ChowChow

Your daily show manager

Prevost Domitille - Hénon Gauthier - Criaud Donatien

Avant-propos

Nous présentons dans ce document notre travail sur le projet final du cours de POOA. Nous avons choisi comme sujet “*Mes séries préférées”,* qui consiste à coder une application web (avec interface graphique) permettant à un utilisateur de se constituer une liste de séries favorites, de recueillir des informations sur celles-ci et d’être informé des nouveaux épisodes lorsque leur date de diffusion est proche.

Table des matières

Introduction

Afin de réaliser ce projet, nous avons utilisé trois environnements de développement:

- un environnement de développement local: utilisation de Docker pour lancer et orchestrer les différents containers.
- un environnement de pré-production, hébergé sur Heroku (https://chowchow-staging.herokuapp.com et https://chowchow-ui-staging.herokuapp.com)
- un environnement de production, hébergé sur Heroku (https://chowchow-prod.herokuapp.com et https://chowchow-ui-prod.herokuapp.com)

Comme nous avons utilisé l'offre gratuite de Heroku, les serveurs mis à disposition prennent du temps à se lancer. Pour tester ces environnements, nous vous conseillons de tenter de charger chaque url une fois "à blanc" avant d'en faire une utilisation normale afin d'éviter des délais de réponse client-serveur, notamment.

Toutes les informations pour mettre en place l'environnement de développement sont disponibles dans le fichier `README.md` à la racine du code source du projet. Pour résumer en quelques lignes, une fois Docker installé, il suffit de :

1. Créer un fichier `.env` à la racine du code source en suivant le modèle du fichier `.env.default`;
2. Faire de même à la racine de l'application React;
3. Packager l'application en utilisant Maven;
4. Utiliser docker-compose pour build & run les containers de l'API Java, la base de données et l'app client;

5. L'application est disponible sur http://localhost:3000 et l'API sur http://localhost:8080

La seule variable d'environnement qu'il vous faudra modifier devrait normalement être la clé API TMDB du fichier .env que nous ne pouvons pas publier pour des raisons de sécurité. Cependant vous devriez en avoir reçu une valide via papier.

Choix techniques

Spring

Spring étant un framework très largement utilisé pour le développement d'API web en Java, nous avons choisi de l'utiliser afin de disposer d'un grand nombre de ressources et de documentation. En outre Spring Boot permet de mettre en place rapidement ce genre d'application.

React

Une interface graphique étant exigée pour ce projet, nous avons fait le choix d'opter pour une UI web légère en utilisant ReactJS. Il permet la réalisation d'interfaces rapidement et nous avons privilégié la simplicité pour nous concentrer sur la POO en Java.

L'app React est située dans `src/main/javascript` et peut tourner localement dans un conteneur Docker. Elle dépend d'une variable d'environnement `*REACT_APP_API_URL*` qui pointe vers l'url de l'API Spring.

PostgreSQL

La figure ci-dessous présente le schéma relationnel de la base PostgreSQL utilisée pour persiter les données de l'application. Chaque migration du schéma de base de données est décrite dans un fichier SQL et déployée automatiquement par l'application Spring grâce à Flyway.

