TP4 - Gestion d’erreur et Tests

Dans ce TP, nous allons améliorer la gestion d’erreur et ajouter des tests unitaires et des tests d’intégration à notre application web.

Base de code

Vous allez utiliser le même dépôt de code que lors de vos TPs précédents.

Objectifs

Pour la partie 1, l’objectif est de gérer correctement certaines erreurs qui peuvent survenir lors de l’utilisation de notre application web.

Dans la partie 2, nous nous concentrons sur l’écriture de tests automatisés.

Pour compléter ce TP, vous devrez suivre les étapes décrites dans les sections suivantes.

Partie 1 - Gestion d’erreur

Étape A - Frontend - Validation du formulaire de connexion

En vous basant sur le code de démo de validation et la documentation Angular assurez-vous que les deux champs (nom d’utilisateur et mot de passe) sont obligatoires pour pouvoir se connecter.

Ajoutez un message d’erreur qui ne s'affiche que si le formulaire est invalide au moment où l'utilisateur appuie sur le bouton pour se connecter.

Voici un exemple:

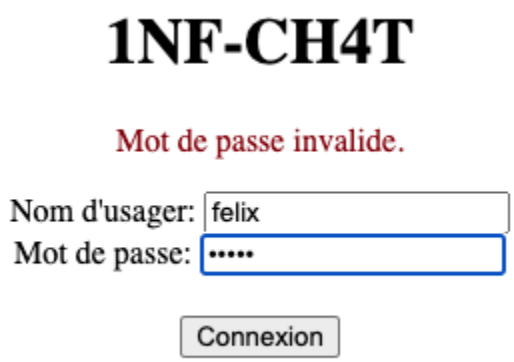
Étape B - Frontend - Problèmes lors de la connexion

Lorsque vous tentez de vous connecter avec un mot de passe invalide, aucun message d'erreur n'est affiché à l'utilisateur.

Dans le **`LoginPageComponent`**, interceptez les exceptions qui pourraient provenir de l'appel à la méthode **`login`** du **`LoginService`**.

Si le code de l'erreur HTTP est 403, affichez le message d'erreur: "Mot de passe invalide".
Pour toutes les autres erreurs affichez: "Problème de connexion".

Indice: Utilisez **`try/catch`** et déterminez le type de l'erreur avec l'aide du mot clé **`instanceof`**. Les erreurs HTTP sont de type **`HttpErrorResponse`**.



Étape C - Frontend - Redirection automatique vers la page du chat

Si l'utilisateur est déjà connecté et qu'il charge l'application, l'application doit le rediriger automatiquement sur la page du chat.

Pour ce faire, nous allons utiliser un *guard* (voir https://angular.io/api/router/CanActivate).

Avec l'outil de ligne de commande Angular, générez un *guard* pour la page de login.

`ng generate guard guards/login-page` - choissisez l'option `CanActivate`

Dans le dossier **`guards`**, vous devriez avoir deux fichiers.

Pour activer le *guard*, vous devez l'ajouter dans la définition de la route qui mène à la page de connexion (login) dans **`app-routing-module.ts`**. Voici un exemple:

```
{
  path: '**',
  component: LoginPageComponent,
  canActivate: [LoginPageGuard],
},
```

Chaque fois que l'application navigue vers cette route, le code de la méthode **`canActivate`** du **`LoginPageGuard`** est exécuté. Si la méthode retourne **`true`**, la navigation sera effectuée normalement. Si elle retourne **`false`**, elle sera bloquée.

Vous pouvez aussi rediriger vers une autre route en utilisant le **`router`** Angular (e.g. **`return this.router.parseUrl('/chat')`**).

Implémentez la méthode **`canActivate`** pour qu'elle permette la navigation sur la page de connexion seulement si l'utilisateur n'est pas déjà connecté (i.e. la méthode **`getToken()`** du **`LoginService`** retourne **`null`**). Sinon, redirigez l'utilisateur vers la page du chat.

Étape D - Frontend - Redirection automatique vers la page de connexion

En créant un autre *guard*, faites en sorte que si l'utilisateur n'est pas connecté, il est automatiquement redirigé vers la page de connexion s'il tente de naviguer vers la page **`/chat`**.

L'utilisateur ne devrait pouvoir naviguer vers la page du chat que s'il est connecté.

Étape E - Frontend - Jeton expiré ou invalide

Lorsque le jeton de l'utilisateur est expiré ou qu'il est invalide, les appels au backend pour récupérer les messages vont échouer avec un HTTP 403.

