1^{re} année Master STL

Rapport de projet

Application web d'évènements

Groupe :
Baaloudj Hakim
Bello Pablito
Lin Veyrack

Juin 2020

Résumé

Ce document a été rédigé dans le cadre de l'unité d'enseignement PC3R. Le but est de mettre en pratique toute les notions de développement web acquises, dans la création d'une application web basé sur une API externe.

Table des matières

1 Manuel

Voici les versions compatibles utilisées lors du développement. Si vous possédez d'autres versions et que vous n'arrivez pas à lancer le serveur et le site, envisagez d'obtenir les versions compatibles.

- node : 8.10.0
- npm : 6.14.5
- tomcat : 7.0.104

1.1 Installation

Les exemples suivant sont donnés pour ubuntu 18.04, d'autre commandes équivalentes peuvent être nécessaires en fonction des différents systèmes d'exploitation.

Installation de la base de données

Installation de mysql :

```
sudo apt install mysql-server
```

Les commandes suivantes sont à effectués depuis le repertoire **server** :

```
sudo mysql
```

Puis dans la console mysql :

Création de l'utilisateur talkaboutevents identifié par le mot de passe "talkaboutevents" :

```
CREATE USER 'talkaboutevents'@'localhost' IDENTIFIED WITH
mysql_native_password BY 'talkaboutevents';
```

Création de la bdd (base de données) talkaboutevents :

```
CREATE DATABASE talkaboutevents;
```

Dons de privilèges sur la bdd :

```
GRANT ALL ON talkaboutevents.* TO 'talkaboutevents'@'localhost';
```

Remplissage de la bdd (assurez vous d'avoir lancer mysql depuis le dossier server et de vous être placé au préalable sur la bonne bdd via la commande suivante) :

```
USE talkaboutevents;

SOURCE talkaboutevents.sql;
```

Lancement de serveur Le serveur est lancé via Eclipse, le projet est déjà configuré, ayant pour dossier source **server**. Une fois sur Eclipse, faire : *Open project from filesystem* et sélectionner le dossier **server**. On pourra lancer le serveur embarqué tomcat (qui se lance sur le port 8080) via la vue *servers* d'Eclipse : *window > show view > servers*

Lancement du client Pour lancer le client, vous devez installer npm & nodejs :

```
sudo apt install nodejs
sudo apt install npm
```

On installe ensuite depuis le dossier **client** les paquets npm utilisés par le projet :

```
sudo npm install
```

On peut maintenant lancé le client toujours depuis le dossier **client** en exécutant :

```
npm start
```

Une fois le client lancé (il se lance sur le port 3000), votre navigateur devrait ouvrir une page internet, s'il ne le fait pas, vous pouvez accéder à l'url : *http ://localhost :3000/*.

Le client et le serveur peuvent maintenant communiquer.

1.2 Utilisation

Un exemple d'utilisation classique pour tester l'ensemble des fonctionnalités peut être le suivant :

- Ouvrir deux pages web *http ://localhost :3000* dans deux navigateurs différents. Les routes étant protégées, vous serez redirigés sur */signin*. Aller sur l'inscription en bas de page et créer deux comptes. Puis se connecter avec les deux comptes.
- Nous arrivons sur la page de recherche d'évènements, chercher un évènement avec un mot clé **"concert"** et sans préciser de ville. Puis cherche un évènement en précisant une ville, **"paris"** par exemple. Appuyer sur s'inscrire à des évènements.
- Aller sur la page profil depuis la barre de navigation, ici s'affichent les évènements auxquels vous êtes inscrits. Attention, certains évènements peuvent ne pas apparaître, en effet certains appels à l'api pour récupérer un évènement depuis son id échouent, cela étant indépendant de notre code. Lors de l'affichage, on n'affiche pas les évènements dont les appels ont échoués, vous pouvez recharger la page si besoin. Essayer de vous désinscrire de certains évènements.
- Depuis votre page profil, accéder à la modification de profil, vous pouvez ajouter une photo en format *.png*, une description, et votre pseudo.
- Inscrivez-vous à au moins un même évènement avec les deux comptes (sur deux navigateurs séparés), ou rejoignez le salon de chat d'un même évènement.
- Les deux navigateurs côte à côte, envoyer un message depuis l'un des deux. Il arrive instantanément sur l'autre. Échangez quelques messages. Recharger la page sur l'un des comptes, les messages seront toujours la grâce à la sauvegarde en bdd.
- Depuis la page internet du compte A, cliquez sur le pseudo du compte B (il s'affiche sur les messages du compte B), vous arriverez sur son profil, vous ne pourrez bien sûr pas modifier son profil. Vous pouvez voir les évènements auxquels il participe mais pas supprimer ses participations.
- Sur un des comptes appuyer sur Déconnexion, vous êtes redirigés sur */signin*, essayer d'accéder à une des routes, par exemple la racine, vous ne pourrez pas y accéder. La session a bien été détruite. Depuis l'autre compte retourner sur votre page de modification de profil, supprimez le compte. Essayer de vous connecter avec les identifiants que vous venez de supprimer : cela est désormais impossible.

