Vékanban

I. INTRODUCTION

1. RAPPEL DU SUJET

Pour notre projet annuel, il nous a été demandé de concevoir une application Java. Cette application doit être un client lourd , possédant une interface graphique de type JavaFX. Elle devra avoir pour but de gérer l'équipe de développement du projet (gestion et définition des tâches, planification, affectation de ressources humaines aux tâches, gestion de tickets...). Un système d'extension de type Plug-in devra être présent. Il faudra vérifier que le plug-in pouvant être construit par 1 tier possède bien la liste des fonctions attendues et que celui-ci s'intègre à l'application sans recompilation. Une gestion des différents plugins devra être intégrées dans l'interface.

2. APPLICATION CHOISI

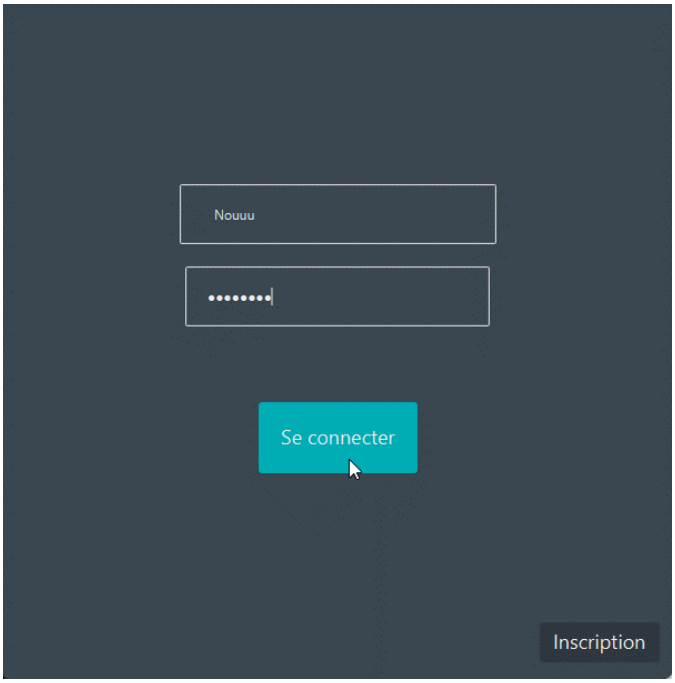
Nous avons choisi de réaliser une application de gestion de projet agile à l'aide de la méthode Kanban. Elle permettra à l'équipe de développement , ainsi qu’aux consultants de suivre l'avancement du projet. Cette application se comportera exactement comme un Trello. Elle permettra de gérer différents projets, la gestion des tâches à faire, en cours et terminé ainsi que les personnes assignés à ces dernières. Il faudra pouvoir gérer également les dates d'échéance. Pour l'implémentation du plug-in, nous avons pensé un plug-in permettant d'importer les tâches d'un projet Github au sein même de notre application.

II. Focus sur l’application

1. Connexion & Inscription

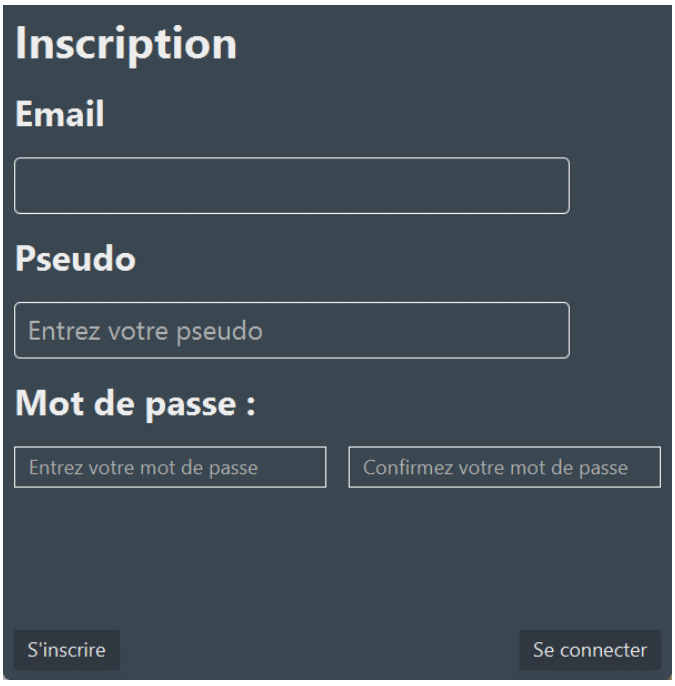
Connexion

Pour pouvoir utiliser l'application, il faut posséder un compte sur celle-ci.
Il suffit ensuite de se connecter en indiquant son pseudo et son mot de passe.



Inscription

Il est possible de s'inscrire directement depuis l'application. Pour ce faire il suffit de cliquer sur le bouton inscription qui se situe sur la page de connexion.
Il faut ensuite renseigner son adresse mail qui doit être unique son pseudo qui doit être unique également ainsi que son mot de passe.



Déconnexion

Il est possible de se déconnecter de l'application grâce au bouton log out présent en bas à gauche du menu.

2. Mes projets

Sur cette page, il est possible de voir les différents projets que nous avons créés ainsi que ceux auxquels nous avons été invités par d'autres.

Liste de mes projets

La première ligne affiche les projets que nous avons créés. Il est possible depuis ici de supprimer le projet.

