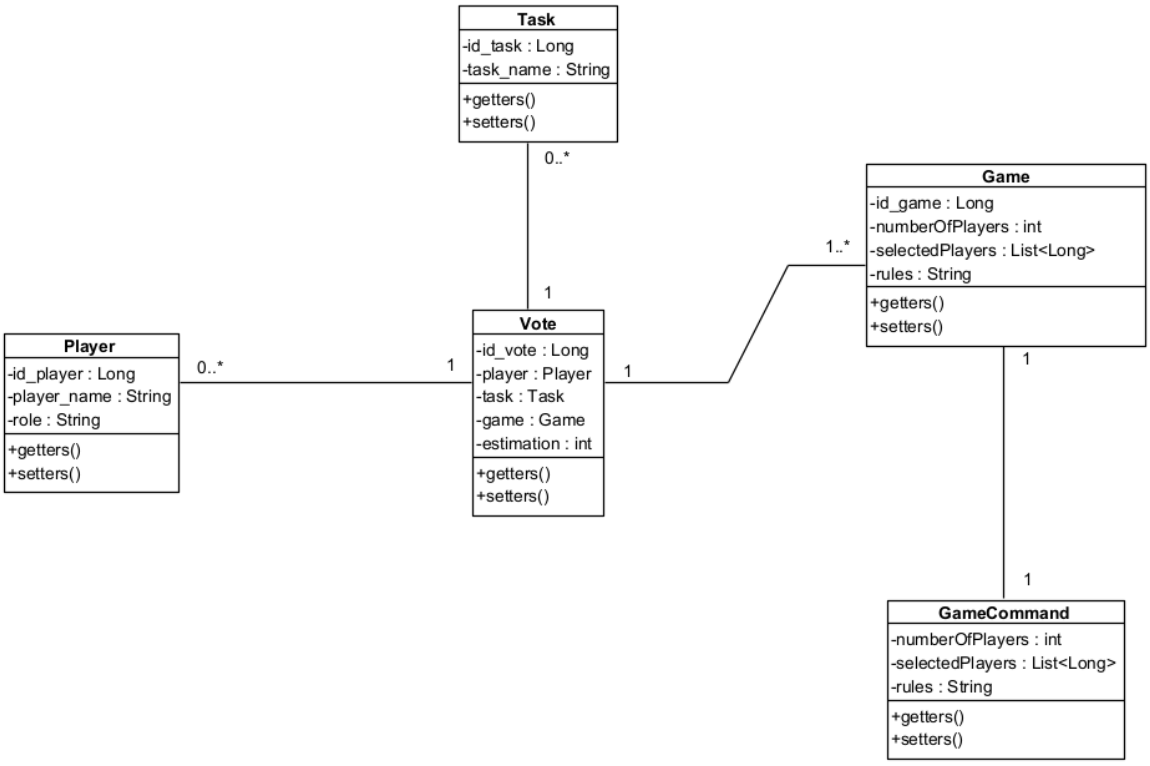
RAPPORT - Projet CAPI : Planning Poker

1. Présentation du projet

Nous avons développé un projet de Planning Poker dans le cadre de la gestion collaborative de projets agiles. Nous avons choisi comme exemple un projet e-commerce. Nous avons une interface conviviale permettant à une équipe de planifier et d'estimer les fonctionnalités de manière efficace. L'utilisateur, arrivant sur l'application, peut choisir de créer une nouvelle partie ou de rejoindre une partie en cours. En créant une nouvelle partie, l'utilisateur peut choisir le nombre de joueurs, ensuite sélectionner les joueurs parmi ceux qui sont déjà dans la base de données et enfin choisir le mode du jeu et démarrer la partie. Il arrive sur une interface où apparaîtront une à une les fonctionnalités enregistrées dans notre backlog en format JSON avec les cartes du jeu juste en dessous. Ces cartes sont associées à une estimation et l'utilisateur peut faire son choix. Grâce à l'utilisation de JUnit, l'application garantit la robustesse de ses fonctionnalités et assure la qualité du processus de planification. De plus, l'intégration continue avec Jenkins automatise les tests, assurant une cohérence dans le développement du projet. Dans notre projet, nous nous sommes orientées vers la programmation orientée objet (POO).

2. Diagramme de classes de conception

Ci-après le diagramme de classe de conception sur lequel se base notre application.



3. Patterns utilisés

Pour réaliser notre projet, nous avons opté pour l'utilisation de trois design patterns dont les patterns Repository, Data Transfer Object et Dependency Injection.

- **Repository pattern**

Le pattern Repository est utilisé pour la gestion des opérations de la base de données et pour encapsuler les détails de la persistance des données. Grâce à l'utilisation de ce pattern, l'application peut travailler avec des objets métier sans se soucier des opérations de stockage ou de récupération de données. L'interface JpaRepository fournit des méthodes prêtes à l'emploi telles que findAll, findById, save, delete, etc, pour interagir avec les bases de données relationnelles. Dans notre projet, les interfaces sont dans le package agile_project.repositories.

