TP3 - Firebase

Vous utiliserez Firebase, en particulier Firestore et Cloud Storage, pour stocker les comptes utilisateurs (Firestore), les messages (Firestore) et des images (Cloud Storage).

Base de code

Vous allez utiliser le même dépôt de code que lors du TP2.

Compte Firebase

Avant de commencer, vous devrez créer un projet **Firebase**. **Firebase** est un ensemble de solutions infonuagiques offert par Google.

Je vous invite à explorer les nombreuses options de **Firebase**, mais nous nous concentrerons seulement sur **Firestore** (base de données orientée documents) et **Cloud Storage** (solution de stockage de fichiers) pour ce TP.

Pour la création de votre projet, allez sur https://firebase.google.com/ et connectez-vous avec votre compte google. Si vous n'en possédez pas, créez-en un.

Ensuite, cliquez sur le bouton Go to console en haut à droite.

Cliquez sur le bouton Ajouter un projet.

Donnez-lui un nom (e.g. inf5190-chat) et appuyez sur Continuer.

Nous n'aurons pas besoin de Google Analytics, vous pouvez donc le désactiver.

Cliquez sur Créer un projet et patientez :)

Vous devriez maintenant voir votre projet.

Dans la barre de navigation à gauche, dans la section Créer, cliquez sur **Firestore Database.**

Cliquez sur Créer une base de données.

Choisissez Démarrer en mode production puis cliquez sur Suivant.

Choisissez la zone **northamerica-northeast1** (c'est à Montréal et à faible émission de CO2!)

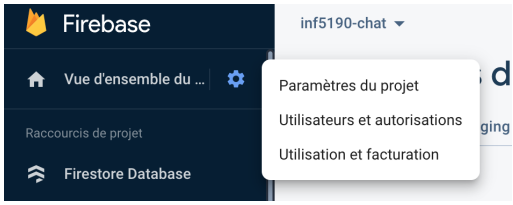
Appuyez sur Activer.

Vous avez maintenant une base de données **Firestore**.

Dans la barre de navigation à gauche, dans la section créer, cliquez sur Storage. Ce sera l'endroit ou l'application stockera des images lors de la partie 3.

Vérifier en bas à gauche que le projet utilise la formule Spark (sans frais).

Avant de commencer à interagir avec la base de données à partir du backend, téléchargez la clé privée.



Pour ce faire, allez dans la section paramètres (l'engrenage) puis cliquez sur Utilisateurs et autorisations.

Dans la section Comptes de service, cliquez sur Générez une nouvelle clé privée, puis Générez la clé. Sauvegardez ce fichier à la racine de votre répertoire `backend` et renommez-le firebase-key.json.

C'est parti!

Objectifs

Dans ce TP nous ajouterons plusieurs fonctionnalités.

- La sauvegarde des comptes utilisateurs dans **Firestore** (nom d'utilisateur et mot de passe encodé avec l'algorithme **BCrypt**).
- L'utilisation du **JWT** pour la gestion de la session.
- Le stockage des messages dans **Firestore**.
- L'option d'envoyer une image dans le *chat*.
- Le stockage des images dans **Cloud Storage**.

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

Partie 0 - Nouvelles bibliothèques et initialisation de Firebase

Dans le fichier `pom.xml`, ajoutez les dépendances suivantes dans la section `<dependencies>`.

```
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-crypto</artifactId>
```

```
            <version>5.7.4</version>
        </dependency>

        <dependency>
            <groupId>com.google.firebase</groupId>
            <artifactId>firebase-admin</artifactId>
            <version>9.1.0</version>
        </dependency>

        <dependency>
            <groupId>io.jsonwebtoken</groupId>
            <artifactId>jjwt-api</artifactId>
            <version>0.11.5</version>
        </dependency>
        <dependency>
            <groupId>io.jsonwebtoken</groupId>
            <artifactId>jjwt-impl</artifactId>
            <version>0.11.5</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>io.jsonwebtoken</groupId>
            <artifactId>jjwt-jackson</artifactId>
            <version>0.11.5</version>
            <scope>runtime</scope>
        </dependency>
```