Partagé avec moi

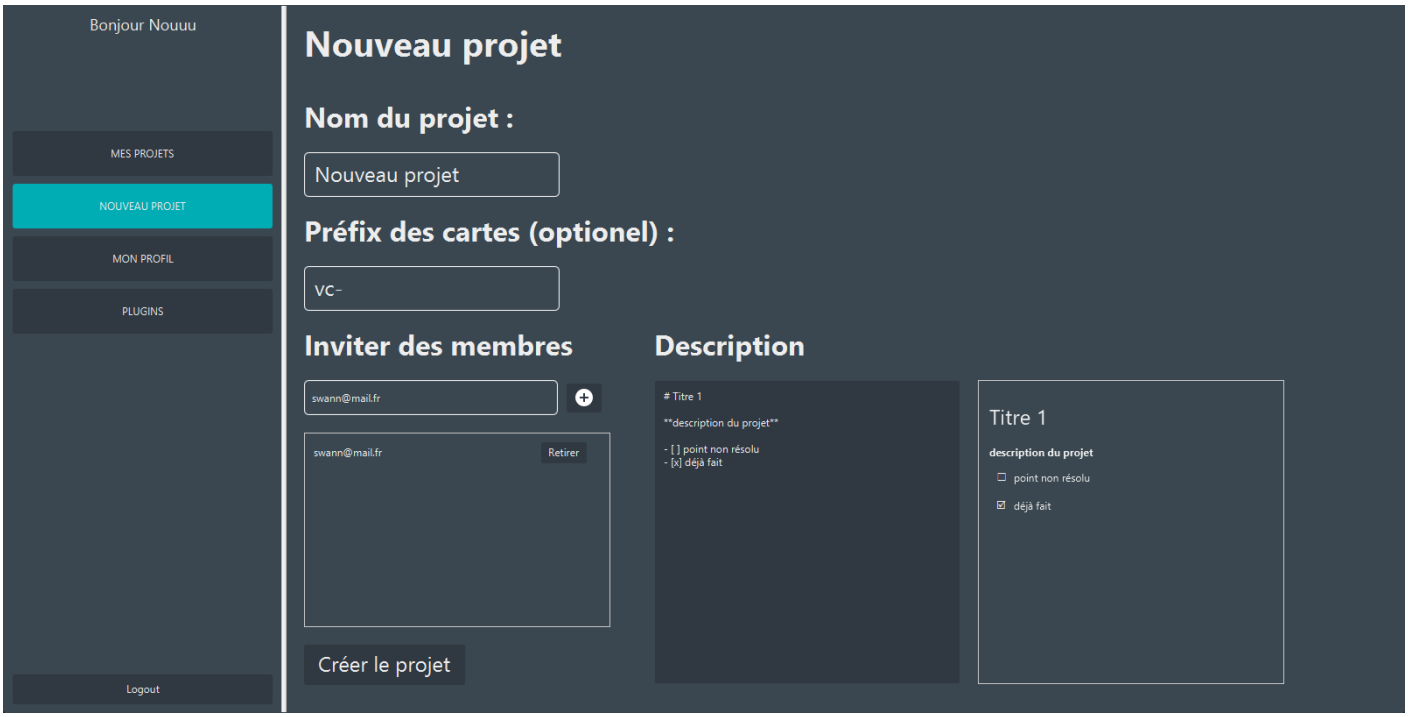
La 2e ligne affiche les projets auxquels nous avons été invités et pour ceux-ci il n'est pas possible de les supprimer, car seule le propriétaire a le droit de le faire.



3. Nouveau projet

Lorsque nous arrivons sur l'interface de création d'un projet, plusieurs champs sont à remplir dont certains obligatoires.

Il faut y renseigner le nom du projet, le préfixe des cartes si l'on souhaite inviter des membres à partir de leur adresse mail (il faut que le membre ait un compte existant sur l'application) ainsi qu'une description. La description prend en charge le markdown.

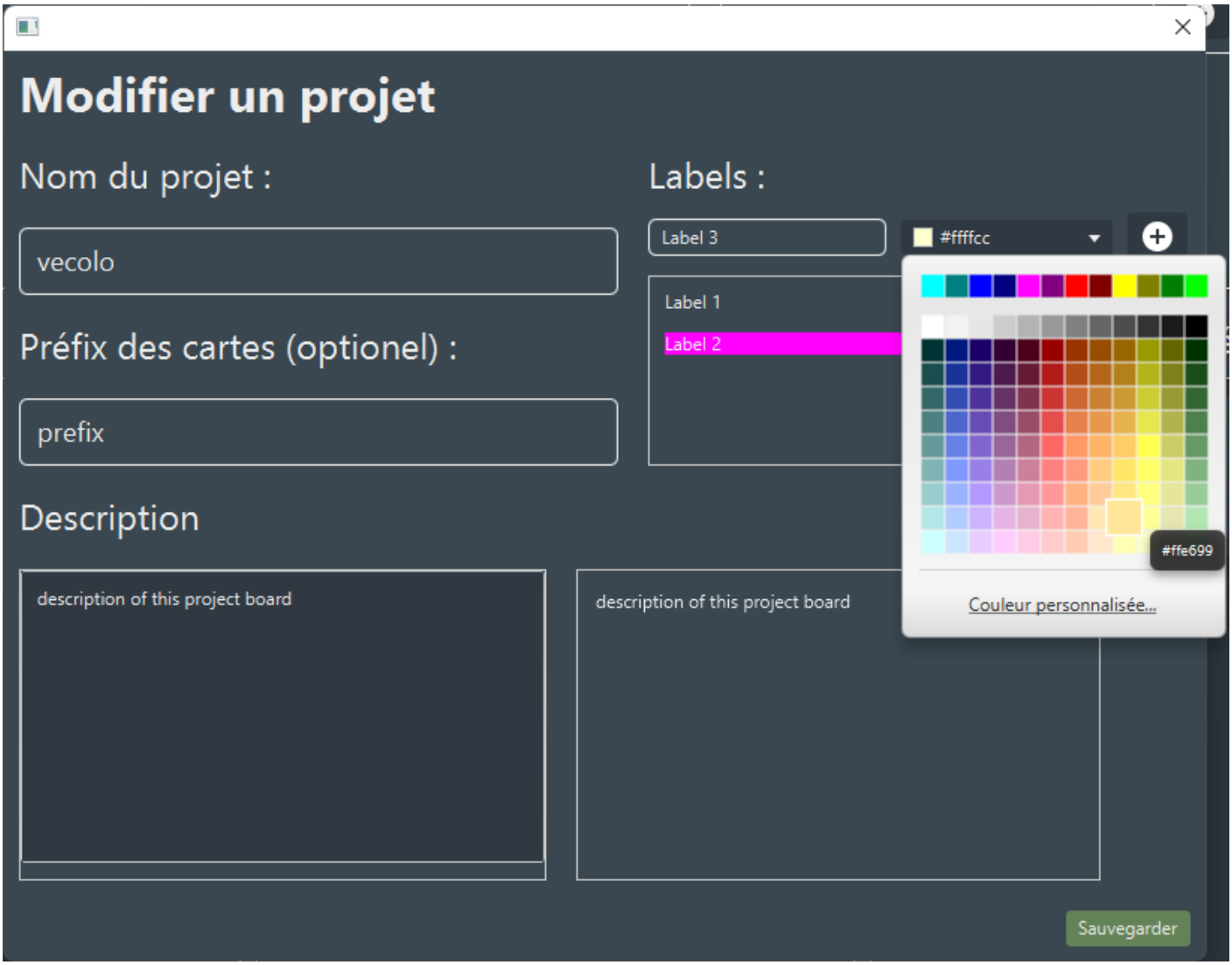


4. Consultation d’un projet

On peut consulter un projet en cliquant dessus depuis, la liste des projets.