2 Description du projet

L'objectif du projet est la création d'une application web basée sur une API externe, Le sujet de l'application et les fonctionnalités proposées sont libres, tant que l'application respecte certaines contraintes que nous allons décrire plus bas.

2.1 Contraintes

- Le serveur doit être écrit en Servlets naturelles (pas d'utilisation de framework) et doit s'exécuter dans Tomcat/Catalina.
- Pour le client, les bibliothèques Javascript usuelles (jQuery, Bootstrap,..) sont autorisées.
- L'application doit proposer du contenu généré par les utilisateurs, un bon exemple est l'intégration d'une composante sociale (profils, commentaires, notes, messages, publications,..)
- Le serveur de l'application doit contacter de manière régulière (en réaction à une requête, ou à des dates précises, par exemple "tous les soirs") une API Web externe, dynamique déjà existante (dynamique signifie que son contenu doit changer régulièrement)
- Les requêtes entre le client et le serveur sont majoritairement asynchrones (AJAX) et le serveur répond avec des données encodées en *JSON*.

3 Dossier

Par la suite vous trouverez une retranscription du dossier que nous avons rendu pour le TME6, le tout ayant été mis à jour.

Tout d'abord, concernant l'API externe que nous utilisons pour récupérer les informations concernant les événements, l'API **Ticketmaster** sera utilisé :

`https://developer.ticketmaster.com/products-and-docs/apis/getting-started/`

Les accès à l'API se font par clé d'authentification (gratuite). Il y a une limite de 5 000 requêtes par jour pour l'offre gratuite, ce qui sera suffisant dans le cadre de ce projet.

L'API permettra donc de retrouver des événements (notamment par lieu) et les informations liées à ceux-ci (nom de l'événement, photos, date...).

Le but est d'afficher de façon très sobre les informations principales concernant un événement et de mettre l'accent sur la discussion autour de l'événement.