Dans le fichier `ChatApplication.java`, ajoutez un `LOGGER` et remplacez la fonction `main` en utilisant le code suivant:

```
private static final Logger LOGGER = LoggerFactory.getLogger(ChatApplication.class);

public static void main(String[] args) {
        try {
            if (FirebaseApp.getApps().size() == 0) {
                FileInputStream serviceAccount = new
FileInputStream("firebase-key.json");

                FirebaseOptions options = FirebaseOptions.builder()
                        .setCredentials(GoogleCredentials.fromStream(serviceAccount))
                        .build();
```

```
                LOGGER.info("Initializing Firebase application.");
                FirebaseApp.initializeApp(options);
            }
            LOGGER.info("Firebase application already initialized.");

            SpringApplication.run(ChatApplication.class, args);
        } catch (IOException e) {
            System.err.println("Could not initialise application. Please check you
service account key path");
        }
    }
```

Ajuster les `imports` en conséquence. Pour les classes `Logger` et `LoggerFactory`, utilisez les imports de `slf4j`.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

Assurez-vous que votre serveur démarre bien. N'oubliez pas votre clé privée **Firebase** qui devrait se trouver à la racine du répertoire `backend` et se nommer **firebase-key.json**.

L'API de Firestore

Pour vous familiariser avec l'API de Firestore, lisez et expérimentez avec le code du `FirestoreDemoController` disponible dans le gist suivant: https://gist.github.com/coderunner/1e74720501b66c2068344c39aafc3e27. Vous pouvez simplement ajouter temporairement le contrôleur dans votre projet.

Vous pouvez aussi consulter la documentation pour plus de détails: https://firebase.google.com/docs/reference/android/com/google/firebase/firestore/package-summary.

Partie 1 - Comptes utilisateurs

Étape A - Backend - Sauvegarde des comptes utilisateurs

Créez un nouveau dossier `repository` dans le dossier `auth`. Ajoutez-y la classe suivante:

```
package com.inf5190.chat.auth.repository;

public class FirestoreUserAccount {
   private String username;
   private String encodedPassword;

   public FirestoreUserAccount() {
   }

   public FirestoreUserAccount(String username, String encodedPassword) {
       this.username = username;
       this.encodedPassword = encodedPassword;
   }

   public String getUsername() {
       return username;
   }

   public void setUsername(String username) {
       this.username = username;
   }

   public String getEncodedPassword() {
       return encodedPassword;
   }

   public void setEncodedPassword(String encodedPassword) {
       this.encodedPassword = encodedPassword;
   }
}
```

Cette classe représente un compte utilisateur stocké dans Firestore.

Dans le même dossier, ajoutez la classe suivante:

```
package com.inf5190.chat.auth.repository;

import java.util.concurrent.ExecutionException;

import org.springframework.stereotype.Repository;

import com.google.cloud.firestore.Firestore;
import com.google.firebase.cloud.FirestoreClient;

@Repository
public class UserAccountRepository {
   private static final String COLLECTION_NAME = "userAccounts";

   private final Firestore firestore = FirestoreClient.getFirestore();

   public FirestoreUserAccount getUserAccount(String username) throws
InterruptedException, ExecutionException {
       throw new UnsupportedOperationException("A faire");
   }

   public void setUserAccount(FirestoreUserAccount userAccount) throws
InterruptedException, ExecutionException {
       throw new UnsupportedOperationException("A faire");
   }
}
```

`UserAccountRepository` sera responsable d'écrire et de lire les comptes utilisateurs depuis **Firestore**.

Remarquez bien les imports pour FirestoreClient et Firestore. Il y a plusieurs versions différentes dans le classpath. Utilisez les imports du code ci-dessus partout dans votre backend.

Pour simplifier, nous allons utiliser le nom d'utilisateur comme `id` dans **Firestore**. Ceci nous assurera que nous n'aurons pas deux comptes pour le même utilisateur.

Compléter l'implémentation de la méthode `setUserAccount` sachant qu'elle doit écrire le `userAccount` reçu en paramètre dans la collection **Firestore** `userAccounts`. Vous n'avez pas à créer la collection, elle sera créée lorsqu'un premier document y sera écrit.

