OnlyQuest

Application de création et gestion de contenu pour un jeu de quêtes

TRAVAIL DE BACHELOR

VICTOR LAMBERT

Mars 2024

Supervisé par :

Prof. Dr. Jacques PASQUIER-ROCHA
Software Engineering Group

UNI
FR
UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Groupe Génie Logiciel
Département d'Informatique
Université de Fribourg (Suisse)

Remerciements

Je souhaite exprimer ma profonde gratitude envers le Prof. Dr. Jacques Pasquier-Rocha pour sa flexibilité, son suivi et ses conseils éclairés durant tout le processus de réalisation de mon travail. Je tiens également à remercier chaleureusement ma famille et mes amis pour leur soutien constant pendant mes études, ainsi qu'à Noëlle pour la relecture et la correction des fautes d'orthographe.

Résumé

L'application que j'ai développée intègre une interface de programmation d'application (API) de type REST construite avec Spring Boot. Cette API permet de créer, stocker et gérer automatiquement divers éléments du jeu dans une base de données MySQL. En utilisant Angular comme cadre de travail (framework) côté client, les utilisateurs administrateurs peuvent créer des catégories de personnages, des compétences et des quêtes. D'un autre côté, les utilisateurs de type joueurs ont la possibilité de créer de nouveaux personnages et, dans une version ultérieure, de lancer des quêtes pour faire évoluer leur personnage. En résumé, l'application offre une plateforme conviviale pour le développement et la gestion des éléments du jeu, favorisant ainsi une expérience immersive pour les administrateurs et les joueurs.

Mot-clé : Spring Boot, API REST, Angular, MySQL, jeu vidéo, gestion des personnages, compétences, quêtes, utilisateur, héros, framework, endpoint, frontend, backend.

Table des matières

1 Introduction **1**
1.1 Motivation et objectifs ... 1
1.2 Structure du rapport ... 1
1.3 Conventions et notations ... 2
2 Fonctionnalités et modélisation **3**
2.1 Jeux de quêtes existants ... 3
2.1.1 Shakes & fidget ... 3
2.1.2 Dofus ... 6
2.2 Fonctionnalités ... 9
2.2.1 Les fonctionnalités observées ... 10
2.2.2 Les fonctionnalités retenues ... 10
2.3 Cas d'utilisation du joueur ... 10
2.4 Cas d'utilisation de l'administrateur ... 11
2.5 Cas d'utilisation partagé ... 12
2.6 Modélisation de la base de données ... 13
3 Application du point de vue utilisateur **15**
3.1 Généralité ... 15
3.2 Pages du client pour le joueur ... 17
3.3 Pages du client pour l'administrateur ... 20
4 Programmation **23**
4.1 Architecture logicielle ... 23
4.1.1 Architecture REST ... 23
4.1.2 Full Stack Spring Boot Angular ... 24
4.2 Frontend ... 25
4.2.1 Angular ... 26
4.2.2 Implémentation de l'interface utilisateur ... 28
4.3 Backend ... 36
4.3.1 Spring boot ... 36
4.3.2 Implémentation REST API ... 37
4.4 Base de données ... 55
5 Conclusion **56**

5.1 Résultats 56
5.2 Améliorations 57
5.3 Perspectives 57
References **58**

Liste des figures

1 Page principale, choix de la classe (Shakes & Fidget) 4
2 Page principale, choix de l'apparence (Shakes & Fidget) 5
3 Page principale, choix du nom (Shakes & Fidget) 5
4 Contenu (Shakes & Fidget)...6
5 Authentification (Dofus)...7
6 Choix du serveur (Dofus)..7
7 Sélectionner un personnage (Dofus)...8
8 Création du personnage (Dofus)...8
9 Modification de l'apparence (Dofus)...9
10 Livre de quêtes (Dofus) 9
11 Use cas utilisateur 11
12 Use case administrateur 12
13 Use case administrateur et utilisateur 12
14 Modèle entité-relation...14
15 Formulaire de connexion 16
16 Formulaire d'enregistrement...16
17 Créer un nouveau héros 17
18 Page d'accueil..18
19 Listes des héros...19
20 Liste des quêtes...19
21 Gestion du contenu (1)..20
22 Gestion du contenu (2)..20
23 Formulaire de création de catégories..21
24 Liste des catégories...21
25 Édition des catégories...22
26 Architecture logicielle...25
27 Initialiseur de projet Spring Boot 37
28 Endpoint (user) 45
29 Schéma de données (user)...46
30 Requête de création d'utilisateur avec Swagger 47