Dans le **`ChatPageComponent`**, interceptez les erreurs HTTP 403 (sur **`GET`** et **`POST`**). Sur une erreur HTTP 403, exécuter le **`logout`** (**`LoginService#logout`** pour nettoyer l'état) et redirigez automatiquement l'utilisateur vers la page de connexion.

Pour tester ce scénario, vous pouvez modifier temporairement la clé secrète qui est utilisée pour signer les jetons JWT dans votre backend. De cette façon, les anciens jetons seront invalides.

Étape F - Frontend - Déconnexion websocket

Présentement, un utilisateur est sur la page du chat et que votre backend redémarre, l'application Angular perd la connexion websocket. Lorsque le backend est fonctionnel à nouveau, nous voulons que la connexion websocket se rétablisse d'elle-même.

Implémentez un mécanisme de réessaie (retry) pour que l'application Angular tente de rétablir la connexion websocket automatiquement. Insérez un délai de 2 secondes entre chaque essai.

Lorsque la connexion est établie, l'application doit charger les derniers messages qu'elle aurait pu manquer entre-temps (pendant la déconnexion).

Indice: Vous pouvez utiliser **`setTimeout`** pour le délai entre chaque essai. Essayez de centraliser la logique de réessaie dans le **`WebsocketService`**. Vous allez devoir gérer le **`Subject`** différemment et toujours la même instance.

Étape G - Backend - Error Handler

Présentement, lorsqu'il y a une erreur dans le backend, le format de l'erreur est celui par défaut de SpringBoot.

Premièrement, nous allons éliminer le stacktrace de l'erreur en ajoutant la configuration `server.error.include-stacktrace=never` dans **`application.properties`**. Ceci évitera de dévoiler trop d'informations sur notre implémentation.

Ensuite, nous allons nous assurer que nos contrôleurs gèrent les exceptions et les messages d'erreur à retourner aux clients.

Dans **`AuthController`** et **`MessagesController`**, assurez-vous de capturer les exceptions et de lancer à la place une exception de type **`ResponseStatusException`**. Par contre, assurez-vous aussi de laisser passer les exceptions du type **`ResponseStatusException`** qui pourraient survenir.

Par exemple:

```
try {
  …
} catch (ResponseStatusException e) {
  throw e;
} catch (Exception e) {
  throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR,
"Unexpected error on get message.");
}
```

Partie 2 - Tests Frontend

Pour exécuter les tests Angular, il faut utiliser la commande

`ng test`

Important!

Avant de commencer, assurez-vous que vos tests passent.

Si vous avez des erreurs d'injection du type:
`NullInjectorError: No provider for HttpClient!`

Assurez-vous de mettre le bon module dans les imports dans la méthode **`beforeEach`** du test. Par exemple:

```
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientModule],
    });
    guard = TestBed.inject(LoginPageGuard);
  });
```

Autre exemple: **`NullInjectorError: No provider for FormBuilder!`**

```
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [LoginFormComponent],
      imports: [ReactiveFormsModule],
    }).compileComponents();

    fixture = TestBed.createComponent(LoginFormComponent);
    component = fixture.componentInstance;
    testHelper = new TestHelper(fixture);
    fixture.detectChanges();
  });
```

Étape A - Frontend - Tests unitaires - **`LoginFormComponent`**

En vous aidant de la [démo sur les tests unitaires], écrivez les tests unitaires pour le composant qui contient le formulaire de connexion (normalement cela devrait être le **`LoginFormComponent`**).

Voici un exemple (à compléter) pour le cas où on valide que les informations sont émises correctement si le nom d'utilisateur et le mot de passe sont présents dans le formulaire lorsqu'on appuie sur le bouton.

Ajoutez la classe TestHelper, qui utilise l'attribut **`data-testid`** dans le HTML pour faciliter l'accès aux éléments HTML, à votre projet pour vous aider.

Écrivez aussi les tests pour les cas suivants:

- le nom d'utilisateur n'est pas présent
- le mot de passe n'est pas présent
- le nom d'utilisateur et le mot de passe ne sont pas présents

Étape B - Frontend - Tests unitaires - LoginService

Écrivez les tests unitaires pour le LoginService. Basez-vous sur le code du gist suivant: https://gist.github.com/coderunner/ebe192b949b41de35f656675b757533f

Partie 3 - Tests Backend

Avant de commencer, nous allons faire quelques changements au projet.