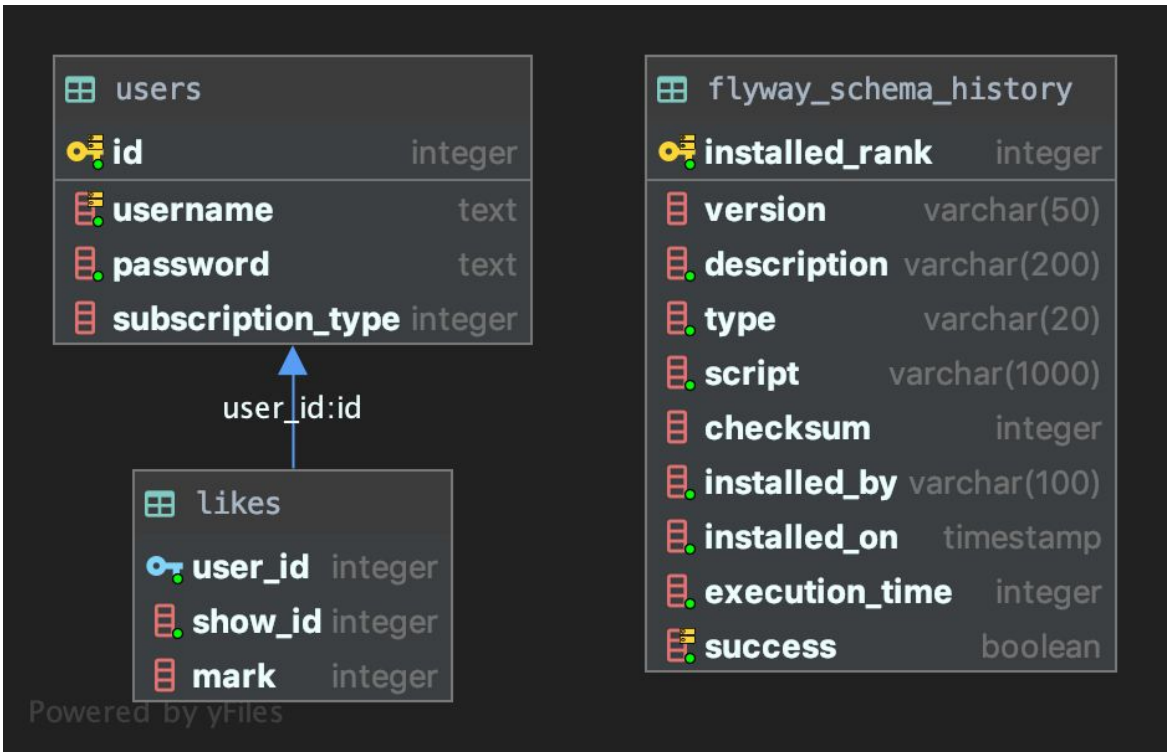

Diagramme UML de notre base de données `chowchow`

Nous nous basons quasi exclusivement sur l'API TMDB pour récupérer les données des séries favorites de l'utilisateur. Cela nous permet de ne stocker localement que les données utilisateur dans une table **users** et la note choisie par l'utilisateur dans une table **likes** comme le montre le schéma UML ci-dessus.

La table **flyway_schema_history** est générée automatiquement par Flyway, un gestionnaire de migrations, pour garder trace du numéro de la dernière migration appliquée.

Réponses aux attendus académiques

Installation

L'installation de l'application se fait en suivant pas à pas les étapes détaillées dans le fichier *README.md* à la racine du projet.

Recette

L'application ChowChow permet bien à un utilisateur d'ajouter une série à sa liste de séries. Ici la liste des séries est une liste des série "à regarder" : Une fois dans cette liste, l'utilisateur peut "liker" la série selon trois niveaux (représentés par des os). Ainsi, l'utilisateur sera alerté de la diffusion des séries "likées" de cette façon. En outre, un utilisateur peut consulter le résumé détaillé des séries.

Clarté du code

Nous avons accordé une grande importance au nommage des champs, classes, méthodes etc… Ainsi nous avons fait en sorte de produire un code lisible dont le nom des variables permette de comprendre le fonctionnement de l'application. Néanmoins, nous avons apporté des précisions sous forme de commentaires à certains endroits.

Afin de permettre une prise en main rapide au lecteur, nous avons mis en place une API Swagger permettant de décrire les différents points d'entrée de l'application. De plus, il est possible de générer la javadoc du projet en utilisant Maven. Les instructions permettant de le faire se trouvent dans le fichier *README.md*.

Respect de la POO

Emploi du DTO pattern

Nous utilisons des objets appelés DTO afin de transmettre des données entre les différents *controllers* de l'application, notre application et l'API TMDB et retourner les réponses aux requêtes effectuées sur notre API. Ces objets sont définis de manière à rendre tous les champs privés. Nous avons défini uniquement des getters afin que ces objets soient read-only. De plus, ces objets nous permettent une sérialisation/désérialisation simple des données.

Architecture de l'application

Afin d'encapsuler au mieux toutes les fonctionnalité de notre application, nous avons décidé d'une architecture sous la forme *WebControllers - Controllers - Services*:

- *WebControllers* : Définit les **points d'entrée de l'API REST**, les routes, le format des requêtes et des réponses. Il ne contient aucune logique, celle-ci est déléguée aux *Controllers*.
- *Controllers* : Encapsulent toute la **logique de l'application**, ce sont les seuls qui manipulent et peuvent modifier les objets persistés en base. Les seuls objets qu'ils retournent sont des DTOs, ceci permet de ne faire sortir des controllers que des objets read-only. Les controllers récupèrent les informations qui leur sont utiles grâce aux services.
- *Services* : Les services sont des interfaces qui contractent des méthodes permettant d'**accéder à de l'information**. Ces méthodes contiennent très peu de logique. Par exemple *UserService* possède une méthode *findById* permettant de retrouver dans la base de donnée un utilisateur, tandis que *SearchService* encapsule les call API effectués à TMDB.
- *ServiceImpls* : Il s'agit des classes implémentant les *Services*.
- *Repositories* : Afin de **définir les transaction avec la base PostgreSQL** au travers de JPA, nous avons utilisé des *repositories*.

