Site web PC3R .
Compte rendu du projet de fin de semestre.

Rozovyk Artemiy
rozovyk.artem@gmail.com

June 11, 2020

1 Introduction

Dans ce rapport je vais vous présenter les points forts du projet en étant le plus concis possible. Le projet devait être fait à deux, mais pour des raisons qui me sont inconnues mon binôme a décidé de faire son projet à part. Donc, tous le travail que je vous présente a été fait exclusivement par moi-même. Malgré le nombre de fonctionnalités relativement bas, ce projet contient un large spectre de techniques qui, avec réitèration peuvent mener à un site web de taille considérable. Je vais vous parler des nombreux challenges techniques que j'ai du surmonter, notamment car j'utilise `React.js` pour la partie frontend et `Hibernate` pour la gestion de la BDD.

2 Installation

La partie back est un projet web java standard plus maven (le pom.xml est inclus) . Normalement la seule modification à faire pour vous sera changer les identifiants pour la connexion vers la BBD mysql: dans les fichiers `src/main/resources/hibernate.cfg.xml` et `src/main/java/util/HibernateUtil.java`.

```
<property name="connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost:3306/...des trucs
            ...</property>
        <property name="connection.username">root</property> $$changer$$
        <property name="connection.password">password</property> $$changer$$
```

Ensuite il a été déployé sous Intellij avec `Tomcat 9.0.34` . Et le lien sous lequel le projet était déployé est :

```
http://localhost:8080/ServletSample_war_exploded/
```

Pour la partie front normalement

```
sudo npm i -g --unsafe-perm now
sudo npm start
```

doivent suffire. Puis accedez en **http://localhost:3000/** (`login:a | password:a` sont disponibles par défaut)

2.1 Scénario d'exécution fonctionnel

Commencer par **http://localhost:3000/**, se connecter avec login **a** et mot de passe : **a** . Clic sur `Forum`, dans le formulaire, choisissez un sujet et tapez un message (parfois il faut en envoyer deux pour qu'il s'affiche, ou bien clickez sur `"Home"` et revenez dans `"Forum"`). Le message s'affiche, clic sur `Consulter` à droite dans la table. La file de discussion va s'ouvrir. Il n'y a pas encore de réponses. Taper une réponse et cliquez sumbit. Elle s'affiche directement car ici j'ai bien géré le dataflow de React et la concurrence des appels à l'api. À coté du message cliquez sur Ouvrir du *Modifier*, un formulaire s'ouvre. Tapez votre réponse et cliquez sur modifier. Ici, pas de mise à jour automatique, car j'ai mal structuré les composants. Pour voir la modification, cliquez sur forum en haut de la page (ou sur le sidebar) et revenez dans la file de discussion. Le message a bien été modifié. Supprimez le et faites la même demarche pour voir qu'il a bien été supprimé.
Maintenant, cliquez sur `Echanges monnaie` sur le sidebar. Le portefeuille de l'utilisateur s'affiche (en contactant le serveur pour le portefeuille correspondant à l'id de l'utilisateur). Dans `Echanges | Banque centrale |`[1] cliquez sur le menu déroulant, où il y a USD par défaut, et choisissez une autre devise, cliquez `Changer`. Les taux sont mis à jour suite à une requête dynamique à mon serveur(qui lui-même contacte une api externe). Je n'ai pas implémenté les autres fonctionnalités de cette page car elles sont identique à la gestion du forum[2]. Voici ce qui reste à faire en grandes lignes: Pour acheter depuis la banque centrale, envoi du formulaire en post en cliquant sur acheter. Définir une entité offre définie par devise offerte, devise demandée, montant, taux, et le vendeur(OneToOne avec User). Puis faire le calcul nécessaire lors de la soumission des deux formulaires : `Soumettre une offre` et `Offres d'échange particulier`.