Ensuite, voici la structure des données que nous pouvons consulter

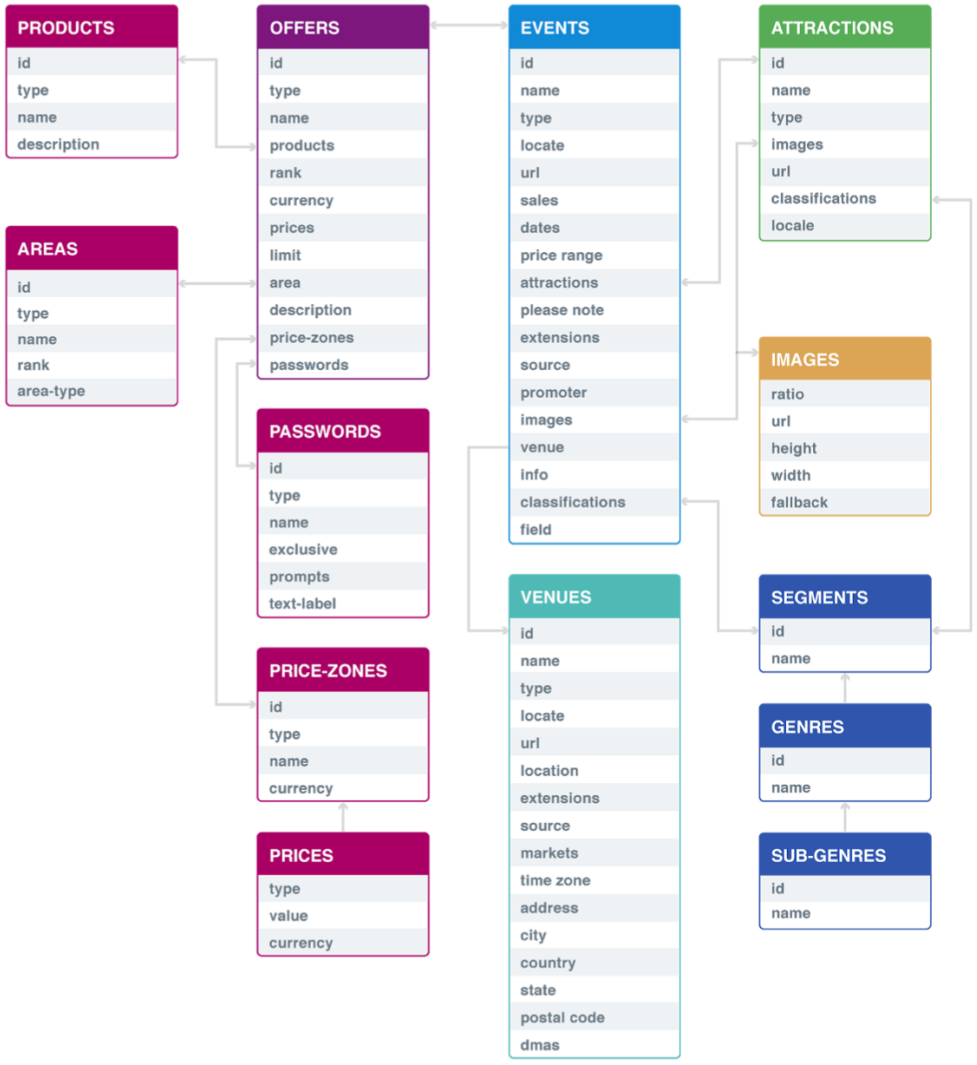

FIGURE 1 – Structure de la base de données

3.1 Fonctionnalités

— Authentification

— Modification (photo de profil, description..) / Suppression de profil

— Récupérer une liste d'événement à partir d'un lieu et mot-clé

— Indiquer une participation à un événement ou retiré une participation.

— Pouvoir retrouver les événements auxquels on participe sur sa page profil avec possibilité d'accéder au salon de chat de l'évènement ou se désinscrire de l'évènement.

— Pouvoir envoyer des messages sur le salon de chat de chaque événement.

3.2 Base de données

La base de données n'aura pas besoin de contenir les informations relatives aux évènements. En effet elles seront récupérées via l'API à chaque recherches d'événements effectuées par l'utilisateur.

Les informations suivantes devront néanmoins être stockées :

- Les informations relatives à l'utilisateur devront être stockés : identifiant, mot de passe, description, photo de profil, etc.
- Les informations relatives à la participation aux évènements : id d'événement (donner par L'API), id de l'utilisateur participant.
- Les informations relatives aux salons de chat : id d'événement (un salon public par événement), messages dans le salon et pour chaque message, l'id de l'utilisateur qui a posté le message.

3.3 Mise à jour et appel à l'API

L'API sera appelé uniquement pour récupérer les événements et informations associées, notamment lors de la recherche, l'utilisateur pourra entrer un lieu ou chercher autour de lui (coordonnées gps utilisées).