Les champs d'administration du projet en lui-même sont réservés aux propriétaires (Nom du projet, description, préfixe et labels.).

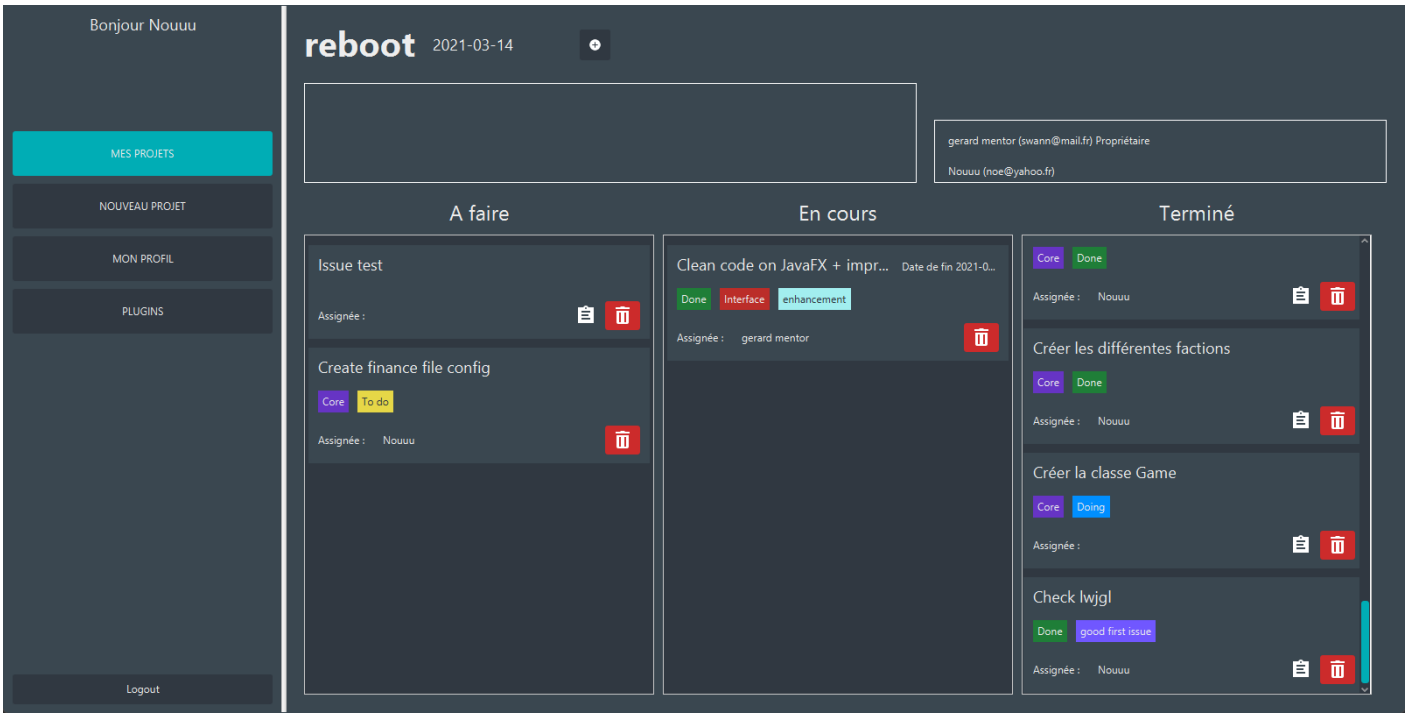
Il est possible pour le propriétaire de créer des labels avec des codes couleurs qui pourront ensuite être assignés à des cartes pour une meilleure visibilité.



Seul le propriétaire peut inviter où retirer des membres au projet.

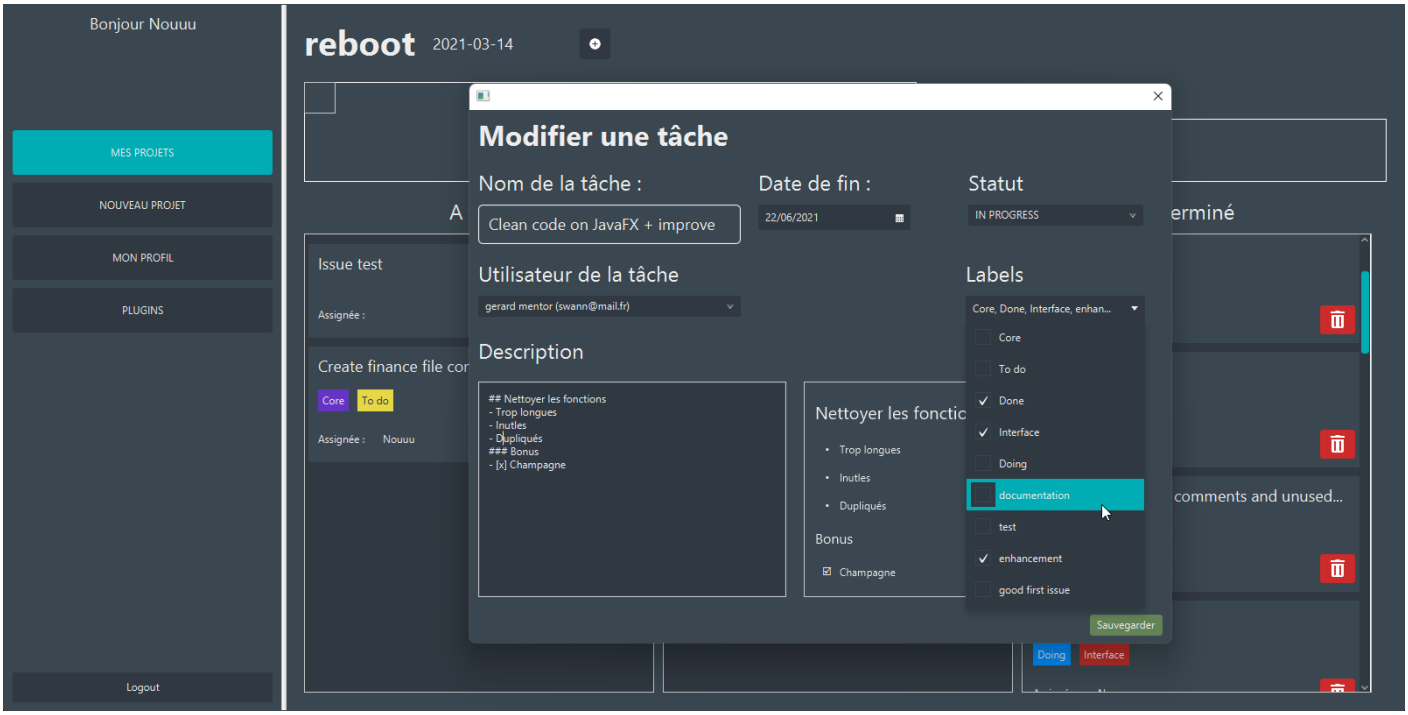
On peut consulter sur l'interface le nom du projet, sa date de création, création, sa description, les membres du projet. Ainsi que la liste des tâches.