Étape B - Backend - Lecture des comptes utilisateurs

Dans la classe `UserAccountRepository`, complétez l'implémentation de la méthode `getUserAccount` sachant qu'elle doit effectuer les opérations suivantes:

1. Lire depuis **Firestore** pour récupérer le compte utilisateur depuis la collection `userAccounts`, l'identifiant du document est le nom d'utilisateur.
2. Si le compte n'existe pas, la fonction retourne `null`.
3. Si le compte existe, la fonction retourne un objet du type `FirestoreUserAccount`.

Étape C - Backend - AuthController - Création des comptes

Nous allons maintenant modifier le `AuthController` pour qu'il utilise le `UserAccountRepository` pour lire et stocker les comptes utilisateurs lors de la connexion. Nous utiliserons un mécanisme d'auto-inscription: nous allons créer le compte lorsque l'utilisateur se connecte pour la première fois.

Dans la méthode login, changez le code pour effectuer les opérations suivantes:

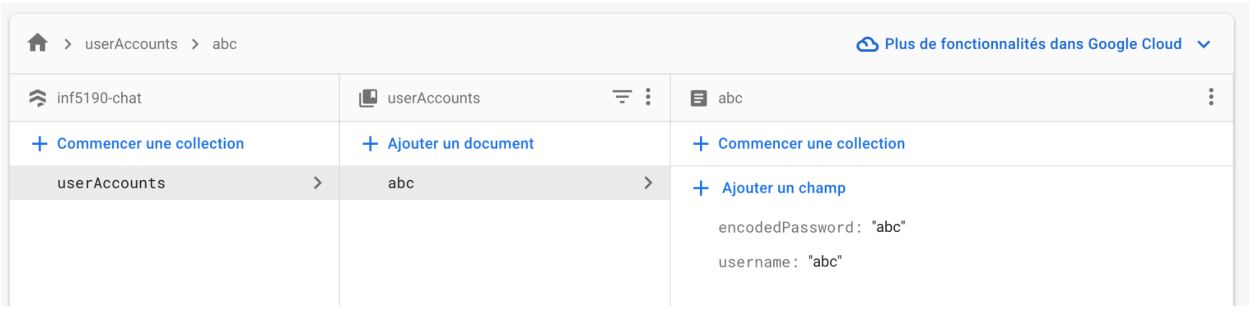
- Vérifiez si un compte existe déjà pour cet utilisateur
    - Si non, créez un compte utilisateur FirebaseUserAccount avec le nom d'utilisateur et le mot de passe et ajoutez le dans Firestore.
- Créez la session et retournez le jeton (comme pour le TP2).

Note sur la gestion des exceptions: vous pouvez seulement ajouter la clause `throws` dans la déclaration de la fonction du contrôleur pour le moment. Nous y reviendrons dans le TP4.

Par exemple la signature de `login` du `AuthController` devient

```
public LoginResponse login(@RequestBody LoginRequest loginRequest) throws
InterruptedException, ExecutionException {
```

Tester maintenant avec le frontend. Sur un login vous devriez voir le compte utilisateur dans Firestore.



Étape D - Backend - Gestion du mot de passe

Nous allons maintenant gérer le mot de passe de façon plus sécuritaire.

Dans `ChatApplication.java`, ajouter la définition suivante:

```
@Bean
public PasswordEncoder getPasswordEncoder() {
    return new BCryptPasswordEncoder();
}
```

Dans `AuthController`, vous allez utiliser cet `Encoder` pour encoder le mot de passe avant de le stocker dans **Firestore**. Vous l'utiliserez aussi pour valider si le mot de passe reçu lors des connexions subséquentes correspond au mot de passe du compte. Ajoutez un paramètre du type `PasswordEncoder` dans le constructeur du `AuthController`.

Dans la fonction `login` du `AuthController`, avant de le stocker dans le `FirestoreUserAccount`, encodez le mot de passe (voir la documentation).

Ensuite, si jamais le compte utilisateur existait déjà, validez que le mot de passe correspond bien à celui stocké dans **Firestore**. Si ça ne correspond pas, lancez l'exception `throw new ResponseStatusException(HttpStatus.FORBIDDEN)`. Ceci retournera une erreur 403 que nous allons gérer correctement du côté frontend dans le TP4.

Supprimez les comptes utilisateurs créés à l'étape précédente et qui se trouvent dans **Firestore** (via la console web), puis créez des nouveaux comptes en vous connectant dans votre application. Vérifiez que, si la deuxième fois que vous utilisez un nom d'utilisateur, la connexion est rejetée si vous n'avez pas spécifié le bon mot de passe.