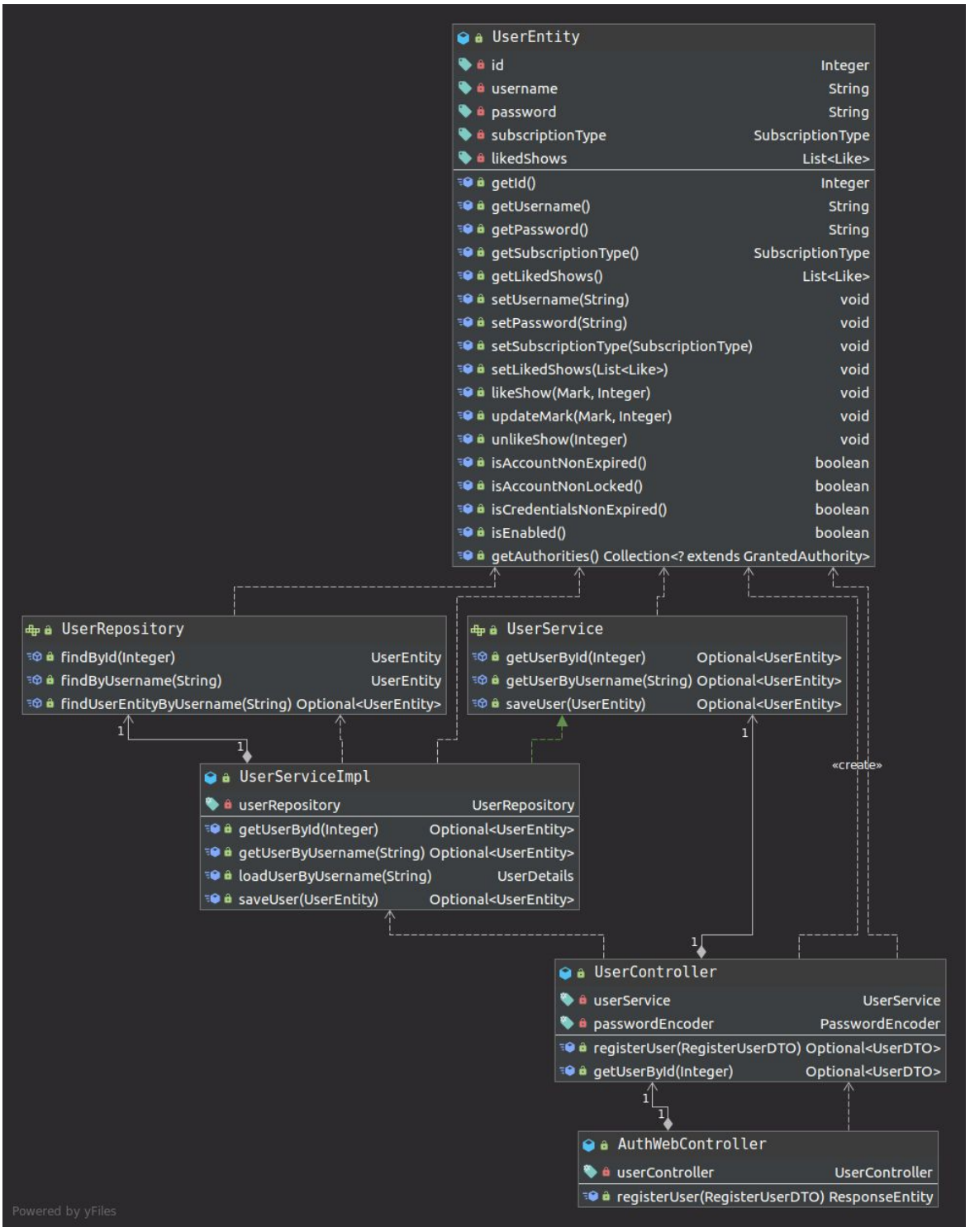

Diagramme UML simplifié du package “user”

UserAlertManager

Nous avons choisi de développer un module de gestion des alertes utilisateur. Une alerte est un objet permettant d’indiquer le prochain épisode d’un show, identifié

par son id. Dans notre application l'utilisateur peut ajouter un show à sa "watch list". Dans le cas où il a vu le show, il peut le "liker" en le notant. Nous récupérons les alertes grâce à l'API TMDB mais nous voulons ne renvoyer à l'utilisateur que celles concernant les shows "likés". Pour ce faire nous avons utilisé les 4 piliers de l'orienté objet: abstraction, encapsulation, polymorphisme et héritage.