31 Endpoint (hero)..48
32 Endpoint (quest) 48
33 Endpoint (category) 49
34 Endpoint (competence) 49
35 PhpMyAdmin concepteur de table 51
36 Toutes les tables 55
37 Table heroes-categories 55

Liste des codes sources

1 Installer npm 26
2 Installer angular/cli 27
3 Générer un projet Angular 27
4 Services Utilisateur 29
5 Composant de connexion et d'authentification 32
6 Méthodes CRUD du service catégorie 33
7 Composant de création de catégories 34
8 Composant d'affichage et modification de catégorie 36
9 Propriété de l'API 38
10 Classe main 38
11 Dépôts utilisateur 39
12 Dépôts héros 40
13 CORS config. 41
14 Contrôleurs pour utilisateur 44
15 Entité (héros) 53
16 Entité User (code partiel) 53
17 Entité Competence (code partiel) 54

1
Introduction

1.1 Motivation et objectifs 1
1.2 Structure du rapport 1
1.3 Conventions et notations 2

1.1 Motivation et objectifs

Depuis mon plus jeune âge, mon intérêt pour les jeux n'a fait que de s'intensifier. Les jeux vidéo ont fait leur apparition au vingtième siècle, j'ai alors eu la chance de grandir en même temps qu'ils évoluaient. De leurs caractéristiques immersives, les jeux actuels permettent d'échapper momentanément à la réalité. Ils offrent, par exemple, la perspective d'incarner des personnages et de participer à diverses quêtes en explorant l'immensité du jeu et de ses graphismes.

Au fil des années, les progrès technologiques ont changé le domaine de l'industrie des jeux vidéo. Au-delà des consoles traditionnelles, les smartphones, les ordinateurs ou encore les tablettes sont devenus de véritables plateformes de jeu qui proposent à chacun la possibilité de jouer à tout moment et dans tous les endroits imaginables.

C'est dans ce contexte que m'est venue l'idée de réaliser mon propre jeu vidéo. Cependant, la complexité du processus m'a amené à me focaliser uniquement sur l'implémentation d'une interface pour la création et la gestion du contenu d'un tel jeu. Le but de l'application est de simplifier le développement, tout en permettant sa maintenance et son expansion ultérieure.

Pour ce faire, j'ai dans un premier temps choisi de créer une interface de programmation d'application (API). La communication entre le serveur et la base de données relationnelles MySQL se fera avec JPA et Hibernate offert par Spring Boot. Enfin, une interface utilisateur sera également implémentée avec Angular afin d'interagir plus facilement avec cette application.

1.2 Structure du rapport

Chapitre 1 : Introduction

L'introduction expose les motivations et les objectifs du présent travail, tout en dévoilant la structure du rapport. Cette dernière consiste en un synopsis de chaque chapitre qui sera abordé.

Chapitre 2 : Fonctionnalités et modélisation

Ce chapitre propose une analyse des fonctionnalités de deux jeux de rôle afin d'identifier celles qui sont récurrentes. Une fois cette analyse effectuée, une présentation de toutes les fonctionnalités implémentées dans l'application "OnlyQuest" sera proposée. La modélisation de la base de données sera également présentée à l'aide d'un modèle entité-relation.

Chapitre 3 : Application du point de vue utilisateur

Ce troisième chapitre représente réellement les fonctionnalités du point de vue de l'utilisateur, avec différentes figures illustrant les possibilités d'interaction avec l'application.

Chapitre 4 : Programmation

Dans ce chapitre, tous les éléments de développement sont présentés. Le choix des technologies, l'architecture logicielle, ainsi que l'implémentation seront détaillés. Un aperçu de la base de données sera également effectué.

Conclusion

Enfin, la conclusion présente le bilan de ce travail ainsi que plusieurs améliorations possibles de l'application.