3 Architecture

3.1 Backend

Comme exigé, il s'agit d'un serveur en Java Servlet. Les 3 servlets gèrent:
les utilisateurs - `UserAccountGestionnaire`, la communication entre utilisateurs à travers un "forum" avec des files de conversation - `ForumGestionnaire` et une pour les Échanges de monnaie - `ExchangeGestionnaire`. Les points d'entrées sont en `REST`:

[1] Où les donnés actuelles correspondent au fetch initial du serveur (qui est périodique, voire la partie back-end)

[2] Cependant, même si je ne fait pas de requêtes pour la soumission des formulaires, je récupère bien les information dedans, on peut s'en apercevoir en regardant la console ou elles sont affichés si on clic sur Sumbit une offre par exemple.

GET /forum	Récupérer les files de discussion
POST /forum	Créer une nouvelle file de discussion
GET /forum/:id	Récupérer une file de discussion par id
POST /forum/:id	postuler une réponse à la file de discussion
GET /forum/:id/reponses	Récupérer toute les réponses de cette file de discussion
PUT /forum/:idm/responses/:idr	Modifier une réponse
DELETE /forum/:idm/reposnes/:idr	Supprimer la réponse
GET /users	Tous les utilisateurs
GET /users/:id	Un utilisateur
POST /login	Créer une connexion (se connecter)
POST /register	Créer un compte
GET /exchanges/:id	Récupérer le portefeuille d'un user
GET /rates/:curr	Les taux d'échange pour une devise donnée(Api ext.)

Coté sauvegarde de donnés, j'utilise `ORM Hibernate`[3]. Les entités les plus intéressantes sont des entités composés :

```
@Entity
@Table(name="forum_message")
public class ForumMessage {
    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    @Column(name="forum_message_id")
    private int forum_message_id;
    @OneToMany(cascade = CascadeType.MERGE,
               fetch = FetchType.EAGER,
               mappedBy="correspondingMessage")
    private Set<ReponseForum> items;
    ...
```

```
@Table(name="response")
public class ReponseForum {
...
    @ManyToOne(cascade = CascadeType.MERGE,fetch = FetchType.EAGER)
    @JoinColumn(name="forum_message_id", nullable=false)
    ForumMessage correspondingMessage;
```

Les 5 entités définies (`ForumMessage, Monnaie, Portefeuille, ReponseForum, User`) possèdent chacune un Data Access Object correspondant qui effectue les opérations CRUD. Leur utilisation peut être vue dans les méthodes do* des Servlets.

3.1.1 Api externe

J'utilise une seule API externe exchangerate-api.com. À titre pédagogique je l'utilise à la fois périodiquement et dynamiquement. Périodiquement : avec une sous-classe de

[3] Assez difficile à mettre en place correctement: la notion de transaction, rollback en cas d'erreur, SessionFactory, API pour les requetes SQL, intergration dans un projet servlet, entity mapping

`ServletContextListener`, notamment la classe `ExchangeRateFetcher`, que j'ai défini, qui a un attribut `ScheduledExecutorService` qui programme un fetch des taux d'échange avec `scheduleAtFixedRate(new FetchRates...)` où `FetchRates` est un `Runnable` qui contacte l'api et sauvegarde la réponse dans `ServletContext`. La devise de référence pour le fetch périodique est USD.
Nous pouvons contacter l'API de manière dynamique (depuis une méthode dans ExchangeGestionnaire) afin de préciser la devise de référence
(rôle du point d'entrée : *GET /rates/:curr*).

3.2 Frontend

Pour la partie frontend j'ai décidé d'utiliser `React.js` [4]. En s'inspirant d'un template fourni par coreui.io, j'ai pu faire une structure qui fonctionne dans sa globalité. Cependant, étant à mi-chemin dans le développement, je me suis rendu compte que c'était probablement un "overkill" d'utiliser quelque chose d'aussi poussé pour un "mini-projet" pédagogique, car même pour faire des choses très basiques on doit maîtriser des notions assez complexes.

