TP2 - Spring Boot

Vous utiliserez Spring Boot pour créer le backend de l'application de messagerie que vous avez commencée lors du TP1.

Prérequis

Vous devez avoir la JDK 17 de java installée sur votre machine.

Voir https://www.oracle.com/java/technologies/downloads/#java17.

Base de code

Vous allez réutiliser le même dépôt de code que lors du TP1.

Avant de commencer, créez 2 répertoires à la racine nommés `frontend` et `backend`.

Déplacez l'ensemble de vos fichiers du TP1 dans le répertoire `frontend` à l'exception du fichier `equipe.txt` que vous garderez à la racine.

Vérifiez que vous pouvez toujours exécuter votre application Angular en exécutant `ng serve` à partir du répertoire `frontend`.

Vous trouverez la base de code à copier dans le répertoire `backend` dans le dépôt gitlab suivant: https://gitlab.info.uqam.ca/trepanie_fel/tp2-base

Une fois l'ensemble des fichiers copiés, vous devriez pouvoir exécuter la commande `./mvnw clean spring-boot:run` à partir du répertoire `backend`. Cette commande devrait compiler le code du serveur et le démarrer localement en utilisant le port 8080.

Comme pour le TP1, la base de code est incomplète.

Objectif

À la fin de cet exercice, votre application devra utiliser le serveur pour effectuer les opérations suivantes:

- connexion et déconnexion (login et logout)
- récupérer les messages
- publier un nouveau message
- recevoir une notification lorsqu'un nouveau message est publié

Pour cette deuxième version, les messages seront maintenant stockés en mémoire dans le serveur. Donc, si vous rafraîchissez la page web, les messages devraient toujours être disponibles. Ils disparaîtront lorsque vous redémarrez le serveur.

Pour compléter l'application, vous devrez suivre les étapes décrites dans les sections suivantes.

Prenez le temps de vous familiariser un peu avec la base de code avant de faire les modifications.

Partie 1 - Connexion et Déconnexion

Le première partie consiste à implémenter la connexion et la déconnexion.

Étape A - Backend - AuthController - login

Écrivez le code pour la fonction `login` qui devra répondre au chemin d'accès `/auth/login`.

Cette fonction ne validera pas le mot de passe pour ce TP.

Lors de la connexion (login), créez un nouvel objet de type `SessionData` qui représente la session de l'usager. Cet objet devra être ajouté comme session dans le `SessionManager` en utilisant la méthode `addSession` de l'instance du `SessionManager` qui est injectée. Enfin, la fonction devra retourner au client un objet de type `LoginResponse` qui contient l'identifiant de session (token) obtenu lors de l'appel à `addSession`.

Étape B - Backend - AuthController - logout

Écrivez le code de la fonction `logout`.

Cette fonction doit récupérer l'identifiant à partir du `ServletContext` (contexte de la requête HTTP) en utilisant la méthode `getToken` du `SessionDataAccessor` et retirer la session du `SessionManager` pour indiquer que la session n'est plus active.

Étape C - Frontend - LoginService - login

Dans le code Angular, dans le `LoginService`, vous devez maintenant appeler le backend pour effectuer la connexion et la déconnexion.

Pour ce faire, utilisez la classe `HttpClient` d'Angular.

Il faut d'abord ajouter le module `HttpClientModule` dans la section `declarations` du `app.module.ts` et ajouter une ligne pour l'importer (`import { HttpClientModule } from "@angular/common/http");`

Faites en sorte qu'Angular injecte l'instance dans le constructeur du `LoginService`.

Inspirez-vous du code ci-dessous pour appeler le backend lors de la connexion. Ajustez au besoin.

```
async login(login: { username: string; password: string }) {
  const loginResponse = await firstValueFrom(
    this.httpClient.post<{ token: string }>(
      `${environment.backendUrl}/auth/login`,
      {
        username: login.username,
        password: login.password,
      }
    )
  );

  localStorage.setItem(LoginService.USERNAME_KEY, login.username);
  localStorage.setItem(LoginService.TOKEN_KEY, loginResponse.token);
  this.token = loginResponse.token;
  this.username.next(login.username);
}
```

Voici quelques explications:

La fonction `firstValueFrom` convertit un `Observable` en `Promise` en ne prenant que la première valeur émise par l'observable, dans ce cas, la réponse HTTP du POST.