1.3 Conventions et notations

- Le français a été choisi pour la rédaction du rapport. Les implémentations de code et certains termes techniques sont en anglais, cependant les commentaires sont en français.
- Ce rapport comporte 5 chapitres, certains d'entre eux sont divisés en sous-chapitre.
- Les abréviations et anglicismes sont définis une première fois, puis sont utilisés directement comme tels.
- Un ton neutre a été choisi pour la rédaction de ce rapport. Le masculin est utilisé, cependant tous les sexes sont concernés.
- Les figures et les codes sources (Listings) sont numérotés par ordre d'apparition dans le rapport.
- Les codes sources se présentent sous la forme suivante :

```
1    public int sum(int x, int y){
2        int res;
3        res = x + y;
4        return res;
5    }
```

- L'intégralité du code source des différents éléments de programmation est accessible sur GitHub [14][15].

2

Fonctionnalités et modélisation

2.1 Jeux de quêtes existants 3
2.1.1 Shakes & fidget 3
2.1.2 Dofus 6
2.2 Fonctionnalités 9
2.2.1 Les fonctionnalités observées 10
2.2.2 Les fonctionnalités retenues 10
2.3 Cas d'utilisation du joueur 10
2.4 Cas d'utilisation de l'administrateur 11
2.5 Modélisation de la base de données 13

2.1 Jeux de quêtes existants

Actuellement, l'univers des jeux vidéo est caractérisé par une diversité remarquable, offrant aux joueurs une multitude d'expériences ludiques. Que ce soit à travers des graphismes stupéfiants, des scénarios immersifs ou des mécaniques de jeu novatrices, la créativité des développeurs repousse constamment les limites de l'industrie. De l'effervescence des jeux multijoueurs en ligne à l'intimité des aventures solo, chaque joueur trouve son terrain de prédilection.

Parmi cette vaste gamme de jeux vidéo, l'analyse se focalisera uniquement sur deux jeux de quête. Cette démarche permettra d'explorer et d'analyser les multiples fonctionnalités que ces applications peuvent offrir afin d'identifier lesquelles sont importantes dans un jeu vidéo.

2.1.1 Shakes & fidget

Shakes & Fidget est un jeu vidéo de rôle (RPG) en ligne et sur mobile développé par Playa Games GmbH, une entreprise allemande existant depuis 2009. Avec une communauté de plus de 5 millions de joueurs et une note de 4,4/5 avec plus de 922 000 avis, il s'affirme comme l'un des jeux les plus populaires en France [17].

À l'origine disponible sur navigateur, le jeu a ensuite étendu sa présence aux plateformes mobiles. Arborant un ton autodérisoire et stéréotypé sur les jeux de type fantasy, Shakes & Fidget présente une esthétique de dessin animé et propose des fonctionnalités telles qu'un

système de guildes, des quêtes divertissantes, une arène joueur contre joueur (JcJ), ainsi qu'un héros personnalisable [33].

La page principale offre deux options. La connexion, qui permet d'accéder à son compte et la création d'un personnage pour les utilisateurs n'ayant pas encore de compte. Celle-ci propose également un aperçu global des différentes classes de personnages parmi lesquelles choisir lors de la création.

Figure 1 - Page principale, choix de la classe (Shakes & Fidget)

Une fois la classe du personnage sélectionnée, il est possible de personnaliser le personnage en choisissant parmi différentes apparences et en décidant du sexe. Pour une sélection rapide, une option par défaut ou aléatoire est également disponible grâce à l'icône en forme de dé. Enfin, il suffit de donner un nom au personnage, ou de générer un nom aléatoire en utilisant la même icône, pour se lancer pleinement dans l'aventure.

Figure 2 - Page principale, choix de l'apparence (Shakes & Fidget)

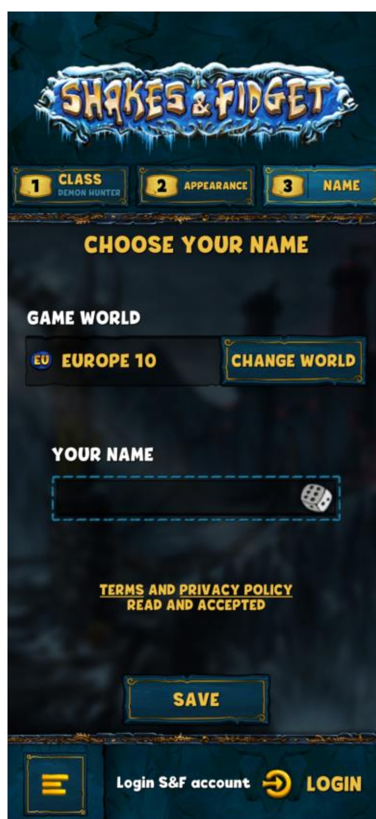

Figure 3 - Page principale, choix du nom (Shakes & Fidget)

Une fois dans le jeu, le menu offre une variété d'options, dont une taverne pour accéder aux quêtes, une arène JcJ, des donjons, et bien d'autres contenus.