La liste des tâches est séparée en 3 colonnes. À faire, En cours, Terminé.



Pour éditer une tache, il faut cliquer dessus. S'ouvre alors une fenêtre depuis laquelle nous pouvons modifier le nom de la tâche, la date d'échéance de celle-ci, son statut, ses différents labels, l'utilisateur assigné ainsi que sa description.

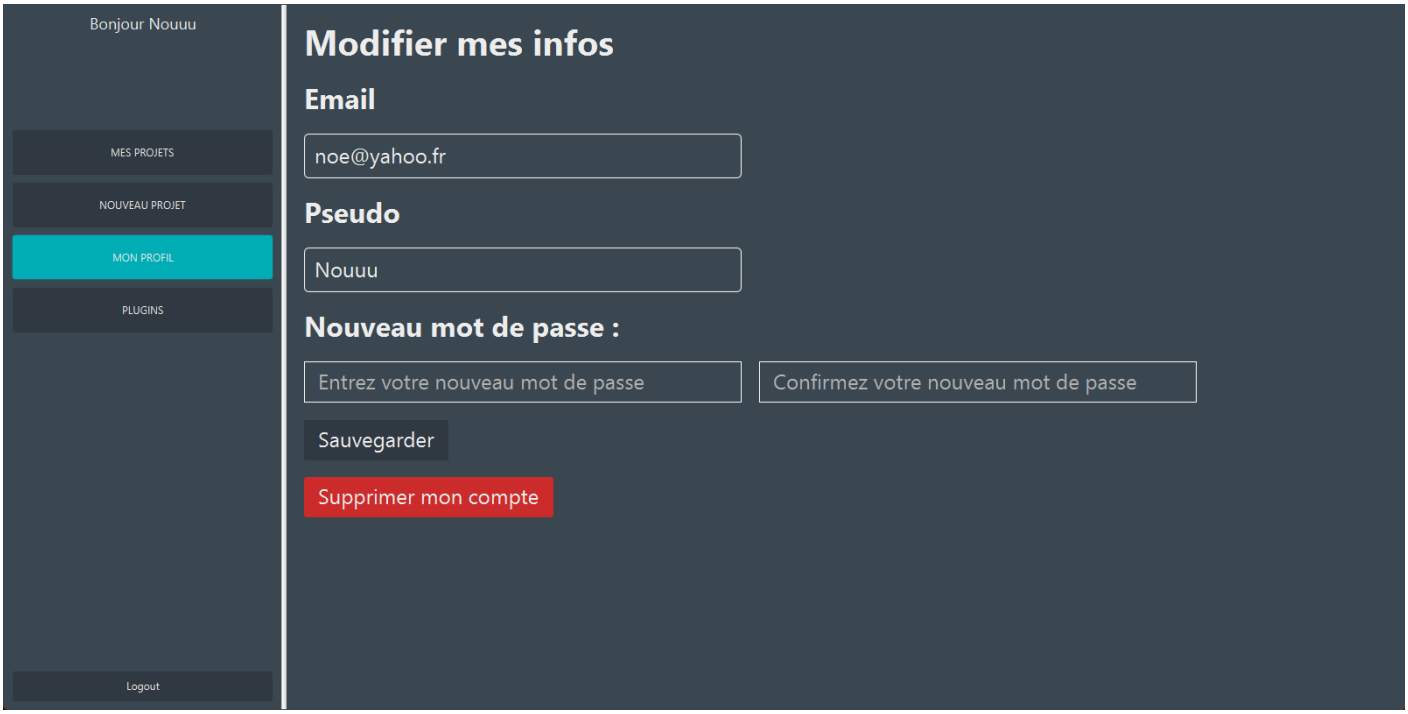
Comme pour la description du projet, la description d'une tâche supporte le markdown.



5. Mon profil

L'application permet de modifier ses informations personnelles comme son mot de passe, son adresse mail, ou encore son pseudo.

On peut supprimer son compte. Attention, cela supprimera tous les projets dont on est propriétaire et nous désassignera de toutes les cartes des autres projets.



6. Plugins

Un menu est disponible pour consulter les plugins chargés dans l'application et permet de voir le nom du plug-in ainsi que sa description.



Il suffit ensuite de cliquer sur celui-ci, renseignez les champs nécessaires et le lancer.

Dans le cadre de notre plug-in d'import Github, il faut renseigner le nom du dépôt ainsi que le nom du projet local sur lequel importer les tâches. La fenêtre se ferme automatiquement une fois que la tâche a été effectuée.