Le mot clé `await` attend que la promesse soit complétée avant d'exécuter la suite du code. Pour pouvoir utiliser le mot clé `await`, la fonction doit être asynchrone. On l'indique en ajoutant le mot clé `async` lors de sa définition.

Vous devrez probablement aussi utiliser `async` et `await` dans les méthodes qui utilisent `login` du `LoginService` si vous voulez que l'opération de connexion soit terminée avant d'exécuter la suite du code (par exemple, une redirection vers une autre page).

`environment.backendUrl` fait référence à une variable d'environnement que vous devrez ajouter dans le fichier existant `environment.ts` (n'oubliez pas d'importer ce fichier dans le `LoginService`). Utilisez la valeur `http://127.0.0.1:8080` (adresse du backend en local) pour cette variable d'environnement.

Vous remarquerez que le code ci-dessus stocke, dans le `localStorage`, le `token` retourné par l'appel au login.

Ajoutez une méthode `getToken` qui retourne `token`.

Étape D - Frontend - LoginService - logout

En vous basant sur l'implémentation du login ci-dessus, écrivez le code pour la fonction `logout` qui devra appeler le backend avec un `POST` sur `/auth/logout`.

Étape E - Frontend - AuthInterceptor

Le backend s'attend à recevoir le `token` dans l'en-tête HTTP `Authorization` (voir `AuthFilter`). Pour simplifier le code, nous utiliserons un `HttpInterceptor`, qui est un filtre appliqué à toutes les requêtes envoyées par le `HttpClient` d'Angular.

Créez un nouveau fichier (auth.interceptor.ts) dans le même répertoire que le `LoginService` avec le code suivant:

```
import { Injectable } from '@angular/core';
import {
 HttpInterceptor,
 HttpEvent,
 HttpRequest,
 HttpHandler,
} from '@angular/common/http';
import { Observable } from 'rxjs';
import { LoginService } from './login.service';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
 constructor(private loginService: LoginService) {}
```

```
 intercept(
   httpRequest: HttpRequest<any>,
   next: HttpHandler
 ): Observable<HttpEvent<any>> {
   const token = this.loginService.getToken();

   if (token) {
     httpRequest = httpRequest.clone({
       url: httpRequest.url,
       setHeaders: {
         Authorization: `Bearer ${token}`,
       },
     });
   }
   return next.handle(httpRequest);
 }
}
```

Changez la valeur du champ `providers` dans le fichier `app.module.ts` pour

```
providers: [
   {
     provide: HTTP_INTERCEPTORS,
     useClass: AuthInterceptor,
     multi: true,
   },
 ],
```

Ce filtre intercepte toutes les requêtes HTTP et ajoute le `token` s'il existe.

À ce point-ci, la connexion et la déconnexion devraient être fonctionnelles.

Partie 2 - Le messages

Dans cette partie, vous devrez implémenter les actions suivantes dans le backend:

`GET /messages` pour récupérer les messages
`POST /messages` pour publier un nouveau message

Étape A - Backend - MessageRepository

La classe `MessageRepository` représente l'accès à la couche de données (base de données). Pour le moment, nous stockerons les messages en mémoire avec l'attribut `messages` déjà présent dans cette classe.

Écrivez le code des fonctions `getMessages` et `createMessage` pour retourner la liste des messages et ajouter un nouveau message respectivement.

Vous pouvez ignorer le paramètre optionnel `fromId` de la méthode `getMessages` pour le moment, car nous y reviendrons dans la partie 3.

Étape B - Backend - MessageController

Ajoutez deux nouvelles méthodes au `MessageController`.

La première sera activée par un `GET` sur `/messages` et retournera l'ensemble des messages publiés en utilisant l'instance du `MessageRepository`.

La deuxième sera activée par un `POST` sur `/messages` et utilisera le corps du message HTTP pour créer un nouveau message.

Étape C - Frontend - MessageService

Note: Changez le type du champ id de `string` à `number` dans message.model.ts.