Étape E - Backend - Création du JWT

Nous allons maintenant simplifier la gestion de sessions. Nous allons utiliser un **JWT** au lieu de stocker la session en mémoire (ou dans une base de données).

Pour ce faire, nous allons modifier le `SessionManager`.

Nous utiliserons la bibliothèque Java JWT.

Modifiez le SessionManager en ajoutant:

```
private static final String SECRET_KEY_BASE64 = "<VOTRE CLÉ SECRÈTE>";
private final SecretKey secretKey;
private final JwtParser jwtParser;

public SessionManager() {
    this.secretKey = Keys.hmacShaKeyFor(Decoders.BASE64.decode(SECRET_KEY_BASE64));
    this.jwtParser = Jwts.parserBuilder().setSigningKey(this.secretKey).build();
}
```

Créez votre propre clé secrète. Voir https://github.com/jwtk/jjwt#secret-keys puis ajoutez-la dans votre code (`SECRET_KEY_BASE64`).

Dans la méthode `addSession`, effacez le code existant et utilisez `Jwts.builder()` pour créer le **JWT**. Dans le jeton, mettez l'audience, la date de création (issuedAt), le sujet (le nom d'utilisateur), l'expiration (2 heures après la création du jeton) et signez-le avec votre clé secrète. Retournez ensuite ce jeton en version compacte. Voir les exemples dans la documentation.

Changez le code de la méthode `getSession` pour qu'il utilise le parser **JWT** pour lire le nom d'utilisateur à partir du jeton et ainsi créer le `SessionData`. Utilisez la méthode `parseClaimsJws` (et non `parseClaimsJwt`). Voir la documentation.

Interceptez les exceptions lors de l'opération de décodage (`catch (JwtException e)`). Sur un échec, retournez simplement `null` pour indiquer que la session n'existe pas (le JWT n'est pas valide).

Supprimez le code du TP précédent qui n'est pas utilisé.

Votre application utilise maintenant un **JWT** pour gérer la session. Vous n'avez plus besoin de stocker la session en mémoire et lorsque votre backend redémarrera, la session sera encore valide! Par contre, notez que le code ne gère pas la déconnexion du websocket. Nous y reviendrons au TP4.

Partie 2 - Gestion des messages

Dans cette partie, nous utiliserons **Firestore** pour stocker les messages de façon permanente.

Étape A - Écriture des messages

Modifiez le code du `MessageRepository` pour qu'il utilise **Firestore** pour écrire les messages dans une collection nommée `messages`.

D'abord, dans le dossier `repository` du package `messages`, créez une nouvelle classe qui sera utilisée pour représenter les messages que nous stockerons dans **Firestore**. Voici le code:

```
package com.inf5190.chat.messages.repository;

import com.google.cloud.Timestamp;

public class FirestoreMessage {
   private String username;
   private Timestamp timestamp;
   private String text;

   public FirestoreMessage() {
   }

   public FirestoreMessage(String username, Timestamp timestamp, String text) {
       this.username = username;
       this.timestamp = timestamp;
       this.text = text;
   }

   public String getUsername() {
       return username;
   }

   public void setUsername(String username) {
       this.username = username;
   }

   public Timestamp getTimestamp() {
       return timestamp;
   }
```

```
    public void setTimestamp(Timestamp timestamp) {
        this.timestamp = timestamp;
    }

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }
}
```

Ensuite, modifiez la méthode `createMessage` du `MessageRepository` pour ajouter les nouveaux messages dans **Firestore**.

Le backend est responsable de la génération de l'id (on veut qu'il soit unique globalement) et de la génération du timestamp (on ne veut pas dépendre de l'horloge des utilisateurs). Donc si vous recevez ces valeurs du frontend, vous devez les ignorer.

Aussi, n'utilisez plus le `AtomicLong` pour générer l'id, nous prendrons celui généré par **Firestore**.

Le `message` retourné au frontend devra contenir le bon id et le bon timestamp.

Puisque Firestore utilise un `id` de type `string`, nous devons modifier de nouveau le type dans la classe `Message` (backend: `Message.java` et frontend: `message.model.ts`).