Ajoutez un fichier **`firebase.properties`** dans le dossier **`ressources`** (avec le fichier **`application.properties)`**.

À l'intérieur, ajouter les deux propriétés suivantes ou *YYYY* est l'id de votre projet **firebase**:

```
firebase.project.id=YYYY
firebase.emulator.port=8181
```

Ensuite, remplacer la classe **`ChatApplication`** pour celle-ci: https://gist.github.com/coderunner/b3c6792cb24bf3a877e8283d28e16150

Cette nouvelle classe définit des accesseurs pour les *beans* **`Firestore`** et **`CloudStorage`**. Vous devriez pouvoir modifier les classes **`UserAccountRepository`** et **`MessageRepository`** pour qu'ils se fassent injecter ces deux dépendances au lieu d'utiliser des méthodes statiques.

Changez aussi la section build du **pom.xml** pour ajouter ce qui suit. Cette configuration nous permettra d'exécuter les tests d'intégration et générera automatiquement un rapport de couverture de code (dans **target/site/jacoco/index.html**).

```
<build>
	<plugins>
		<plugin>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-maven-plugin</artifactId>
		</plugin>
		<plugin>
			<groupId>org.apache.maven.plugins</groupId>
			<artifactId>maven-failsafe-plugin</artifactId>
			<version>3.0.0-M7</version>
			<executions>
				<execution>
					<goals>
						<goal>integration-test</goal>
						<goal>verify</goal>
					</goals>
				</execution>
			</executions>
		</plugin>
		<plugin>
			<groupId>org.jacoco</groupId>
			<artifactId>jacoco-maven-plugin</artifactId>
			<version>0.8.8</version>
			<executions>
				<execution>
					<goals>
						<goal>prepare-agent</goal>
					</goals>
				</execution>
				<execution>
					<id>generate-code-coverage-report-test</id>
					<phase>test</phase>
					<goals>
						<goal>report</goal>
					</goals>
				</execution>
				<execution>
					<id>generate-code-coverage-report-int-test</id>
```

```
                    <phase>integration-test</phase>
                    <goals>
                        <goal>report</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Vérifiez que vous pouvez exécuter les tests avec:
`./mvnw clean verify`

Étape A - Backend - Tests unitaires - AuthController

Pour les tests unitaires du backend, nous allons nous concentrer sur les tests du **`AuthController`**.

La classe de test utilise *mockito* pour mocker les dépendances.

Créez une nouvelle classe dans **`test/java/com/inf5190/chat/auth`** nommée **`TestAuthController`**.

Voici à quoi peut ressembler la classe de test: https://gist.github.com/coderunner/20cb35d2f1761bc0492faad7c945f473. Un premier test est déjà implémenté. Vous devez implémenter les tests pour les autres scénarios du login. Nous ne testerons pas le **`logout`**.

Pour que les validations fonctionnent, vous devez implémenter les méthodes **`equals`**, **`hashcode`** (et **`toString`** pour faciliter le débogage) correctement pour les classes dont vous voulez valider l'égalité (e.g. `FirestoreUserAccount`, `FirestoreMessage`). Notez que pour les **`records`**, ces méthodes sont déjà générées automatiquement.

Ajoutez les tests pour les autres scénarios.

Étape B - Backend - Tests d'intégration - MessageController

Pour les tests d'intégration du backend, nous allons utiliser l'émulateur de **firestore**.

Il faut d'abord initialiser l'environnement **`firebase`**.

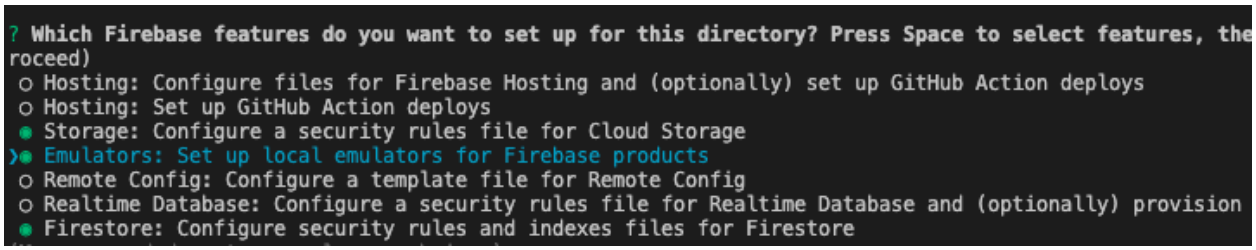
Installez l'outil de ligne de commande de firebase: https://firebase.google.com/docs/cli.

Ensuite, dans le dossier **`backend`**, exécutez:

```
firebase init
```

Vous allez devoir vous connecter avec votre compte google. Utilisez le même que celui avec lequel vous avez créé votre projet firebase au TP3.

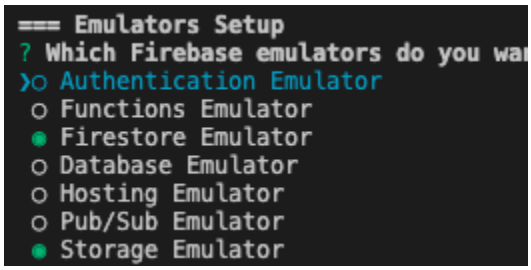
Puis choisissez les options: Firestore, Cloud Storage et Emulators



Utilisez votre projet existant.

Choisissez les options par défaut pour les différents fichiers proposés.

Sélectionnez les émulateurs de **`firestore`** et **`storage`**.



Un fois ces étapes complétées, dans le fichier **`firebase.json`**, modifiez le contenu de l'attribut **`emulators`** pour utiliser le port 8181 pour l'émulateur de **firestore**.

```
"emulators": {
   …
   "firestore": {
     "port": "8181"
   }
   …
 }
```

Ceci permettra à l'émulateur d'utiliser le port 8181 au lieu du port 8080 (déjà utilisé par SpringBoot).

Vous pouvez lancer l'émulateur avec la commande:

`firebase emulators:start`

Vous pouvez voir la console de l'émulateur en vous connectant sur http://127.0.0.1:4000/ avec votre navigateur.

Nous allons utiliser l'émulateur de **firestore** pour nos tests d'intégration.

Nous allons maintenant lancer nos tests avec la commande suivante:

`firebase emulators:exec "./mvnw clean verify"`

L'émulateur de firestore va démarrer les émulateurs, configurer correctement les variables d'environnement, exécuter la commande maven spécifiée et éteindre les émulateurs.

Il est possible de lancer les émulateurs avec la commande **`firebase emulators:start`** et de spécifier les deux variables d'environnement suivantes avant de déboguer les tests avec votre éditeur ou de lancer **`./mvnw clean verify`** par exemple.

```
FIRESTORE_EMULATOR_HOST=localhost:8181
FIREBASE_STORAGE_EMULATOR_HOST= localhost:9199
```

Note: Il semble y avoir un bogue avec la connexion vers l'émulateur de Storage depuis la bibliothèque admin java. La connexion ne se fait pas correctement et le serveur se connecte sur le vrai projet firebase au lieu de l'émulateur. Donc nous ne testerons pas les images dans les messages pour le moment.

Nous sommes prêts pour les tests d'intégration!

Créez une nouvelle classe dans **`test/java/com/inf5190/chat/messages`** nommée **`ITestMessageController.java`**.

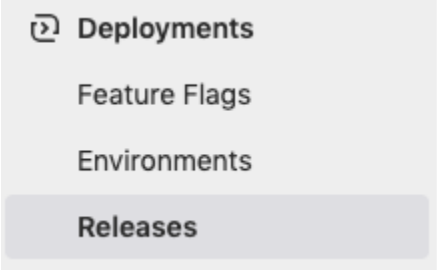
Utilisez le gist suivant comme base:
https://gist.github.com/coderunner/43f013dff62294d345706a442952c9bf

Ajoutez des tests pour **`GET /messages`** et **`POST /messages`** sans jeton valide.
Ajoutez des tests pour **`GET /messages`** avec et sans paramètre **`fromId`**.
Ajoutez un test pour le cas où il y a plus de 20 messages dans **firestore**.
Ajoutez un test avec un **`fromId`** invalide et valider que le code d'erreur HTTP est 404.
Implémentez ce scénario si ce n'est pas déjà fait :)

Remise

La date de remise pour ce TP est le 2 décembre à 23h59.

Avant cette date limite, vous devrez créer une nouvelle distribution (release) avec le nom *tp4* le tag *tp4*.



Pondération

Pour la partie 1, chaque étape complétée donne 1 pt pour un total de 7 pts.
Pour la partie 2, chaque étape complétée donne 2 pts pour un total de 4 pts.
Pour la partie 3, chaque étape complétée donne 2 pts pour un total de 4 pts.

La qualité du code (lisibilité, structure, efficacité) vaut 5 pts pour un total de 20 points.