IVVQ

Projet Magnetrade

2018-2019

https://github.com/M2DL/2019-ivvq-magnetrade

https://codecov.io/gh/M2DL/2019-ivvq-magnetrade

https://travis-ci.com/M2DL/2019-ivvq-magnetrade

https://shielded-reaches-59110.herokuapp.com/swagger-ui.html

https://sonarcloud.io/dashboard?id=M2DL_2019-ivvq-magnetrade

ANDRIAMISEZA Rialy

JACQUES Louis

CLÉMENT Hugo

LIEVRE Mathieu

UNIVERSITÉ TOULOUSE III PAUL SABATIER

MASTER DÉVELOPPEMENT LOGICIEL

Sommaire

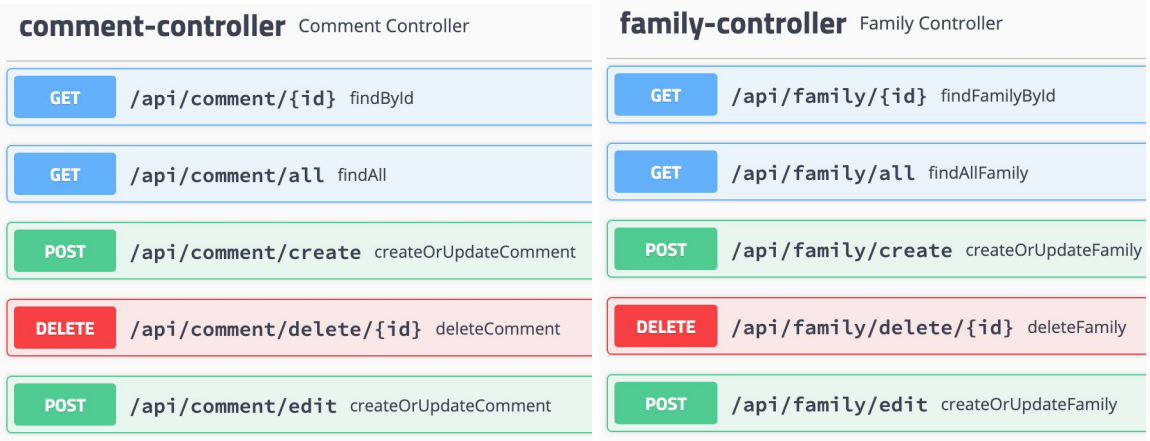
Description du projet

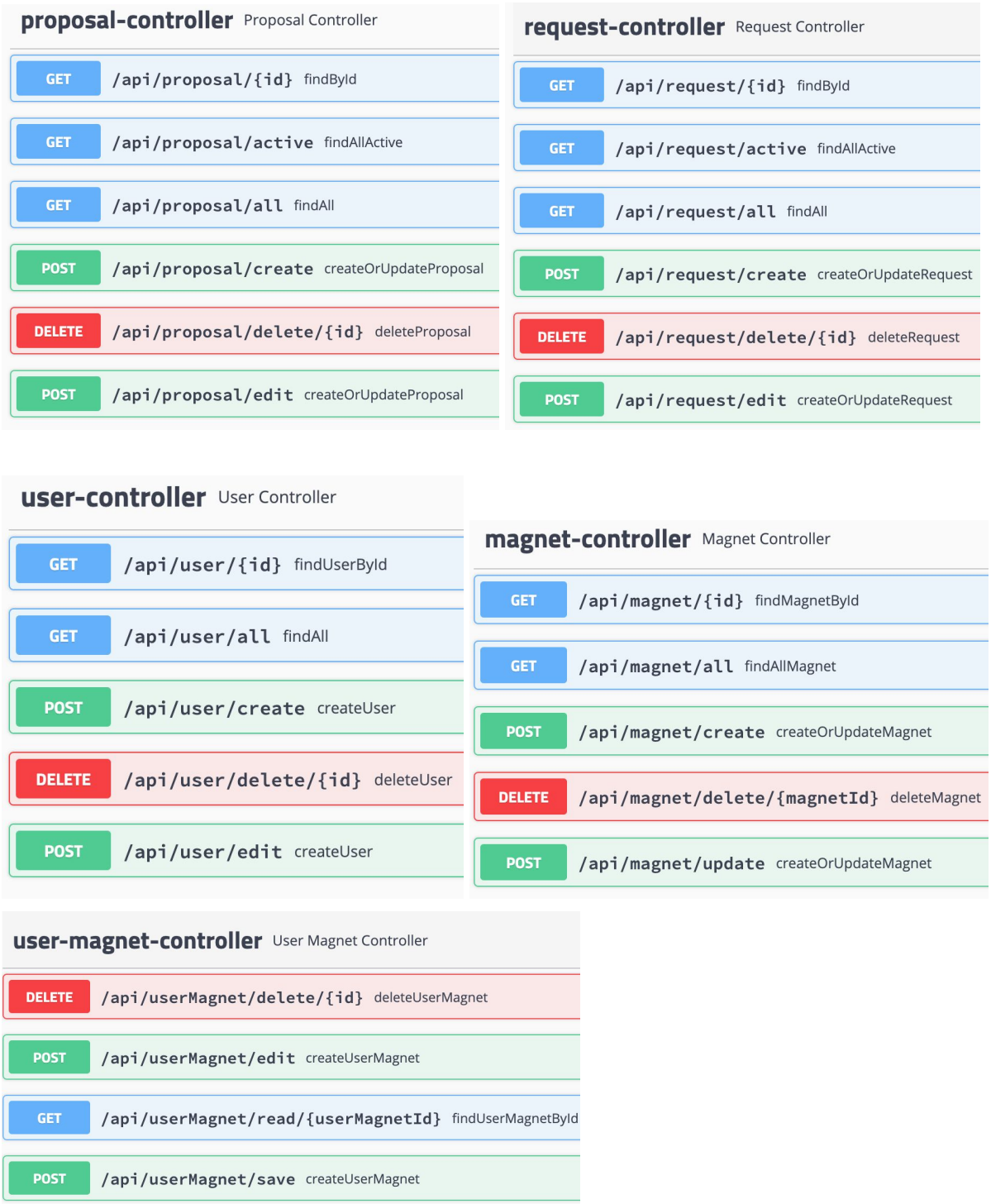
Magnetrade est un projet qui se base sur un constat flagrant de la société actuelle. Seule une minorité privilégiée de la population parvient à finir la carte de France faite à partir des magnets récupérés dans les cordons bleus ou encore la carte du monde à l'aide des précieux magnets récupérés dans les meilleurs quatre-quarts. En effet, selon la loi de Pareto astucieusement appliquée par les fournisseurs de magnets, environ 20% des consommateurs détiennent 80% des magnets les plus rares. Pour pallier à ces inégalités sociologiques, nous avons décidé de concevoir une application Web qui permettrait à tous les fervents collectionneurs de magnets – néophytes ou grands magnétophiles – de s'inscrire sur notre solution afin de :

- Visualiser l'ensemble des magnets qu'ils possèdent
- Savoir à quelle grande famille ils appartiennent
- Effectuer des échanges avec d'autres utilisateurs de la solution grâce à un mécanisme de demande/proposition