3.2.1 Structure projet

La génération du code en react est organisée en "composants" qui ont un état qui évolue grace aux "handlers" attachés aux éléments internes. Des "props" (les inputs provenant d'un niveau supérieur[5] et qui definissent le "dataflow". Les composants ont une fonction `render` qui défini la manière dont le composant sera présenté à l'écran.
Le point d'entrée du programme est le fichier `src/index.js` qui initialise le composant principal `src/App`. À l'intérieur de `App` un premier routage redirige vers `src/containers/DefaulLayout` si on est authentifié ou vers `Login` sinon. `DefaulLayout` est le conteneur englobant les autres composants : header, side etc.. Sur le `AppSidebarNav` j'ai défini 2 rubriques Forum et Echanges monnaie. C'est donc dans ces deux rubriques que sont contenu les 9 composants que j'ai défini dans le dossier `src/views/Base`

3.2.2 Exemple de composant présentant les notions:

Pour ne pas citer toute les petites choses que j'ai du faire dans les 9 composants que j'ai construit "from scratch", je vais vous présenter le plus intéressant:
`src/views/Base/Forum/ForumSingle.js`

```
class ForumSingle extends Component {...
this.state = {
      messageToSend: '',
      forumMessage: {},
      correspondingResponses: [],
```

Listing 1: 3

[4] Pour quelqu'un qui n'as jamais fait de js, c'était une sacré introduction.
[5] Il faut garder en tête toute l'arborescence des composants pour ne pas s'y perdre

L'état du composant comporte le message à envoyer mis à jour par le
`<Input onChange={(e) = >this.handleChange(e)}`
Le message du forum (objet json avec l'auteur, le message, et le sujet) est initialisé dès que le composant est installé, dans
`componentDidMount()`
Cette méthode fait un appel asynchrone via la librairie connue axios qui se base sur les promesses.

```
axios.get('monapi/forum/${id}').then(response => {
    this.setState({forumMessage: response.data})
}).catch(function (error) {
    console.log(error);
});
```

Listing 2: 3

Quand le résultat du get sera disponible, on exécutera se qui est dans .then. La fonction setState force le composant à se (rerender). En réalité, je pense qu'on arrive à la page ou ce composant se trouve, il comporte pas de message initial pendant une fraction de seconde, puis il apparaît suite à la réussite de la méthode get.
`correspondingResponses` est initialisé de même manière, par un appel asynchrone à GET /forum/id/reponses.
Finalement pour l'envoie d'un nouveau message, un point de concurrence que j'ai pas toute de suite compris:
La soumission, declanché par `<Form onSubmit={(e) =>this.submitForm(e)}`
fait un appel à POST, pour envoyer le message. Puis, afin que le composant se "rerender" on doit récupérer les `correspondingResponses` contenant également le message nouvellement posté. Pour cela, nous devons faire l'appel à GET /reponses/ **après** que POST a réussi:

```
axios.post(`api/id/reponses`, {..data..}).then(response => {
         axios.get(`forum/${id}/responses`).then(response => {
            this.setState({correspondingResponses: response.data})
            alert("message has been sent")
         })
}
```

Listing 3: 3

3.2.3 Difficultés

React est destiné à faire des sites mono-page, c'est pourquoi il était difficile de faire les choses correctement car toute l'expérience en web que j'ai est basée sur la notion de navigation entre les différentes pages.
Je dois avouer que j'ai compris que très tardivement comment fonctionne le dataflow en React et comment mettre à jour (et "rerender") les composants enfants(voir parents depuis

enfant) suite à la modification dans le composant parent (et vice versa) [6], c'est pour cette raison que les mises à jour de l'état du composant ne rafraîchissent pas les donnés de certains composants et on doit le faire manuellement (en clinquant sur `Échanges monnaie` ou `Forum`, ce qui déclenche un rerender des composants). L'exemple qui montre que j'ai compris peut être vu à la ligne 31 de `Forum.js`, en fait c'est une référence qui est passé au composant enfant et qui permet d'appeler une méthode (qui met à jour l'état et déclenche le rerender) du détenteur(`ForumMessages`) de l'objet et ce depuis le composant créateur `Forum`.

[6] Les tentatives de restructuration cassait très facilement le tout, donc ont été abandonnées.