La base de données sera mise à jour dans les cas de figures suivants :

- Inscription / suppression de compte / modification de profil : Mise à jour de la partie utilisateur.
- Clique sur "participe"/"ne participe plus" sur un évènement : Mise à jour de la partie participation aux événements.
- Envoie un message dans un salon de chat : Mise à jour de la partie salons de chat

3.4 Description du serveur

Le serveur offrira une API REST pour pouvoir interagir avec les données. Les ressources principales seront les suivantes :

- utilisateurs (/users) : créer un compte, voir un profil, modifier son profil/le supprimer.
- événements (/events) : voir des événements, participer/ne plus participer à un événement.
- Chats (/rooms) : voir les messages, poster un message.

L'API serveur proposera de faire des requêtes *CRUD* classiques via *HTTP*. La transmission de données sera faite au format *JSON* : informations utilisateurs, événements, messages

3.5 Description du client

Le client sera un site monopage, avec quatre "écrans" principaux :

- Création de compte / Connexion : fera des appels sur /users
- Événements (page principale) : Possibilité de chercher des événements (appel sur /events). Chaque événement possédera un bouton participé /ne plus participer (appel sur /events) ainsi qu'un bouton pour ouvrir le salon chat de l'événement.
- Profil : consulter le profil de quelqu'un ou son profil, si on est sur son profil, on peut le modifier : appels sur /users.
- Chats : voir les messages dans le salon, poster un message dans le salon : appels sur /chat.

4 Architecture

Voici l'architecture du projet présenté à travers la séparation client/serveur et l'organisation de leurs fichiers respectifs

4.1 Serveur

Le dossier **src** contient les sources du serveur, voici les différents packages et leur description.

- chat : contient toutes les classes pour l'échange de messages dans un salon de chat.
- db : contient toutes les classes pour les requêtes sur la bdd.
- event : contient une classe faisant des requêtes sur l'api, et une servlet pour la gestion des événements.
- members : contient les servlets gérant l'inscription / connexion / déconnexion.
- participation : contient le servlet gérant la participation aux événements.
- security : contient le gestionnaire de CORS et le gestionnaire de mots de passes pour les crypter / comparer.
- user : contient la servlet pour gérer les utilisateurs.

4.2 Client

Le dossier **src** contient un sous-dossier style contenant toute les pages css. Il contient aussi les différents fichiers *.js* :

- Chat.js : Composant chat, représente un salon de chat, permet l'envoie et la réception de messages.
- Config.js : Configuration du client, variable global pour le préfixe de l'URI du serveur.
- EditProfil.js : Composant de modification de profil.
- Events.js : Composant d'évènements, permet la recherche et l'affichage de ceux-ci.
- index.js : Point d'entrée du programme
- Main.js : Gère les routes du client.
- Navigbar.js : Composant de la barre de navigation.
- PrivateRoute.js : Composant servant à la gestion de la sécurité des routes.
- Profil.js : Composant d'affichage du profil.
- Signin.js : Composant de connexion.
- Signout.js : Composant de déconnexion.
- Signup.js : Composant d'inscription.
- User.js : Stockage des informations de l'utilisateur connecté.

5 Implémentation

5.1 Points forts, spécificités et challenges

Une spécificité que nous souhaitons mettre en avant est la gestion de la sécurité coté client. Un utilisateur non connecté doit avoir accès seulement à l'inscription et à la connexion.

On créer ainsi un système de route privée, le composant Main définit les routes vers tous les composants :

```
<Switch>
    <PrivateRoute exact path="/" component={Events} />
    <PrivateRoute path="/chat" component={Chat} />
    <PrivateRoute path="/signout" component={Signout} />
    <PrivateRoute path="/profil" component={Profil} />
    <PrivateRoute path="/editProfil" component={EditProfil} />
    <Route path="/signin" component={Signin} />
    <Route path="/signup" component={Signup} />
</Switch>
```

Ici, on voit que les composants non-sécurisé sont représentés par un composant "Route", ayant pour propriété le chemin et le composant à renvoyer. Pour les PrivateRoute, on agit de la même façon, on passe le chemin de la route et le composant qui doit être renvoyé si l'accès à la route est autorisé.