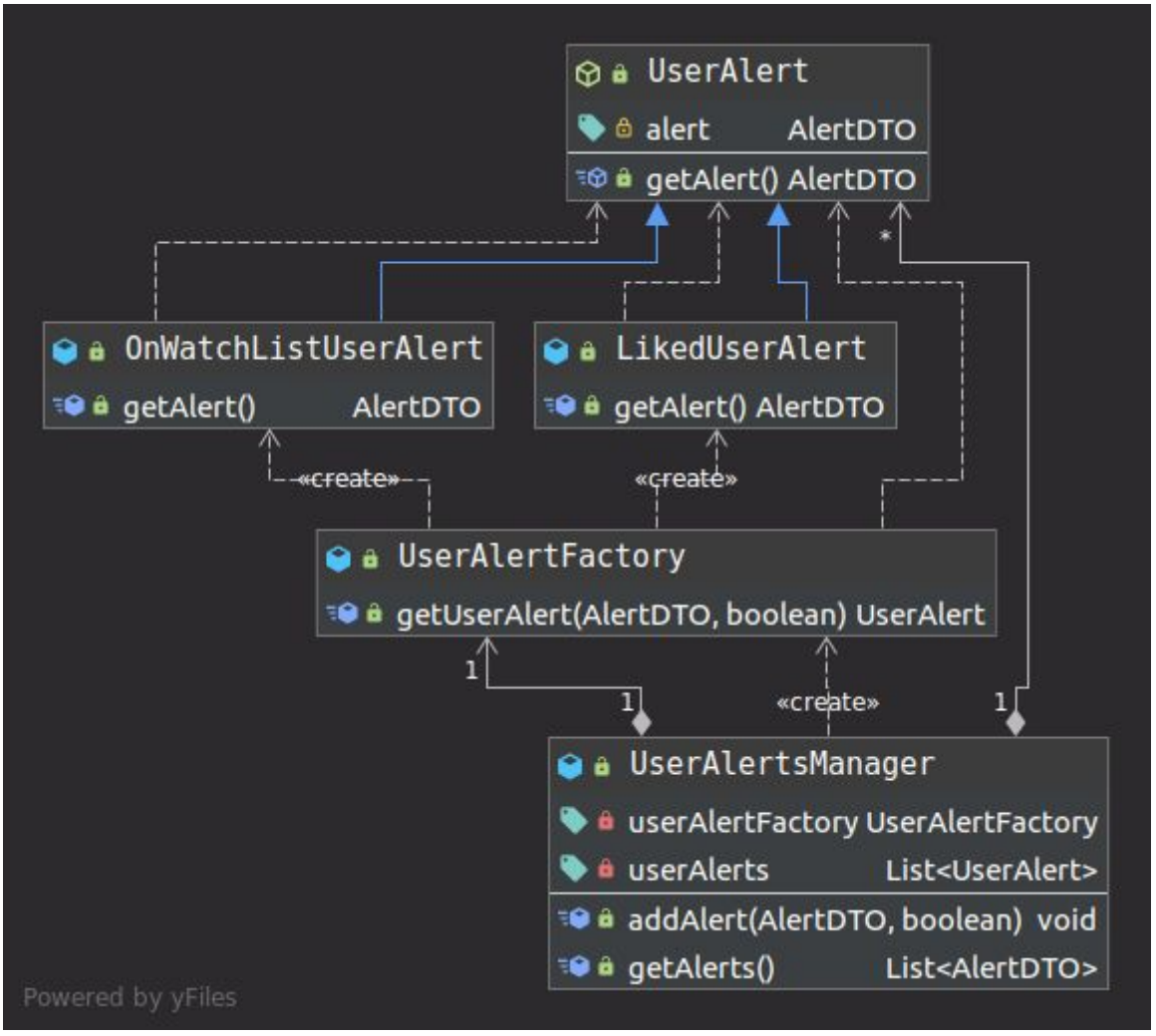

Diagramme UML du module UserAlertsManager

- *UserAlert* : Classe abstraite encapsulant une alerte et définissant la méthode abstraite *getAlert*.
- *OnWatchListUserAlert* : étend *UserAlert* en définissant *getAlert* pour ne retourner que des valeurs *null*.
- *LikedUserAlert* : étend *UserAlert* en définissant *getAlert* pour ne retourner que le champ encapsulé *alert*.