Figure 4 - Contenu (Shakes & Fidget)

2.1.2 Dofus

Dofus, un jeu de rôle en ligne massivement multijoueur français, a été développé et édité par Ankama à sa création en 2004, puis par sa filiale Ankama Games. En 2016, Ankama a étendu son univers en lançant une version mobile appelée "Dofus Touch". Bien que basée sur une version antérieure du jeu, cette adaptation a évolué de manière significative avec des mises à jour distinctes, tout en conservant une partie importante du contenu d'origine [16].

Le jeu propose l'exploration d'un univers médiéval fantastique humoristique en quête des puissants œufs de dragon, les Dofus. Il existe 19 classes (13 pour la version mobile) à choisir pour créer un personnage et monter des niveaux en effectuant des quêtes et des donjons ou en affrontant des monstres. Il est également possible d'engager des combats joueurs contre joueurs (JcJ) pour une expérience stratégique captivante. Avec sa multitude de métiers, Dofus offre la possibilité de récolter, créer et améliorer des équipements et des ressources [13].

L'analyse des fonctionnalités du jeu commence avec la page d'accueil illustrée par la Figure 5. Celle-ci offre la possibilité de créer un compte ou de se connecter si l'utilisateur en possède déjà un.

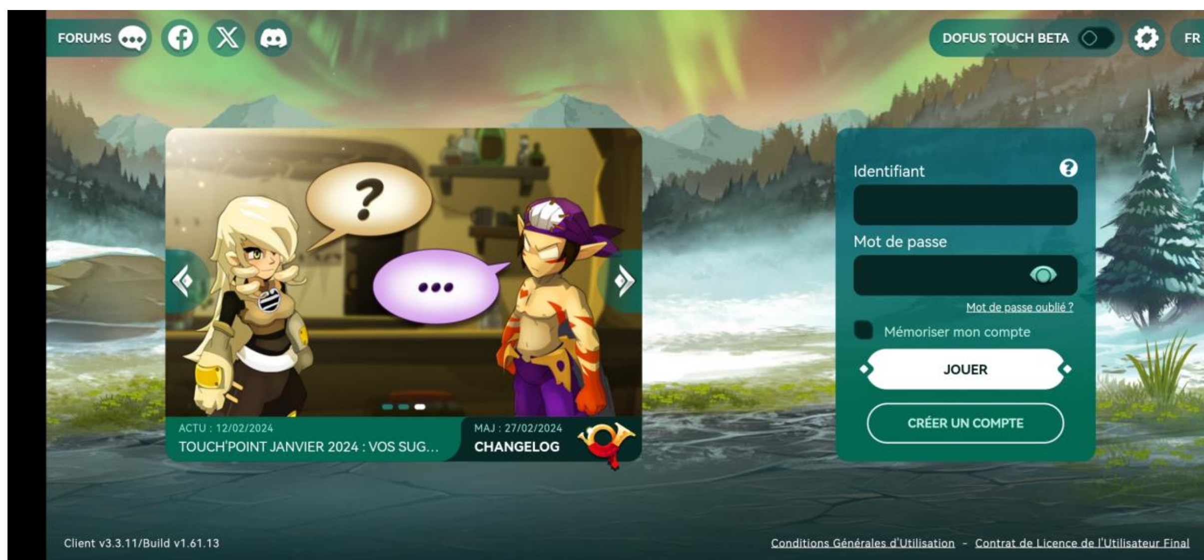

Figure 5 - Authentification (Dofus)

Une fois connecté, le joueur a la possibilité de créer immédiatement un personnage ou de choisir un serveur avant de le faire. Si le joueur possède déjà un ou plusieurs personnages, il peut sélectionner parmi les serveurs où ils se trouvent pour jouer, ou opter pour la création d'un personnage supplémentaire.