Regardons de plus près le composant PrivateRoute :

```
state = {
    loading: true, // la page est en train de charger
    isLoggedIn: false
};

componentDidMount() {
    //on verifie si l'utilisateur est connecté
    User.getIsLoggedIn().then(islog => {
        this.setState({
            loading: false,
            isLoggedIn: islog
        });
    });
}
```

À la création du composant, on indique via l'état *loading* que la page est en train de charger, et via l'état *isLoggedIn* que l'utilisateur n'est pas connecté.

Avant que le composant soit monté, on lance un appel sur *User.getIsLoggedIn()*, cette méthode asynchrone renvoie **true** si l'utilisateur est connecté, sinon elle vérifie auprès du serveur qu'il existe une session ouverte pour l'utilisateur et stocke les informations de l'utilisateur. Une fois qu'on obtient le résultat de cet appel, on met à jour l'état : la page ne charge plus et *isLoggedIn* prend pour valeur le retour de l'appel de fonction.

On peut s'intéresser à la méthode render :

```
render() {
    const { component: Component, ...rest } = this.props;
    return (
        <div className="mainDiv">
            <Navigbar islog="true" />
            {/* on redirige vers la bonne route une fois que
            loading est fixer a false, en fonction de isLoggedIn */}
            <Route
                {...rest}
                render={props =>
                this.state.isLoggedIn ? (
                    <Component {...props} />
                ) : this.state.loading ? (
```

```
                <div>LOADING</div>
            ) : (
                <Redirect
                    to={{
                        pathname: "/signin"
                    }}
                />
            )
        }
        />
    </div>
);
```

Pour effectuer son rendu, la PrivateRoute vérifie d'abord si l'utilisateur est connecté, si c'est le cas on affiche le composant qu'on avait passé en propriété de PrivateRoute, sinon on regarde si la page est encore en chargement, si elle charge on l'affiche, sinon ça veut dire qu'on a déjà vérifié le statut de connexion de l'utilisateur et que celui-ci n'est pas connecté, on redirige vers la route */signin*.

Un point fort que nous souhaitons mettre en avant est le système de salons de chat. Le système de chat repose sur le protocole websocket, qui permet notamment au serveur d'envoyer des paquets vers le client. Le client pourra ainsi afficher les messages reçus sans avoir à recharger la page.

Voyons le code coté client :

```
// création de la connexion
this.socket = new WebSocket("ws://localhost:8080/TalkAboutEvents/chat/"
                            + this.toId);

// evenement reception message
this.socket.addEventListener("message", async (event) => {
  const message = JSON.parse(event.data);
  // on récupère des infos sur l'expéditeur
  const user = await this.getUser(message.from);
  message.from_pseudo = user.pseudo;
  message.from_pdp = user.pdp;
  message.from_id = user.id;
  this.addMessage(message); // ajout du message à la liste des messages
})
```

Dans le constructeur du composant Chat, on crée un nouvel object WebSocket vers l'adresse du serveur traitant des websockets, en rajoutant l'id du salon de chat concerné, ici représenté par this.toId, l'id de salon de chat est simplement l'id de l'événement associé au salon.

Ensuite, on ajoute un écouteur d'événements sur "message", ainsi à chaque fois qu'un événement "message" ce produit, la fonction asynchrone passée sera appelée. Cette fonction récupère le contenu du message. On complète ensuite quelques informations sur l'expéditeur du message afin de pouvoir les afficher avec le message, puis on ajoute le message à la liste des messages du composant (contenu dans l'état de celui-ci).

Lors de l'envoi d'un message, après l'avoir saisi et appuyer sur entrer, ce code est appelé :

```
const toSend = {
    from: User.getId(),
    to: this.toId,
    message: this.state.message
};
this.socket.send(JSON.stringify(toSend));
```

On voit que le message contient trois champs. "from" est l'id de l'expéditeur, "to" l'id du destinataire, qui est donc un id d'événement, et "message" le contenu du message. Puis cet objet serialisé et envoyé sur le websocket.