Étape B - Lectures des messages

Réimplémenter la méthode `public List<Message> getMessages(Optional<String> fromId)` en utilisant l'API de **Firestore**.

Notez qu'il faut modifier le type de `fromId` pour que ce soit une `String` (dans le `MessageRepository` et dans le `MessageController`).

Pour éviter de charger trop de messages si le paramètre `fromId` n'est pas spécifié, limitez la requête aux 20 derniers messages.

Pour l'implémentation du `fromId`, vous utilisez la méthode startAfter.

Votre application devrait être fonctionnelle et vos messages sont maintenant stockés dans **Firestore**.

Partie 3 - Envoi et stockage d'images

Dans cette partie, nous allons permettre aux utilisateurs d'envoyer une image avec le texte dans le *chat*.

Étape A - Frontend - Préparation du modèle

Comme première étape, nous allons ajuster le modèle des objets qui transitent entre le frontend et le backend.

Dans le fichier `message.model.ts`, modifiez le contenu pour définir les 3 interfaces suivantes:

```
export interface Message {
 id: string;
 username: string;
 text: string;
 timestamp: number;
 imageUrl: string | null;
}

export interface MessageRequest {
 username: string;
 text: string;
 imageData: ChatImageData | null;
}

export interface ChatImageData {
 data: string;
 type: string;
}
```

L'interface `Message` représente un message tel que reçu de l'API backend sur un `GET /messages`. Notez l'ajout d'un champ `imageUrl` qui permet au frontend d'afficher une image présente dans le message.

L'interface `MessageRequest` représente un nouveau message publié que le frontend envoie dans le corps de la requête au backend lors du `POST /messages`. Ceci formalise le fait que le frontend ne devrait envoyer ni d'`id`, ni de `timestamp`.

L'interface `ChatImageData` représente les données de l'image. Le champ `data` contient le contenu de l'image encodé en base64 et le champ `type`, contient l'extension originale du fichier.

Dans le `MessageService`, modifiez la signature de la méthode `postMessage` pour qu'elle prenne un `MessageRequest` au lieu d'un `Message`. Ajustez les appels à cette méthode pour qu'elle n'envoie ni d'`id`, ni de `timestamp`. Assignez la valeur `null` au champ `imageData` pour le moment.

Étape B - Backend - Ajustement du modèle

Dans le backend, dans le dossier `model` du dossier `messages`, ajuster la définition de `Message` et ajouter les définitions pour `MessageRequest` et `ChatImageData` pour qu'elles correspondent avec celles du frontend.

N'oubliez pas de modifier aussi `FirestoreMessage` pour qu'il contienne un champ `imageUrl`. Ajoutez le nouveau paramètre dans le constructeur et ajoutez une paire de *getter* et *setter* pour ce nouveau champ.

Ajustez le `MessageController` pour que la méthode qui réagit au `POST /messages` prenne un `MessageRequest` et non un `Message`.

Modifiez le `MessageRepository` pour que la méthode `createMessage` prenne un `MessageRequest` en paramètre et qu'elle assigne une valeur de `null` au champ `imageUrl` (temporairement) lors de la création d'un nouveau message.

À ce moment-ci, tout devrait fonctionner comme avant. Nous avons seulement modifié le modèle de données associé aux messages sans modifier le comportement de l'application.

Étape C - Frontend - Ajout d'images

Lors de cette étape, nous allons permettre à l'utilisateur d'ajouter une image à son message. Dans le composant qui affiche le formulaire et qui permet à l'utilisateur de publier un nouveau message, ajoutez un `input` de type `file` pour afficher le dialogue de sélection de fichiers.

Voici un exemple. L'input ci-dessous est invisible (`display: none`), mais possède un nom: `file` (syntaxe `#file`). L'élément `span` réagit au clique et fait suivre le clique au `input` (`file.click()`) qui ouvre le dialogue de sélection de fichier. Remarquez que lorsque la valeur de l'input change, la méthode `fileChanged` est appelée.

```
<input
  #file
  name="file"
```

```
  type="file"
  style="display: none"
  accept="image/*"
  (change)="fileChanged($event)"
/>
<span (click)="file.click()"
  ><img src="/assets/image-icon.svg" height="25px"
/></span>
```

Voici un exemple possible du code de la méthode `fileChanged` où, lors du changement, on conserve la référence au fichier sélectionné.

```
 file: File | null = null;
 [...]
 fileChanged(event: any) {
  this.file = event.target.files[0];
}
```

Modifiez maintenant le code pour que, lorsque l'utilisateur publie le message, si un fichier a été sélectionné, un objet de type `ChatImageData` est créé et ajouté au `MessageRequest`.