III. Choix d'implémentations

Technologies et librairies principales utilisées

Java FX

C'est la librairie principale imposée dans le sujet. Java FX permet d'avoir une interface graphique à l'aide de contrôleurs Java et est donc indispensable à notre application.
Nos fichiers d'interface sont sous la forme de fichier XML balisant les différents widgets présent sur les fenêtres de notre application.
Une sous librairie intitulée controlsfx a également été utilisé dans le projet et permet d'utiliser des widgets qui ne sont pas disponibles de base dans la librairie Java FX. Notamment le widget permettant de sélectionner plusieurs labels lors de la création d'une tâche.

Spring Boot

C'était le grand défi de ce projet. Spring boot est habituellement un framework orienté back end. Cependant, son système d'injection de dépendance ainsi que son ORM est très intéressant.
L'idée était de l'utiliser pour faciliter l'utilisation de Java FX qui de base n'est pas très simple côté code. Cela nous a permis de garder un contexte des contrôleurs actifs de notre application graphique et a énormément enrichi l'application finale. Son ORM nous permettait aussi de manipuler des objets comme de vraies entités de base de données a permis une meilleure intégration de ceux-ci au sein de l'interface graphique.
L'implémentation de base était complexe, mais le résultat final est très satisfaisant !

Flexmark

Flexmark et une librairie Java permettant de parser du texte et retrouver l'architecture markdown de celui-ci. Il nous a permis d'implémenter la prise en charge du markdown dans l'interface graphique à l'aide de balises CSS personnalisées ajoutées sur nos textes rendus à l'utilisateur.

SL4J

SL4J (Simple Logging Facade for Java) permet une implémentation de log plus facile au sein de l'application. Elle permet également de séparer les logs d'erreur des lots standard et de sauvegarder tout ça dans des fichiers.

PF4J

PF4J (Plugin Framework For Java) est un framework d'injection d'extension.
Il permet de charger au démarrage de l'application différents plugins compilés en .jar et de les exécuter au besoin. La version spécifique pour Spring nous permet d'injecter le contexte de notre application au sein du plug-in, ce qui permet à ce dernier d'utiliser le système d'injection de dépendance et les services mis à disposition par notre client lourd.

Mariadb JDBC

Mariadb JDBC est un driver permettant à l'ORM de Spring de se connecter à la base de données distante qui est sous Maria DB

Github API (pour le plugin)	GitHub  Rest API V3
Github API et une librairie Java permettant d'interagir avec l'API de Github avec des classes et des objets. Elle a été utilisée au sein du plug-in que nous avons développé pour faciliter la récupération des tâches sur un dépôt Github.	

2. Architecture du code

Chargement du contexte Java FX dans Spring

Le point d'entrée de l'application est celui de Spring. Il faut donc lui indiquer dès le départ qu'il doit charger une application Java FX.

Java FX a une classe principale se nommant **Application**, c'est le point d'entrée.

Nous enregistrons ainsi dans le contexte Spring une seule instance de cette application afin de pouvoir derrière utiliser l'injection de dépendance dessus.

Nous enregistrons également d'autres Bean propre à Java FX afin que tout fonctionne correctement.

```
public class VekanbanApplication extends Application {

    private ConfigurableApplicationContext context;

    @Override
    public void init() throws Exception {

        ApplicationContextInitializer<GenericApplicationContext> initializers =
                ac -> {
                    ac.registerBean(Application.class, () -> VekanbanApplication.this);
                    ac.registerBean(Parameters.class, this::getParameters);
                    ac.registerBean(HostServices.class, this::getHostServices);
                    ac.registerBean(PluginConfiguration.class);
                };

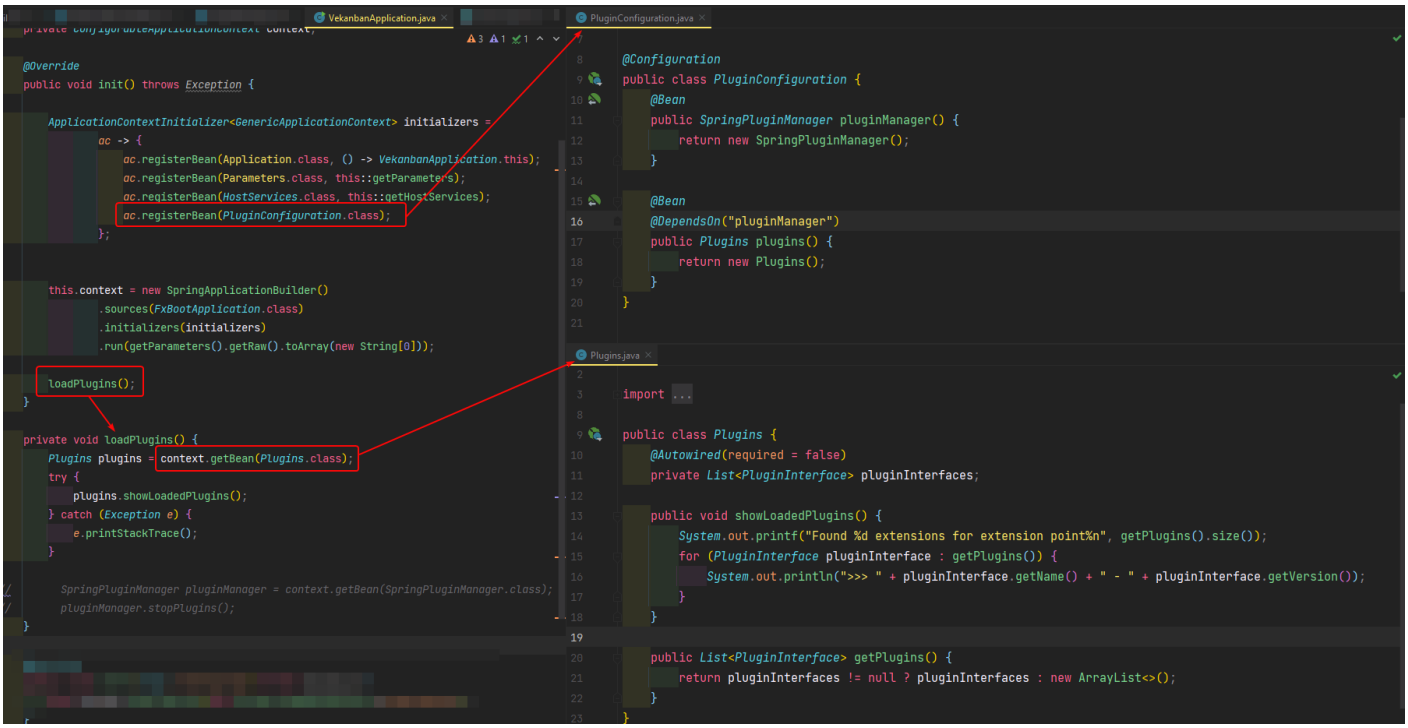

        this.context = new SpringApplicationBuilder()
                .sources(FxBootApplication.class)
                .initializers(initializers)
                .run(getParameters().getRaw().toArray(new String[0]));

        loadPlugins();
    }
```