- Acheter à d'autres utilisateurs les magnets qui leur manquent
- Vendre leurs doublons aux utilisateurs qui cherchent ces magnets

API

Pour documenter et tester notre API nous avons utilisé l'outil Swagger.
Voici les méthodes disponibles sur notre API :





Pour plus de détails, vous pouvez retrouver la documentation complète de l'API avec Swagger : https://shielded-reaches-59110.herokuapp.com/swagger-ui.html.

Caractéristiques techniques

Comme exigé par l'équipe encadrante du projet, nous avons utilisé les outils et technologies suivants :

Langage	Java 8
Environnement de développement intégré	Intellij IDEA
Outil de build	Apache Maven (embarqué dans IntelliJ)
Framework Java	Spring Boot 2.0.5
Framework de test	JUnit 4.12
Système de contrôle de version	Git
Forge	GitHub Classroom (dépôt public créé automatiquement via Moodle)
Gestion des exigences	Issues GitHub
Suivi et répartition des tâches	Board GitHub intégré
Plateforme d'Intégration Continue	Travis CI
Couverture de code	JaCoCo
Déploiement	PaaS Heroku
Virtualisation applicative	Docker

De plus, nous avons effectué certains choix complémentaires :

Framework Front-end [1]	Angular 6
Outils d'analyse statique de code	SonarCloud Plugin SonarLint pour IntelliJ
Frameworks de Mock pour les tests	Spring MVC Test (MockMvc) Mockito
Licence logicielle	MIT

[1] Le front n'étant pas prioritaire dans le cadre du projet, il n'est pas complet. Il est cependant intégré dans le processus DevOps (bien qu'Heroku n'en permette pas l'affichage).