Utilisez le service suivant pour lire le fichier, convertir le contenu en base64 et obtenir le `ChatImageData` à partir du fichier.

```
import { Injectable } from '@angular/core';
import { ChatImageData } from './message.model';

@Injectable({
 providedIn: 'root',
})
export class FileReaderService {
 constructor() {}

 async readFile(file: File): Promise<ChatImageData> {
  const reader = new FileReader();

  const fileRead = new Promise<ArrayBuffer>((resolve, reject) => {
    reader.onload = (b) => resolve(reader.result as ArrayBuffer);
    reader.onerror = (e) => reject('could not read file');
  });

  reader.readAsArrayBuffer(file);
```

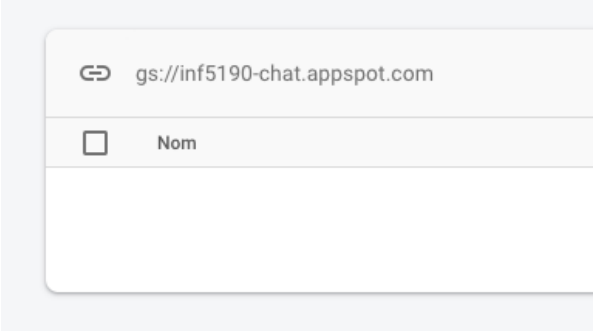
```
    const type = file.name.split('.').pop() || '';
    const b = await fileRead;
    return { data: this.arrayBufferToBase64(b), type: type };
  }

  private arrayBufferToBase64(buffer: ArrayBuffer): string {
    const bytes = new Uint8Array(buffer);
    let binary = '';
    for (var i = 0; i < bytes.byteLength; i++) {
      binary += String.fromCharCode(bytes[i]);
    }
    return window.btoa(binary);
  }
}
```

Étape D - Backend - Stockage de l'image et création du URL

À ce point-ci, le frontend devrait envoyer le champ `imageData` dans le `MessageRequest`. Il faut donc modifier le backend pour stocker l'image dans **Cloud Storage** et générer un URL qui sera ajouté dans le message stocké dans **Firestore**.

Déterminez d'abord le nom de votre *bucket* pour **Cloud Storage**. Dans la console **Firebase** (https://console.firebase.google.com/), sélectionnez votre projet. Puis sélectionnez Storage. Vous devriez voir le nom de votre *bucket*. Par exemple, dans le cas ci-dessous, le nom du *bucket* est `inf5190-chat.appspot.com`.



Lorsque vous stockez l'image, utilisez l'id du message comme nom de fichier.

Utilisez le code suivant pour vous guider:

```
Bucket b = StorageClient.getInstance().bucket(BUCKET_NAME);
String path = String.format("images/%s.%s", ref.getId(),
messageRequest.imageData().type());
b.create(path, Decoders.BASE64.decode(messageRequest.imageData().data()),
      BlobTargetOption.predefinedAcl(PredefinedAcl.PUBLIC_READ));
imageUrl = String.format("https://storage.googleapis.com/%s/%s", BUCKET_NAME, path);
```

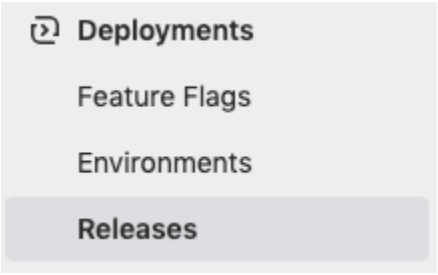
Étape E - Frontend - Affichage de l'image

Il suffit maintenant d'ajouter l'affichage de l'image (avec une une balise `img`) dans le composant Angular responsable de l'affichage des messages (si le champ `imageUrl` est présent).

Remise

La date de remise pour ce TP est le 11 novembre à 23h59.

Avant cette date limite, vous devrez créer une nouvelle distribution (release) avec le nom *tp3* le tag *tp3*.



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

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