Chargement des Plugins

Les plugins utilisent l'injection de dépendances, ils sont alors eux-mêmes considérés comme une dépendance injectable par l'application principale.

Une fois le contexte Spring chargé au démarrage, ce sont aux plug-ins d'être détectés.



Architecture MVC

L'architecture du projet respecte le modèle MVC (Model, Vue, Controller).

Bien entendu, une adaptation a été nécessaire pour respecter le fonctionnement de Java FX et celui de Spring.

- Nos modèles correspondent aux classes de nos entités (utilisateur, projet, tâche, etc.).
- Nos vues sont les contrôleurs de Java FX qui font l'interface utilisateur.
- Enfin, nos contrôleurs correspondent aux services associés au modèle qui vont effectuer les traitements de données.

Intégration de l’ORM

Pour intégrer correctement Hibernate, nous avons besoin de 3 types de classes différentes.

- Tout d'abord les entités (ou modèles) qui vont représenter nos tables en base de données et qui correspondent à des objets Java.

```java
@Entity
@Table(name = "USER")
public class User extends DateAudit {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "USER_ID", updatable = false, nullable = false)
    private long id;

    @Column(name = "USER_EMAIL", unique = true, updatable = true, nullable = false)
    private String email;

    @Column(name = "USER_PSEUDO", unique = true, updatable = true, nullable = false)
    private String pseudo;

    @Column(name = "USER_PASSWORD", unique = true, updatable = true, nullable = false)
    private String password;

    @OneToMany(mappedBy = "owner")
    private List<Board> owning;

    @ManyToMany(mappedBy = "members")
    private List<Board> boardsMember;

    @OneToMany(mappedBy = "assignedUser")
    private List<Card> Assignedcards;

    public User() {
    }
```

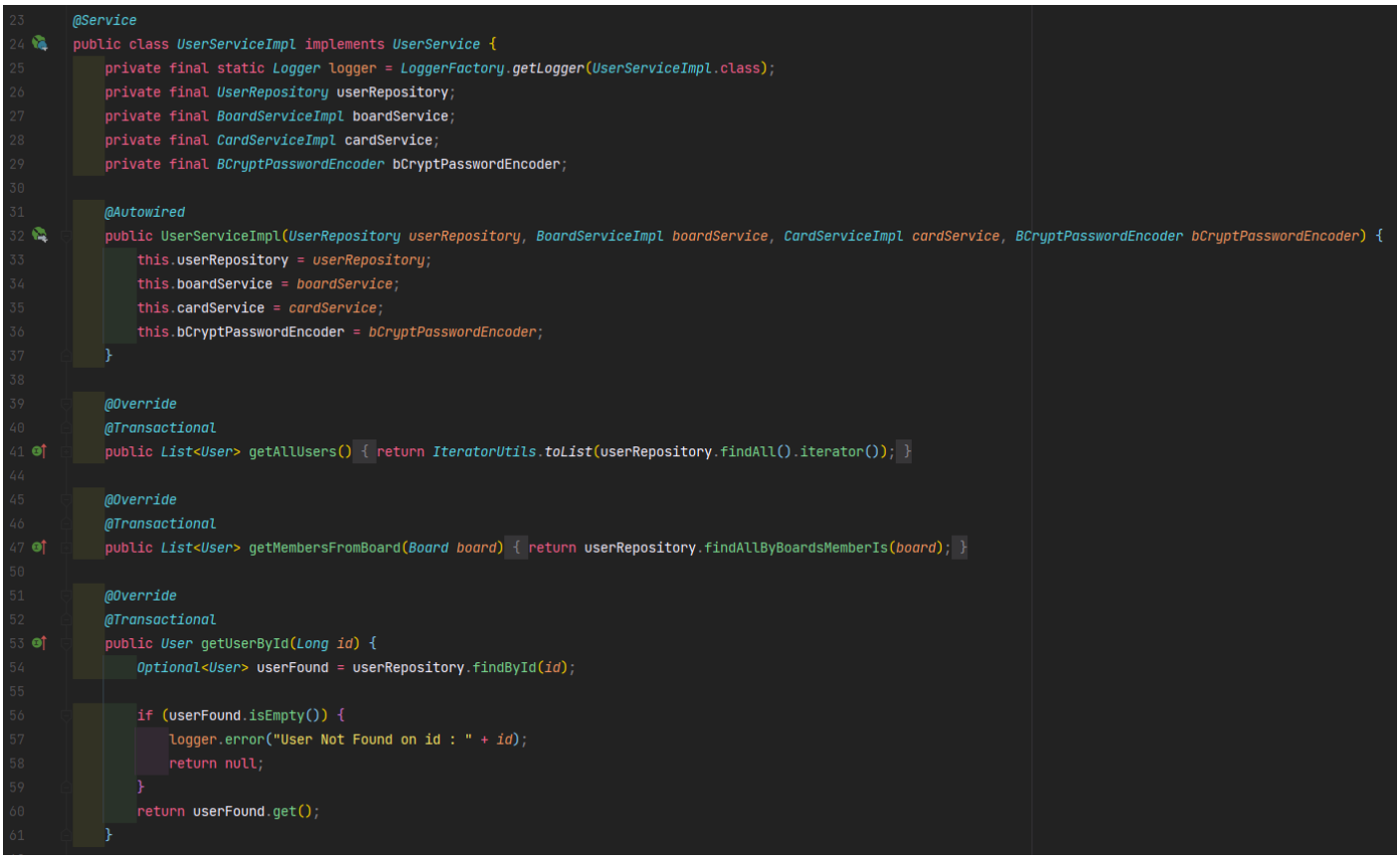
• Ensuite, les repository JPA qui vont être nos interfaces entre nos objets et la base de données, c'est eux qui nous permettront d'effectuer des requêtes.

```java
public interface BoardRepository extends JpaRepository<Board, Long> {
    List<Board> findByOwner(User owner);

    List<Board> findByMembersContaining(User member);

    List<Board> findByOwnerOrMembersContaining(User owner, User member);
}
```

