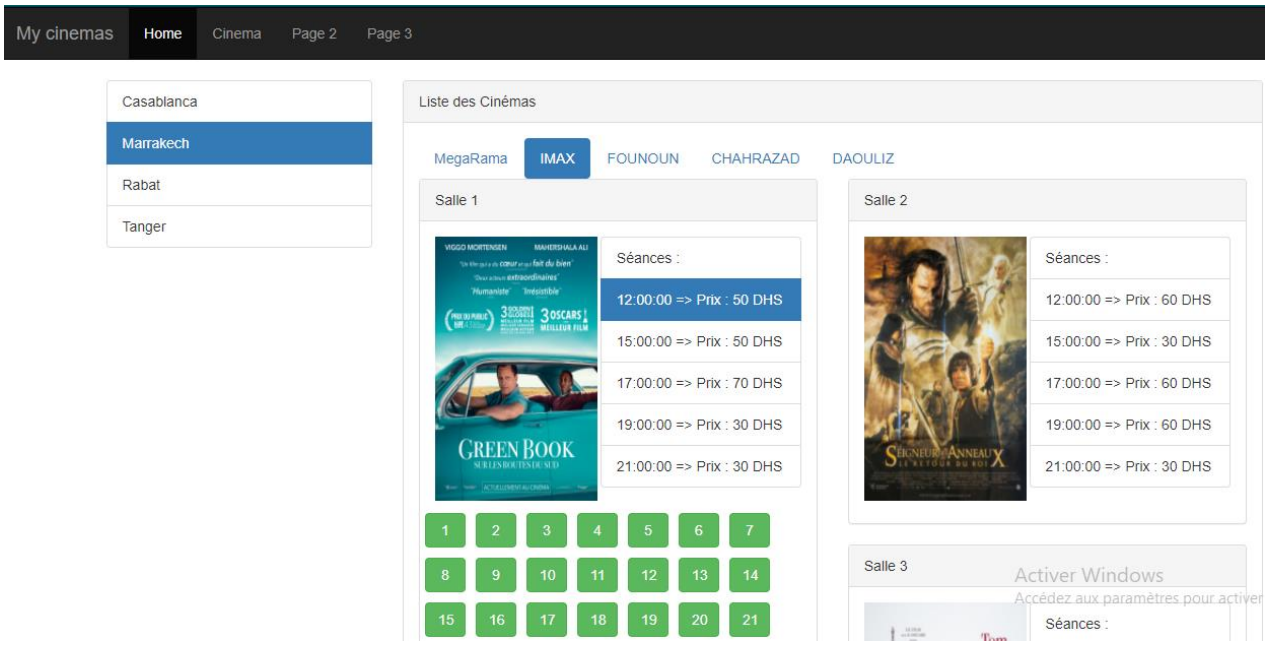
Nom & prénom : Fahd KORAICHE
Classe : 4IIR G2
Site : CENTRE



C'est la première interface qu'on voit on accédons au port :4200 ! en cliquant sur le nom d'une ville, le Card droit va se remplir de liste des cinémas existantes dans cette ville qu'on vient de cliquer :