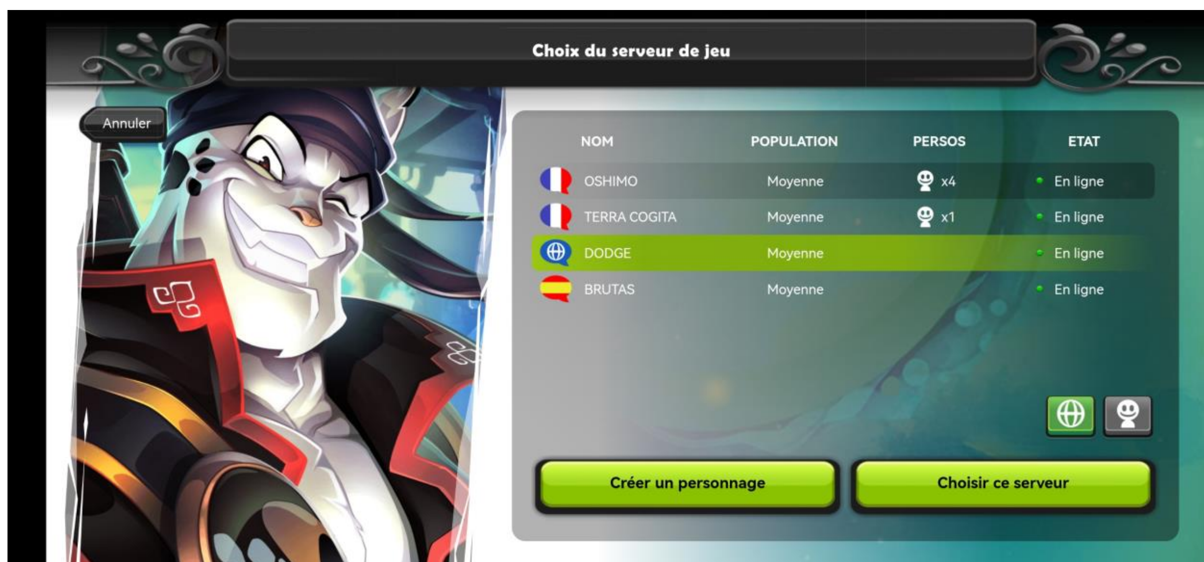

Figure 6 - Choix du serveur (Dofus)

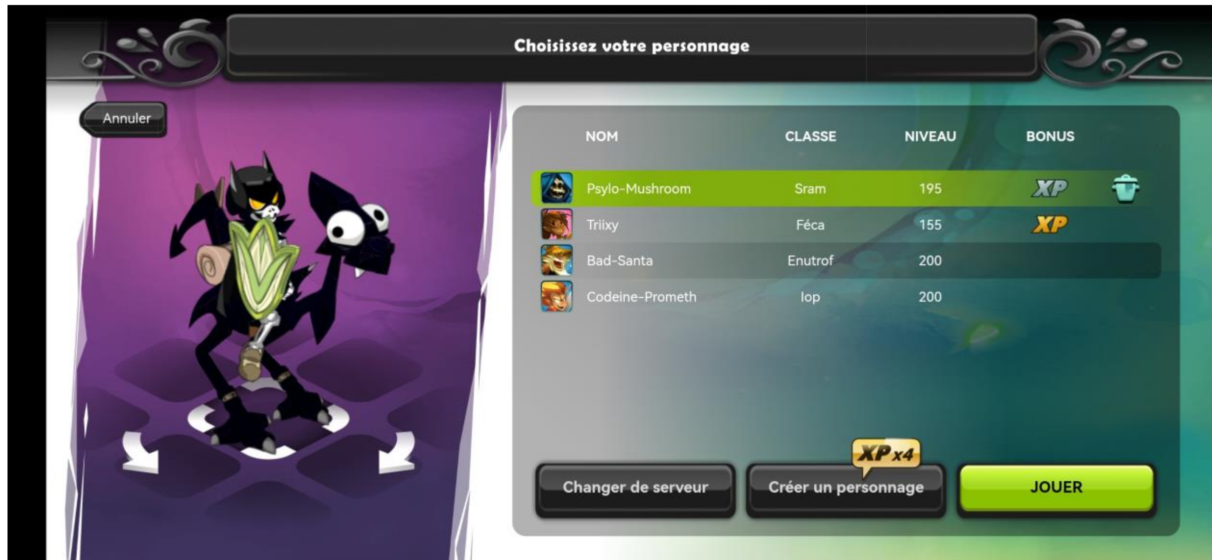

Figure 7 - Sélectionner un personnage (Dofus)

Ensuite, le joueur peut sélectionner la classe, le nom et le sexe du personnage qu'il souhaite jouer. Il a également la possibilité de répondre à trois questions pour recevoir des recommandations sur la classe qui lui conviendrait le mieux.