• Enfin, les services qui vont se charger de traiter la donnée avant de l'envoyer en base ou de la récupérer, ce sont les services qui seront utilisés par les contrôleurs Java FX.



```java
@Service
public class UserServiceImpl implements UserService {
    private final static Logger logger = LoggerFactory.getLogger(UserServiceImpl.class);
    private final UserRepository userRepository;
    private final BoardServiceImpl boardService;
    private final CardServiceImpl cardService;
    private final BCryptPasswordEncoder bCryptPasswordEncoder;

    @Autowired
    public UserServiceImpl(UserRepository userRepository, BoardServiceImpl boardService, CardServiceImpl cardService, BCryptPasswordEncoder bCryptPasswordEncoder) {
        this.userRepository = userRepository;
        this.boardService = boardService;
        this.cardService = cardService;
        this.bCryptPasswordEncoder = bCryptPasswordEncoder;
    }

    @Override
    @Transactional
    public List<User> getAllUsers() { return IteratorUtils.toList(userRepository.findAll().iterator()); }

    @Override
    @Transactional
    public List<User> getMembersFromBoard(Board board) { return userRepository.findAllByBoardsMemberIs(board); }

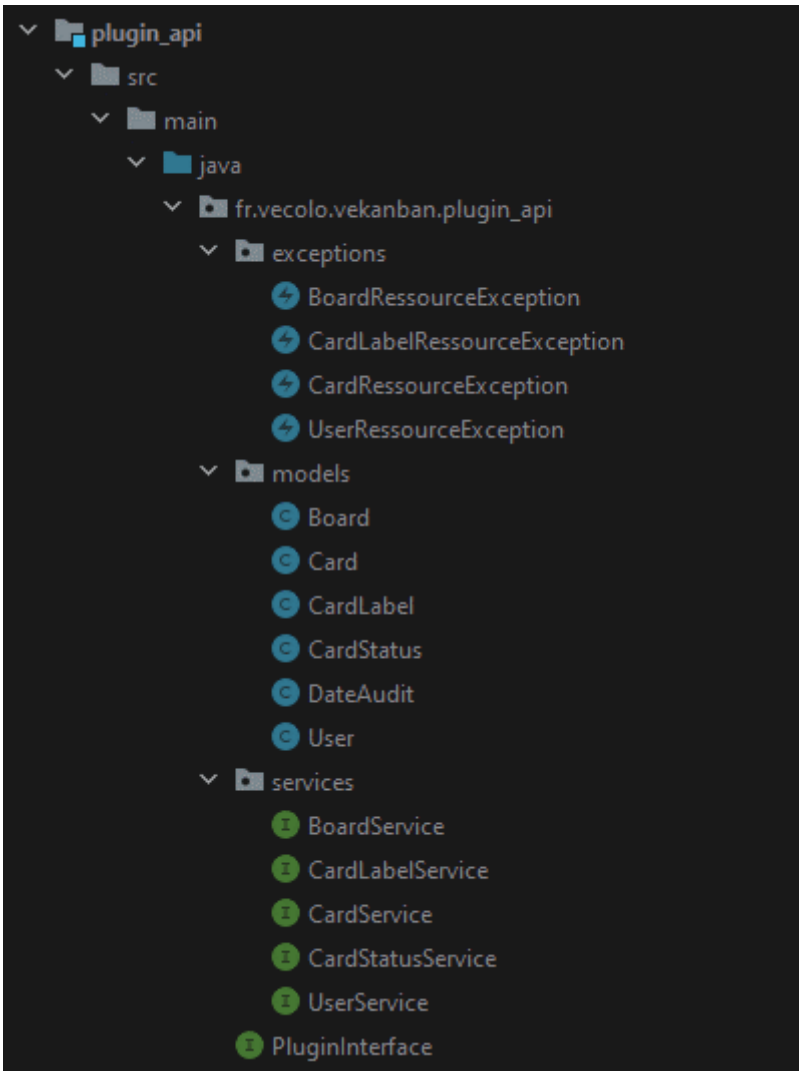
    @Override
    @Transactional
    public User getUserById(Long id) {
        Optional<User> userFound = userRepository.findById(id);

        if (userFound.isEmpty()) {
            logger.error("User Not Found on id : " + id);
            return null;
        }
        return userFound.get();
    }
```

API pour les plugins

Pour permettre le développement des plugins, un package séparé de l'application principale est mis à disposition pour être importé dans un projet Maven.

Ce package contient uniquement la définition des services par des interfaces, l'interface du plug-in devant être rempli, ainsi que les modèles de classes et les exceptions existant dans notre application.