Architecture logicielle

Nous avons choisi de découper l'architecture de notre projet verticalement, en suivant une organisation par fonctionnalité (*package by feature*) plutôt que par couche (*package by layer*) généralement utilisée dans les autres matières comportant du J2EE. Ce choix a été effectué au début du projet pour nous permettre de sortir des sentiers battus en explorant une structure différente de nos habitudes, mais également parce qu'il nous paraissait plus pertinent d'opter pour ce découpage sachant que notre application n'allait pas embarquer de service très complexe ni de classes annexes (utils, exceptions, enums, config, etc.) transverses à toute l'application, dont la présence aurait certainement justifié une architecture horizontale.
Ainsi, nous pouvons observer l'architecture suivante :

Diagramme de paquetage

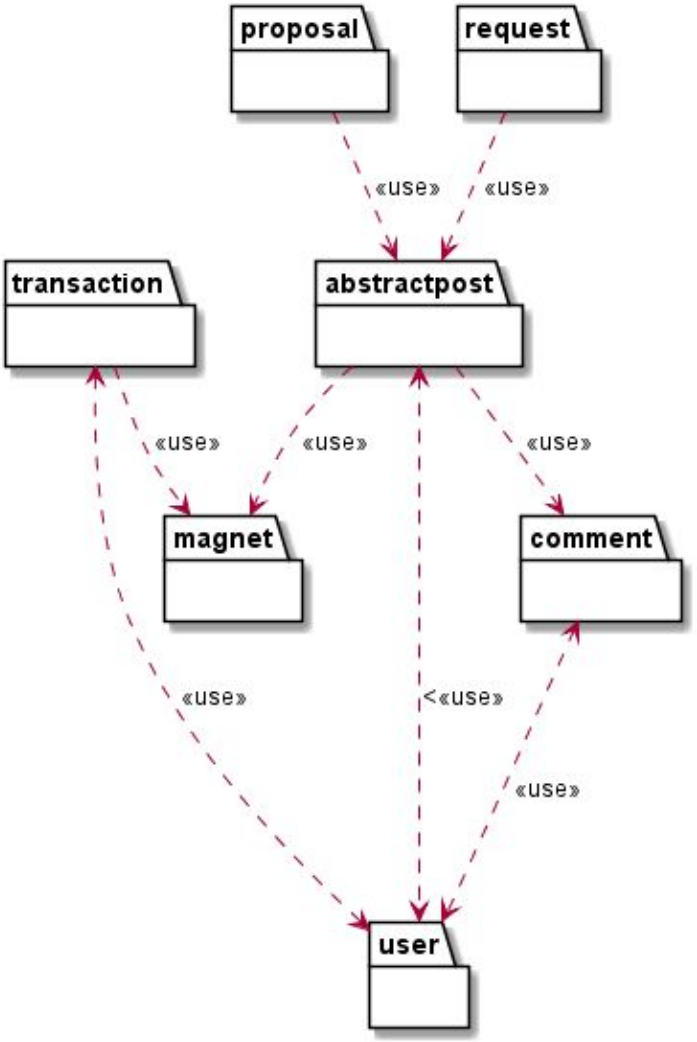
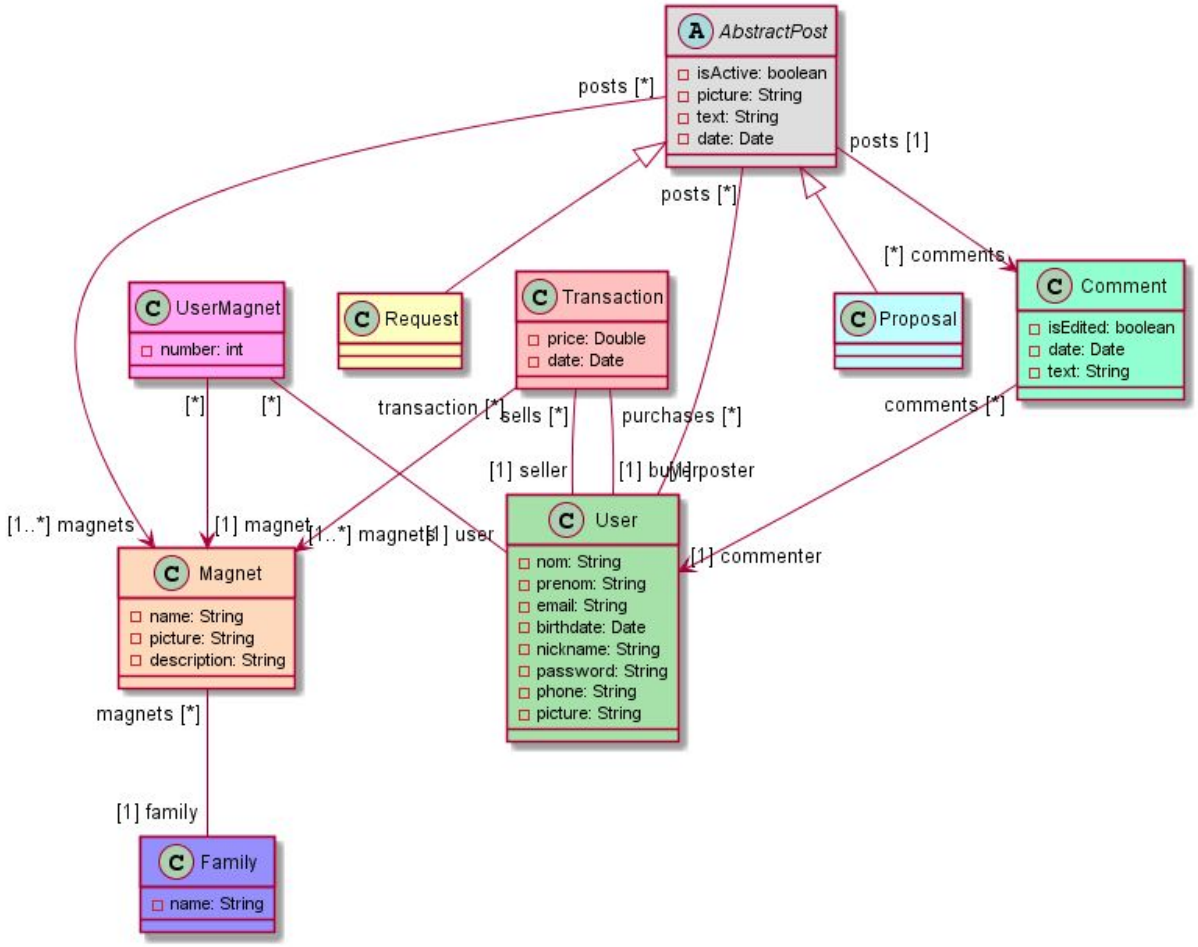
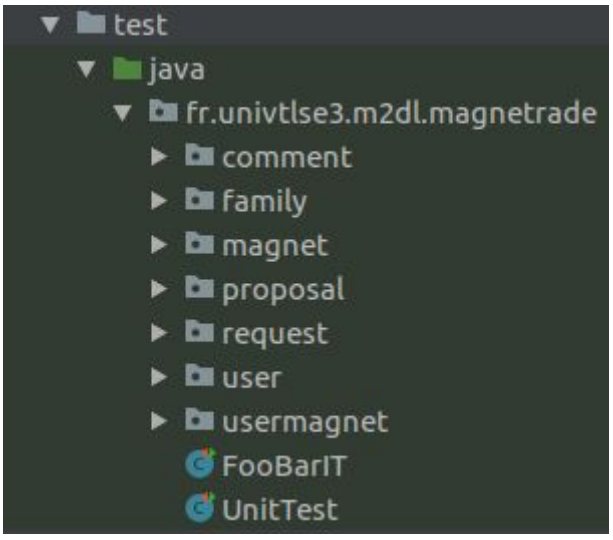


