Rapport

Keschubay Jun

L'idée du projet était de permettre à l'utilisateur d'entrer une description de son humeur en continu, puis le service de communication (le service d'humeur) la soumettrait de manière asynchrone, et le service de statistiques serait en mesure de fournir des statistiques sur la fréquence. Ces statistiques seraient récupérées par le service d'humeur et associées à une couleur qui serait ensuite utilisée par l'interface utilisateur.

Architecture Implémentée

Le projet est conçu selon une architecture microservices, divisée en deux applications Spring Boot principales :

- **mood-service** : responsable de la soumission des humeurs par les utilisateurs et de la communication avec des services externes (API citation).
- **stats-service** : chargé de la persistance des données, de la consommation des messages via ActiveMQ, et de la fourniture des statistiques sur les humeurs.

Composants clés :

- **Communication asynchrone** : ActiveMQ queue («mood.queue») utilisée pour transmettre les humeurs du mood-service au stats-service.
- **Communication synchrone** : test de connectivité via endpoint REST (/ping).
- **Base de données** : stats-service utilise une base H2 en mémoire pour stocker les données d'humeur.
- **Authentification** : Intégration de Keycloak en tant que serveur d'authentification Oauth2 avec JWT (configuration gérée dans SecurityConfig).
- **Sérialisation JMS** : une configuration personnalisée JmsConfig permet la conversion correcte des objets JSON entre services.
- **API externe** : consommation de l'API ZenQuotes.io pour fournir des citations motivantes.
- **Swagger (Springdoc)** : documentation automatique de l'API REST.
- **Docker Compose** : permet le lancement simultané du broker ActiveMQ et Keycloak.

Problèmes, Résolutions, Choix

- **Manque d'idée initiale** : on a adopté un angle pratique à faible complexité mais cohérent avec les critères : suivi d'humeur, statistique par utilisateur, intégration d'API externe. Il a pris le plus de temps.
- **Docker-compose** : Au cours du projet, j'ai essayé d'implémenter clean docker compose pour tous les microservices et ai passé pas mal de temps si-dessous. Cependant, il s'est avéré difficile de les connecter correctement et le prix des erreurs était trop élevé pour mon ordinateur portable, comme il a freezé quand les services prisent 300%+ CPU. Afin de gagner du temps, j'ai abandonné l'idée de mettre en place tous les services via docker-compose. A la place, seuls activeMQ et keycloak sont restés, les autres devant être lancés manuellement.
- **Désynchronisation du binôme** : la charge a été assumée par une seule personne. Aucun blocage n'a été introduit.
- **Problèmes techniques de JMS** : erreurs de type entre services lors de la désérialisation d'objets Mood. Solution : uniformisation des DTO entre services.
- **Difficulté de test avec Keycloak** : a motivé la mise en place d'une interface frontend simple pour accéder à l'API avec token JWT.
- **Difficultés frontend**: La présence de l'interface utilisateur a été principalement motivée par le fait que la fonction principale du service renvoyait une couleur d'ambiance pour l'interface utilisateur. Mais le catalyseur final a été l'ennuyeux test avec des jetons, qui, rétrospectivement, aurait pu être évité par une mise en œuvre plus tôt des tests unitaires. Pour ma défense, je n'avais pas réalisé que la sécurité Spring offert les tests avec des utilisateurs fictifs (MockUser). Le web était besoin des permissions différentes que les tests via curl, alors j'ai crée un nouveau profil dans keycloak uniquement pour le UI. Par contre, je n'ai pas réussi à résoudre la problème avec services spring, alors, frontend est seulement utile pour récupérer l'access token confortablement pour le moment de rendu.
- **Difficultés de test** : l'idée du projet étant simpliste, j'ai commencé par des tests de fonctionnalité plutôt que des tests unitaires. Lorsque j'ai découvert security.test, il s'est avéré trop difficile à mettre en œuvre compte tenu des contraintes de temps. J'ai donc abandonné et testé manuellement via curl.

Planning Initial vs Effectif

Bien que le projet ait été conçu comme travail de semestre, la réalisation effective s'est concentrée sur les deux derniers jours avec une haute efficacité.

Bilan

Le projet atteint l'ensemble des objectifs techniques obligatoires :

- deux services REST indépendants
- persistance de données H2
- communication asynchrone et synchrone
- intégration d'API externe
- authentification par JWT
- documentation Swagger