Modifiez le `MessageService` pour qu'il utilise le `HttpClient` pour récupérer et publier les messages en utilisant un `GET` sur `/messages` et un `POST` sur `/messages`.

À la fin de cette étape, vous devriez être capable de récupérer les messages depuis le serveur et de les afficher côté client (application Angular). Vous devriez aussi être capable de publier des nouveaux messages.

Par contre, si un autre usager publie un message, il est possible que votre code ne le récupère pas encore.

Partie 3 - Notifications et websocket

Dans cette partie, vous développerez un mécanisme de notifications de nouveaux messages qui sera utilisé par le backend pour avertir les clients qu'un nouveau message a été publié. Nous utiliserons une connexion websocket.

Étape A - backend - WebSocketManager

Dans le dossier `websocket` du serveur, vous trouverez plusieurs classes responsables de la gestion des connexions websocket.

`WebSocketConfig` active les websockets sur le chemin d'accès `/notifications` et enregistre un objet qui s'occupe de la gestion des connexions: `WebSocketHandler`.

Pour le moment, `WebSocketHandler` ne fait qu'enregistrer la session websocket auprès du `WebSocketManager`. Ce dernier peut ensuite être utilisé pour envoyer une notification sur toutes les websockets connectés en utilisant la méthode `notifySessions()`.

Ajoutez, lorsqu'un nouveau message est publié, l'appel à cette fonction.

Étape B - frontend - MessageService

Voici une classe qui gère une connexion websocket côté client.

```
import { Injectable } from '@angular/core';
import { Observable, Subject } from 'rxjs';
import { environment } from 'src/environments/environment';

@Injectable({
 providedIn: 'root',
})
export class WebsocketService {
 private ws: WebSocket | null = null;

 constructor() {}

 public connect(): Observable<'notif'> {
   this.ws = new WebSocket(`${environment.wsUrl}/notifications`);
```

```
    const events = new Subject<'notif'>();

    this.ws.onmessage = () => events.next('notif');
    this.ws.onclose = () => events.complete();
    this.ws.onerror = () => events.error('error');

    return events.asObservable();
  }

  public disconnect() {
    this.ws?.close();
    this.ws = null;
  }
}
```

Pour utiliser cette classe, vous devez appeler la méthode `connect` pour établir la connexion avec le serveur. Puis `disconnect` pour se déconnecter.

Ajouter un nouvel attribut (`wsUrl`) dans votre fichier `environment.ts` avec l'adresse du serveur de websocket (`ws://127.0.0.1:8080`)

La méthode `connect` fournit un `Observable` qui émet la chaîne de caractères `'notif'` chaque fois qu'un message est reçu du serveur sur la websocket.

Utilisez cet événement pour rafraîchir la liste des messages qui sont affichés sur la page de messagerie.

Étape C - frontend et backend - Optimisation

Ajoutez un paramètre de requête `fromId` qui permet au frontend de spécifier un identificateur de message (id) et de recevoir seulement les messages qui suivent le message qui possède cet identificateur.

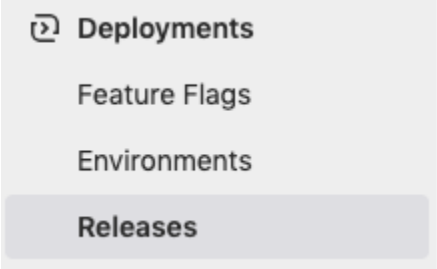
Implémentez la logique dans le backend.

Utilisez ce paramètre pour optimiser vos requêtes lorsque le frontend reçoit une notification sur la websocket.

Remise

La date de remise pour ce TP est le 21 octobre à 23h59.

Avant cette date limite, vous devrez créer une nouvelle distribution (release) avec le nom *tp2* le tag *tp2*.



Pondération

Pour la partie 1, chaque étape complétée donne 1 pt pour un total de 5 pts.
Pour la partie 2, chaque étape complétée donne 2 pt pour un total de 6 pts.
Pour la partie 3, chaque étape complétée donne 2 pts pour un total de 6 pts.

La qualité du code (lisibilité, structure, efficacité) vaut 3 pts pour un total de 20 points.