Diagramme de classes

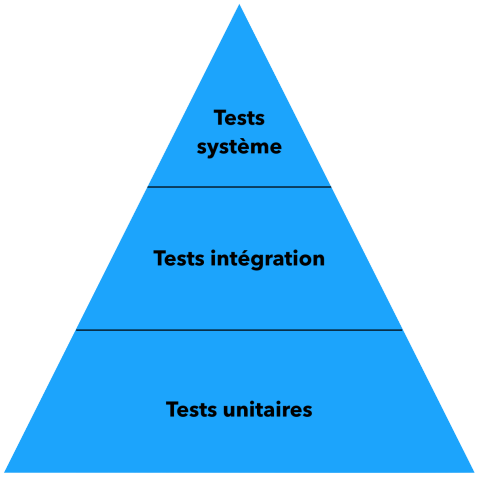


Politique de tests

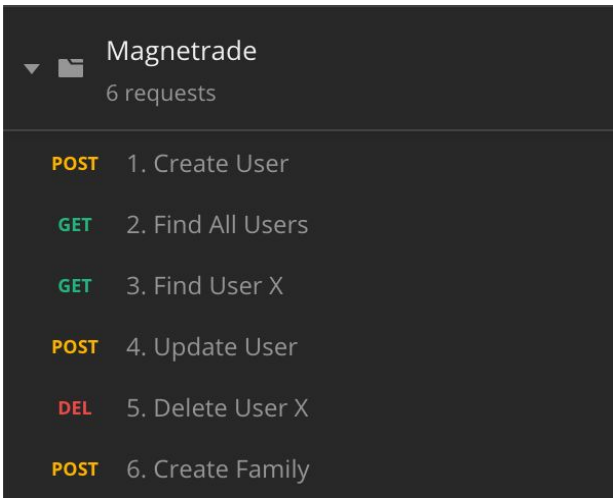
Lors du développement de Magnetrade, les tests unitaires ainsi que le code métier ont été réalisés en parallèle. En effet nous pouvions constater une constance entre code métier et couverture de test pour chaque incrément. Bien évidemment, une des conditions *sine qua non* pour valider une pull request était le passage à 100% des tests développés dans le nouveau code mais également des tests précédents, afin de garantir une non-régression. Il est aisé de comprendre que nous n'avons donc pas opté pour la méthode Test-Driven Development, bien que cela ait été envisagé au début du projet.
Dans un second temps, nous avons réalisé un incrément essentiellement centré sur les tests d'intégrations qui nous permettent de vérifier le bon fonctionnement de nos contrôleurs et de l'application de bout en bout.
Les tests unitaires et d'intégration sont regroupés dans les mêmes paquetages, suivant le découpage préalablement établi pour le code métier :

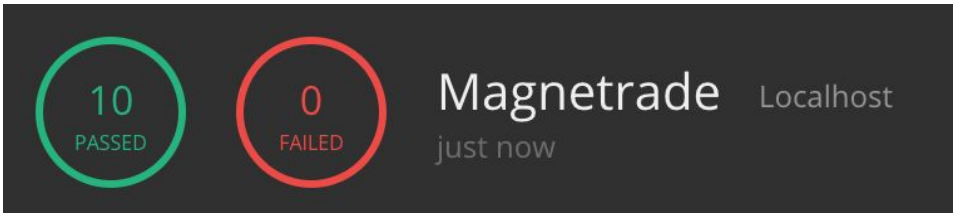


Nous avons essayé de respecter la pyramide des tests, nous avons donc concentré notre attention sur les tests unitaires, un peu moins sur les tests d'intégration et moins encore sur les tests système.



Pour les tests système, nous les avons principalement réalisés manuellement via Swagger. Mais nous avons également créé une collection Postman afin de pouvoir réutiliser rapidement des cas définis *(collection disponible sur GitHub)*.
Nous avons choisi de ne pas les automatiser via Travis car nous estimons qu'il ne faut pas faire ce genre de tests sur la base de données de production.





Workflow de développement

Nous avons choisis d'appliquer le GitHub Flow[2] lors du développement de notre projet. C'est une méthode de gestion du dépôt basé sur les branches. Une branche est créée depuis master, les développements sont effectués sur cette branche, puis sont fusionnés sur master via une Pull Request.
Nous avons choisi cette méthode car cela permet d'avoir un workflow très léger, de faire des revues de code (via la review de la PR) et de faire des déploiements réguliers.
Une branche par personne et par fonctionnalité a été créée lors des différents sprints. Pour chaque incrément fonctionnel, les branches ont été soumises à une review par les autres développeurs puis, mergées ou invalidées selon les cas.
Pour améliorer notre workflow git, nous aurions pu définir une convention de nommage de branches.

En ce qui concerne la qualité du code, nous avons décidé d'intégrer à nos postes de développement le plugin SonarLint afin de repérer en temps réel une majeure partie des problèmes (bugs, vulnérabilités ou code smells), ce qui nous a évité d'attendre que Travis lance l'analyse suite à un commit.
Suite aux résultats de l'analyse complète de SonarQube, nous avons ensuite fait des améliorations de codes généralement en fin de Sprint sur de nouvelles branches.

[2] https://guides.github.com/introduction/flow/

Notre “pipeline” consiste en l’enchaînement de tâches suivant :

- Génération du front (npm run build)
- Copie du front généré dans le back (afin de l’embarquer dans le Jar)
- Génération du back
    - Téléchargement des dépendances
    - Lancement des TU
    - Publication de la couverture de code des TU
    - Lancement des TI
    - Publication de la couverture de code des TI
    - Génération du Jar
- Lancement de l’analyse statique de code Sonar
- Publication sur Heroku (uniquement depuis la branche master)

Le pipeline est gérée par Travis CI avec utilisation d’une image Docker.