C'est tout pour le code client ! Voyons maintenant côté serveur. La classe la plus importante dans le traitement des messages est la classe "ChatEndpoint", qui possède notamment la structure de données suivante :

```
private static Map<String, Set<Session>> rooms =
    Collections.synchronizedMap(new HashMap<String, Set<Session>>());
```

C'est ici que tous les messages vont être stockés. La Map recense l'ensemble des salons, à chaque identifiant de salon (String) est associé un ensemble de Session, chaque Session représente le websocket sur lequel l'utilisateur peut être contacté.

Trois méthodes entrent ensuite en jeu :

```
@OnOpen
public void onOpen(Session session, @PathParam("room") String room)
throws IOException {
        // on vérifie si la room existe déjà
        if (rooms.containsKey(room)) {
                logger.debug("Un utilisateur rejoint la room : " + room);
                rooms.get(room).add(session);
        } else { // la room n'existe pas
                 // on créer la room
```

```
                logger.debug("Création de la room : " + room);
                Set<Session> peers =
                    Collections.synchronizedSet(new HashSet<Session>());
                rooms.put(room, peers);
                // on ajoute la session dedans
                logger.debug("Un utilisateur rejoint la room : " + room);
                peers.add(session);
        }
}

@OnMessage
public void onMessage(Session session, Message message)
throws IOException, EncodeException {
        logger.debug("Message reçu : (from :  " + message.getFrom() + ", to : "
        + message.getTo() + "), content : " + message.getMessage());
        // on recupère le salon destination
        String room = message.getTo();
        for (Session peer : rooms.get(room)) {
                // envoie du message à tout les membres de la room
                peer.getBasicRemote().sendObject(message);
        }
        // on stocke le message en bdd
        try {
                DBMessage.addMessage(message);
        } catch (SQLException e) {
                e.printStackTrace();
        }
}

@OnClose
public void onClose(Session session, @PathParam("room") String room)
throws IOException, EncodeException {
        logger.debug("Un utilisateur part de la room : " + room);
        // on supprime la session de la room
        rooms.get(room).remove(session);
        // si la room est vide on la supprime
        if (rooms.get(room).isEmpty()) {
                logger.debug("La room " + room + " est vide, suppression.");
                rooms.remove(room);
        }
}
```

Nous allons décrire les méthodes une par une :

1. onOpen est appelée lorsque la websocket est crée côté client, son rôle est d'ajouter l'utilisateur au bon salon de message, si le salon n'est pas encore créé, on le fait.
2. onMessage est appelée lorsque le client envoi un message sur la websocket, son rôle est de transmettre le message à tous les utilisateurs connectés dans le salon. On récupère l'id du salon qui est stocké dans le message, et l'envoie a tous les membres de l'ensemble. Puis le message est sauvegardé en bdd.
3. onClose est appelée lorsque la websocket se déconnecte. On retire l'utilisateur du salon, et si le salon est vide, on le supprime, il sera recrée quand un nouvel utilisateur le rejoindra.

D'autres classes rentrent en jeux, notamment la classe Message qui contient la structure d'un Message, et les classes MessageEncoder et MessageDecoder permettent de sérialiser, dé-sérialiser un message automatiquement.

On peut voir que la mise en place de websocket n'est pas particulièrement compliqué, mais permet des utilisations très intéressantes quand le serveur a besoin d'envoyer des informations au client.

6 Conclusion

Les enseignements que nous avons suivis au cours de ce semestre nous ont permis de mettre un pied dans le monde du développement web et ses idiomes, et ce projet nous a permis de mettre en pratique les notions acquises et de découvrir des technologies populaire (react, servlets).

De plus, nous nous sommes documenté sur les pratiques courantes dans le développement d'application web (sécurité, techniques de cryptage des mots de passe etc...), donc nous avons appris beaucoup de choses.

L'application aurais pu bénéficier de plus de fonctionnalités mais les circonstances étant difficile, en plus de la gestion du temps difficile entre tout les projets. Cela dit, nous sommes plutôt satisfait car nous avons un résultat assez proche de ce qu'on avais imaginé.