- *UserAlertFactory* : Utilise le **factory pattern** pour instancier les deux classes précédentes. On lui passe en argument une alert et un boolean (si le show est “liké” ou non.
- *UserAlertManager* : Utilise *UserAlertFactory* pour instancier des objets UserAlert. Un **observer pattern** est mis en place afin d’appeler la fonction *getAlert* de chaque objet. Ainsi nous encapsulant totalement la logique de retourner ou non l’alerte

Gestion des exceptions

Nous nous sommes assurés de bien gérer toute exception pouvant survenir, et ce sur les 3 principaux points suivants :

- Les appels à l’API TMDB : Des problèmes peuvent survenir facilement en utilisant le protocole HTTP pour communiquer.
- Récupérer les objets dans la base de données: Afin d’éviter des erreurs au moment de récupérer des données, nous avons utilisé les *Optional* Java afin de wrapper les entitées. Ainsi, si une entitée n’est pas trouvée, l’objet correspondant est *null*. La classe *Optional* nous permet alors d’effectuer des transformations sur cette entité. Si elle est *null* la transformation n’est pas appliquée et nous évitons une *NullPointerException* au runtime.
- Persister des objets dans la base de données: Afin de fournir de meilleurs informations aux utilisateurs de l’API (Curl, Postman, le front React), nous avons crée des Exceptions simples nous permettant de statuer l’état d’une requête API (*UserNotFoundException*, *ShowIsAlreadyLikedException* etc…).

Originalité

Nous avons choisi de créer une interface graphique simple mais fonctionnelle en React afin de pouvoir nous concentrer sur l’originalité de notre projet : La mise en place d’une CI complète, trois environnements (développement, pré-production, production), des migrations automatiques et des pré-commit hooks permettant de

formater notre code. Ce choix est motivé par le fait que nous étions curieux de comprendre comment une équipe travaillant sur un projet Spring pouvait mettre en place des moyens de travailler à distance. Nous souhaitions en effet éviter tout problème relatif à nos environnements locaux, obtenir un code uniformisé et lisible, pouvoir tester à tout moment nos nouvelles features et enfin nous affranchir des problèmes survenant lors du changement du modèle relationnel de notre base de données.

Plus précisément, la CI définie dans le fichier *".travis.yml"* nous permet d'effectuer les opérations suivantes de manière automatique (le nom des branches concernées est indiqué en gris) :

1. Packaging de l'application Spring pré-production - production - pull request
2. Build et Tag de l'image Docker pré-production - production
3. Déploiement de l'image docker sur Heroku pré-production - production
4. Lancement de l'application Spring dans le container sur Heroku pré-production - production
5. Migration de la base de donnée hébergée sur Heroku pré-production - production
6. Rebase de la branche de pré-production sur la branche master production
7. Build de l'application React pré-production - production
8. Déploiement de l'application React pré-production - production

De plus, nous proposons un environnement de développement local qui se lance très simplement. Celui-ci est défini dans le fichier *"docker-compose.yml"* et permet de lancer localement:

- Un container pour l'application Spring
- Un container pour l'application React
- Un container pour la base de données

Enfin, les pré-commit hooks sont définis dans le *pom* à la racine du projet. La classe *MigrationManager* permet d'effectuer les migrations SQL définies dans le dossier *"./src/main/resources/db/migration"*.

Notre méthodologie

Nous avons utilisé *GitHub* pour le versionnement de notre code, le repository est le suivant : https://github.com/hinosxz/chowchow. Chaque modification a été soumise par *Pull Request* et validée par au moins un membre du groupe.

Nous avons également utilisé *Travis CI* pour une intégration et un déploiement continu sur *Heroku* pour les environnements de *staging* et *production*. Cela nous a permis d'itérer très rapidement sur le projet en groupe.

Chaque migration du schéma de base de données est décrite dans un fichier SQL et déployée automatiquement par l'application Spring grâce à Flyway.

D'un point de vue méthodologie projet, nous avons utilisé Trello : https://trello.com/b/onkNqYJE/chowchow. Nous avons créé les cartes nécessaires à la réalisation du projet et nous nous sommes attribué ces cartes. Chaque carte est citée dans la *Pull Request* correspondant aux changements relatifs dans le code.

Enfin comme évoqué en introduction, les différents environnements nous ont permis d'avoir accès en local à un environnement de développement; à un environnement de test en pré-production afin de vérifier la compatibilité des nouvelles features; et à un environnement de production fournissant l'accès à une version stable de l'application