- **Data Transfer Object (DTO) pattern**

Le pattern DTO est un modèle de conception utilisé pour transférer des données entre les composants. Dans notre projet, il est mis en évidence avec la classe GameCommand qui est un objet de transfert de données utilisé pour transporter les données entre la vue et le contrôleur lors de la création d'une nouvelle partie de jeu. Elle permet de transférer uniquement les données nécessaires pour cette opération spécifique, ce qui permet d'éviter la surcharge des entités de modèle avec des données temporaires.

- **Dependency Injection (DI) pattern**

Le pattern DI permet de gérer les dépendances entre les composants d'une application en inversant le contrôle de leur création et de leur gestion par le conteneur Spring ou Spring IoC Container (Inversion of Control). Dans notre projet, l'annotation @Autowired permet d'injecter automatiquement les instances des interfaces dans les contrôleurs. L'utilisation de ce pattern permet de réduire le couplage entre les composants pour les rendre ainsi plus modulaires. Il permet donc la flexibilité, la testabilité et la maintenabilité du code.

4. Architecture : Model-View-Controller (MVC)

Dans notre projet, l'architecture principale utilisée est le modèle MVC. Chaque classe de l'application est placée dans un des trois composants de ce modèle. Il permet la séparation des responsabilités, favorise une conception modulaire, une maintenance simplifiée, une réutilisation du code et une adaptabilité aux changements.

Le **Modèle** représente la logique métier et les données de l'application. Les entités sont persistées dans une base de données PostgreSQL grâce à l'utilisation de l'API Java Persistence (JPA) avec les annotations @Entity, @Id, @GeneratedValue. Ces entités sont transformées directement en tables dans la base de données lors de l'exécution de l'application. Dans notre projet, ils sont placés dans le package agile_project.models.

La **Vue** est la responsable de l'affichage des données au travers de l'interface utilisateur. Dans notre projet, les vues sont dans le dossier template, elles ont été développées avec Thymeleaf.

Le **Contrôleur**, lui, gère les interactions entre la Vue et le Modèle. Il reçoit les requêtes des utilisateurs depuis la Vue, effectue les opérations nécessaires sur le Modèle, puis renvoie la réponse à la Vue. Dans notre projet, ils sont placés dans le package agile_project.controllers.

5. Langage

Pour le développement de notre application, nous avons choisi le langage Java avec Spring Boot et Thymeleaf.

Java est un langage de programmation polyvalent et robuste, apprécié pour sa portabilité et sa performance. Grâce à son écosystème mature, Java offre une vaste bibliothèque standard et une communauté active.

Spring Boot est un framework Java qui simplifie le processus de développement en fournissant des conventions par défaut et des mécanismes d'auto-configuration. Facile à utiliser, Spring Boot favorise la création d'applications modulaires grâce à son système d'injection de dépendances. Il intègre également des fonctionnalités telles que la gestion des transactions et l'accès aux données via JPA.

Thymeleaf est un moteur de template conçu pour être intégré de manière transparente avec le framework Spring. Sa syntaxe HTML naturelle rend le code lisible et intuitif. Thymeleaf facilite la création de pages web dynamiques en permettant l'incorporation simple de code Java dans les fichiers de modèle. Son rendu côté serveur contribue à la création d'interfaces utilisateur réactives et interactives.

6. **Dépendances**

Les dépendances nécessaires à la réalisation de notre application se trouvent dans le fichier pom.xml à la racine du projet.

Les dépendances **spring-boot-starter-data-jdbc** et **spring-boot-starter-data-jpa** simplifient l'accès aux bases de données en fournissant des abstractions et des fonctionnalités de mapping objet-relationnel (ORM) pour JDBC et JPA. Ces starters facilitent l'intégration de la couche de persistance dans l'application.

La dépendance **spring-boot-starter-data-rest** simplifie la création d'une API REST en exposant automatiquement les entités JPA comme des points de terminaison REST. Cela permet une manipulation facile des données via des opérations CRUD standard sur les entités.

Les dépendances **spring-boot-starter-thymeleaf** et **spring-boot-starter-web** facilitent le développement d'applications web en intégrant Thymeleaf comme moteur de templates et en fournissant les composants nécessaires pour construire des applications web avec Spring Boot.

La dépendance **spring-boot-starter-test** regroupe les dépendances nécessaires pour les tests unitaires et d'intégration. Elle inclut des bibliothèques telles que JUnit, AssertJ et Hamcrest, offrant un environnement complet pour le développement et la validation du code.

La dépendance **spring-boot-devtools** améliore la productivité du développeur en facilitant le rechargement automatique de l'application pendant le développement. Cela permet des cycles de développement plus rapides en évitant la nécessité de redémarrer manuellement l'application après chaque modification.

La dépendance **org.postgresql:postgresql** fournit le pilote JDBC nécessaire pour établir une connexion avec une base de données PostgreSQL. Elle permet à l'application d'interagir efficacement avec une base de données PostgreSQL.

7. **Documentation**
8. **Intégration Continue**