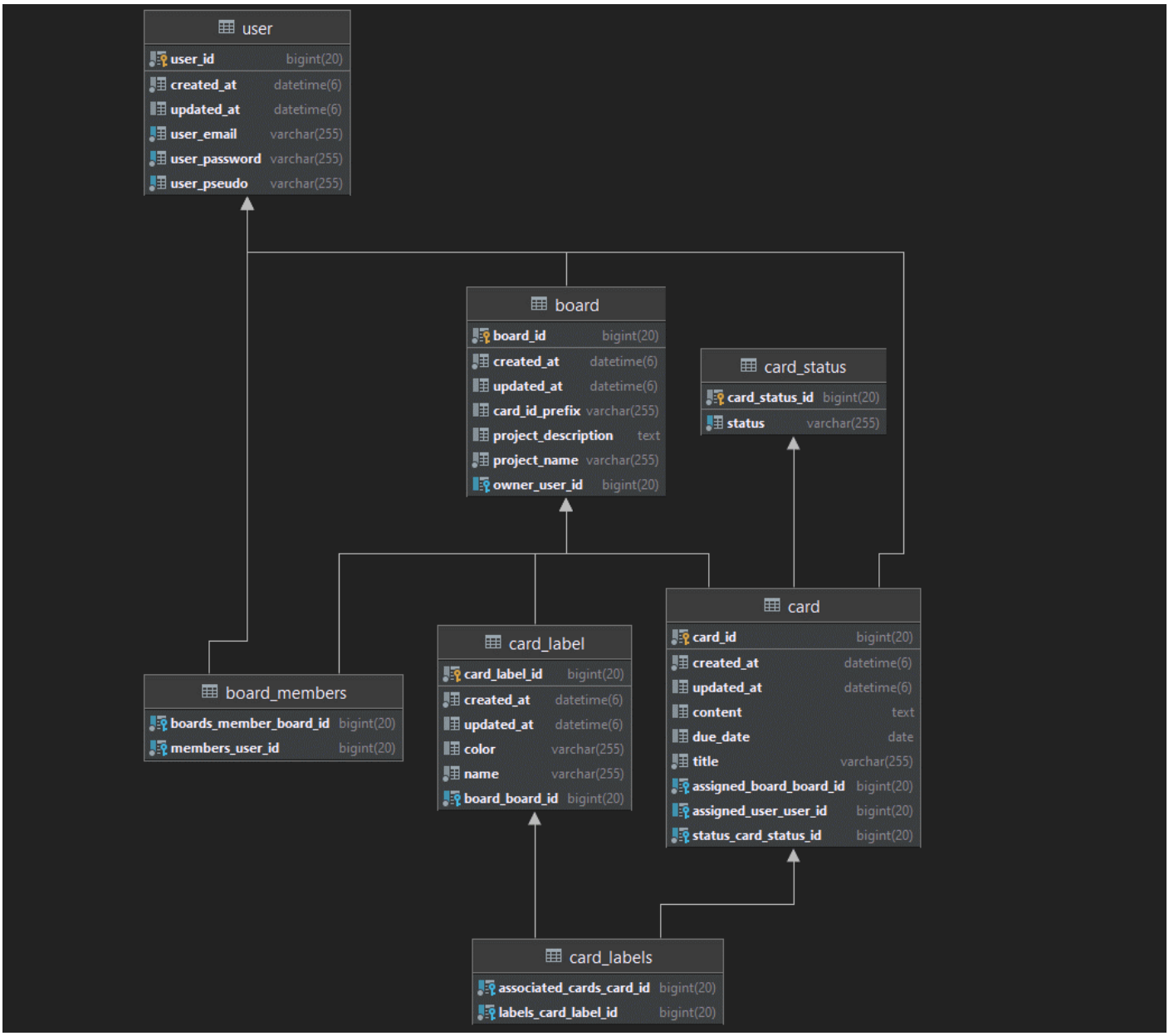


3. Architecture base de données

Pour sauvegarder nos données, nous utilisons une base de données MariaDB hébergé en ligne.

Il est possible via le fichier **application.properties** de changer le driver et de passer sur une base de données PostgreSQL ou même H2 (local).

Voici le schéma de la base de données Vekanban :



V. Bilan du projet

1. Problèmes rencontrés

Fusion de Spring Boot et JavaFX

Spring Boot et Java FX n'ont jamais été conçus pour être utilisés ensemble. Le projet a mis donc énormément de temps à démarrer car il a fallu faire cohabiter ces deux Framework ensembles.

Thème sombre de l’application

Les éléments graphiques de Java est fixe son de base claire. Or nous voulions un thème sombre. Il est possible de personnaliser ses composants à l'aide de balises CSS et nous avons dû passer beaucoup de temps à réécrire le thème CSS composants par composants pour avoir un thème sombre qui nous plaisait (500 lignes de CSS au total).

Gestion des différents contrôleurs JavaFX

Puisque nous avions décidé d'intégrer Spring boot, il fallait que les contrôleurs Java et fixes puisse être auto injectés. Cela a posé plusieurs problèmes au début notamment sur les contrôleurs qui devaient être présents en plusieurs instances (les cartes des taches par exemple) carte de base Spring fonctionne avec des singletons. Nous avons pu cependant résoudre cela en utilisant une annotation supplémentaire sur nos contrôleurs **(@Scope("prototype")**) et tout est rentré dans l'ordre.

Intégration du markdown

Pour intégrer le markdown il fallait déjà pouvoir le parser et reconnaître les différentes balises utilisées. Pour cela nous avons utilisé flexe marque mais même grâce à cela ça restait très difficile d'identifier certaines balises (notamment les bullets, les liens et checkbox qui sont assez similaires). Une fois cela fait, il a fallu rajouter une surcouche CSS sur notre thème sombre déjà présent. Enfin, nous avons rendu l'édition et le rendu du markdown visible en temps réel ce qui a demandé beaucoup de travail supplémentaire pour l'afficher dynamiquement.

Plugins

Nous voulions dès le départ que le plug-in puisse utiliser les différents services permettant de manipuler notre base de données. Pour cela il, fallait que le plug-in, au moment du chargement, puisse récupérer le contexte de Spring afin que l'injection de dépendance puisse se faire. Le projet a subi des changements majeurs à ce moment-là car il a fallu repenser son architecture pour que cela puisse être possible. Le fait également d'intégrer un plugin en soi sans recompiler le code a été tout un défi.

2. Conclusion

Pour conclure, nous sommes très fiers de cette application. Nous avons fait un pari risqué en voulant combiner différentes technologies ensemble et nous sommes parvenus à en faire ce que nous souhaitions. L'injection de dépendance Spring a été d'une énorme aide et a permis beaucoup de simplification dans l'architecture du code. Même si l'intégration plug-in aurait pu être poussé encore plus, nous sommes épatées d'avoir pu injecter à l'intérieur de celui-ci le contexte de Spring afin de pouvoir réutiliser l'injection de dépendance dans ce dernier. Il faut le voir pour le croire mais Spring et Java FX fonctionne plutôt bien ensemble.

Revision #6
Created 2022-09-26 20:54:21 UTC by Noé Larrieu-Lacoste
Updated 2022-09-27 19:00:24 UTC by Noé Larrieu-Lacoste