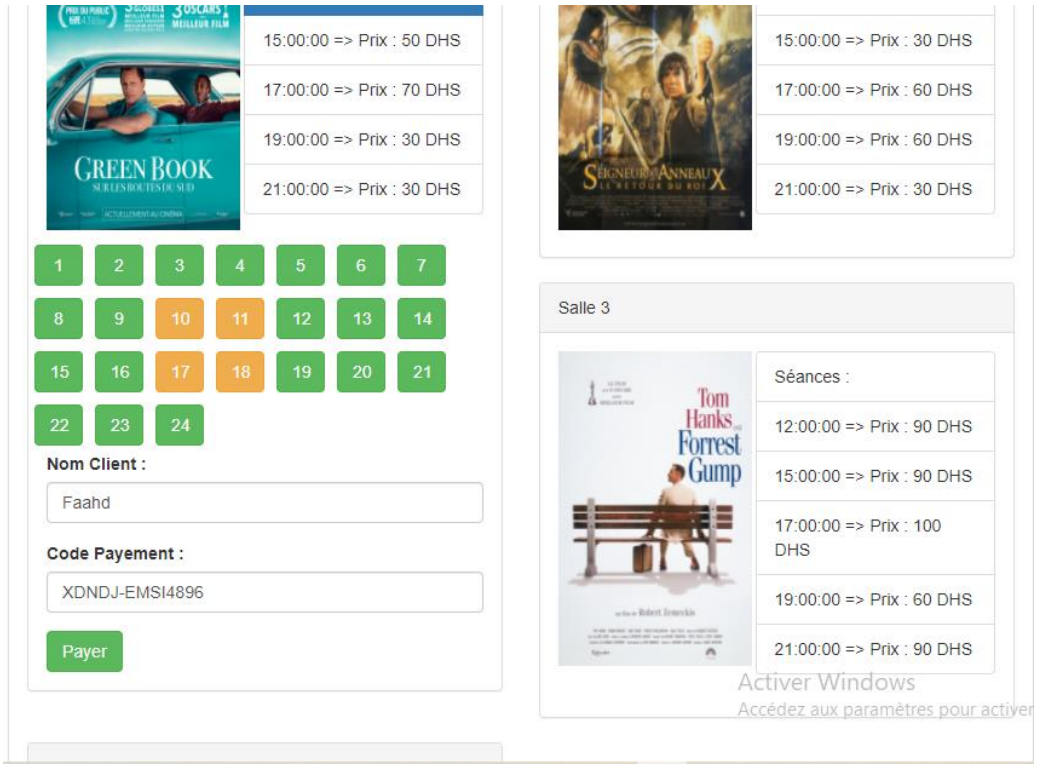
Cela fonctionne de la manière suivante : Angular nous fournit le Event Binding, c'est le faite de lier le déclenchement d'un évènement a l'exécution d'une fonction, ici suivant a l'event click sur le nom d'une ville on appelle la fonction getCinemas() en lui passant la ville dont on a cliqué comme paramètre ! Comme cela le programme va pouvoir requêter l'API de notre back end pour choisir les cinémas existante dans cette ville puis les retourner à Angular pour qu'il les affiche ! et en cliquant sur un cinéma le programme fait de meme pour afficher salles, puis ses films, puis ses projections, puis ses places et ainsi de suite !



Nom & prénom : Fahd KORAICHE
Classe : 4IIR G2
Site : CENTRE

4.5 – Partie Angular 4 – formulaire :

Jusqu'à maintenant notre application ne faisait que des requêtes GET, puisque la seule fonctionnalité c'était l'affichage, dans cette partie on a rajouté la réservation des places dans les projections à travers la création des tickets adéquats ! pour faire cela on a besoin de toujours se rappeler des places que l'utilisateur a choisis au moment avant la validation du payement. Une fois que le user clique sur le bouton payer, on va appeler le service cinema pour solliciter notre méthode POST payerTickets() qu'on a déclaré dans notre contrôleur, en lui envoyant la liste des IDs des places que le user a choisis, comme ça les autres utilisateurs ne vont pas pouvoir choisir ces dernières, et on a pas à s'inquiéter au users qui font leur chois simultanément car, ici, on a l'annotation @Transactional qui va garantir le non chevauchement des requêtes qui accèdent aux mêmes tables.



- 4 - Conclusion :

Ce projet a était d'une grande utilité, de sorte qu'on a pu implémenter tous le savoir qu'on a acquis pendent le semestre concernant l'architecture JEE et tous ses aspects depuis l'injection des dépendances jusqu'aux fonctionnalités avancées, ainsi qu'en travaillant le JEE avec le framework AngularJS, ce qui nous bien montrer la coordination entre différentes applications, ainsi que les API, pour conclure je me permet de remercier Mr El-Youssfie Mohammed, pour la riche formation qu'il nous a offert !