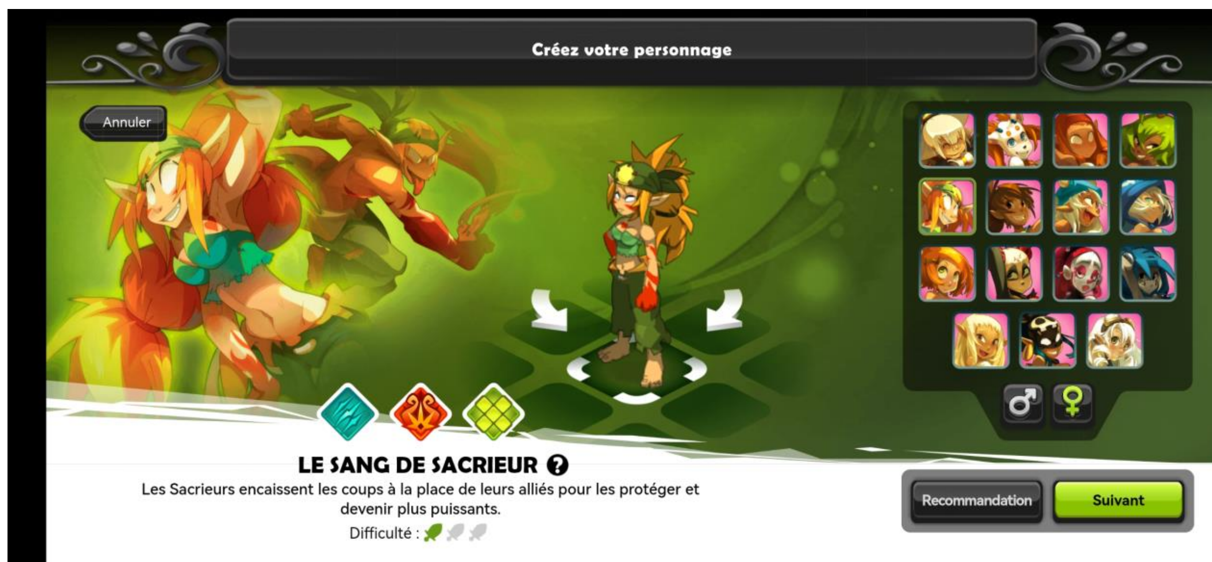

Figure 8 - Création du personnage (Dofus)

Finalement, quelques ajustements peuvent être effectués sur l'apparence, comme le choix des couleurs et du visage du personnage.

Figure 9 - Modification de l'apparence (Dofus)

Une fois dans le jeu, plusieurs éléments sont visibles en bas à droite de l'écran, notamment les caractéristiques, les sorts, la carte, etc. L'accent sera mis uniquement sur le livre de quêtes, qui répertorie toutes les quêtes en cours ainsi que celles déjà accomplies. Cette fonctionnalité permet au joueur de connaître exactement l'avancement de chacune des quêtes qu'il accomplit, facilitant ainsi le déroulement et la progression de celles-ci.

Figure 10 - Livre de quêtes (Dofus)

2.2 Fonctionnalités

Les fonctionnalités d'une application se traduisent par une liste de caractéristiques et de propriétés qu'elle peut offrir. Dans le contexte d'un jeu, elles englobent des éléments tels que la création de personnages, l'exploration de mondes virtuels, la possibilité d'interagir avec d'autres joueurs, et bien d'autres éléments. Ces fonctionnalités jouent un rôle crucial dans la

définition de l'expérience utilisateur en fournissant un contenu riche et de nombreuses possibilités d'interaction.

2.2.1 Les fonctionnalités observées

Les deux jeux précédemment analysés présentent une gamme de fonctionnalités similaires. La première fonctionnalité clé est la création d'un compte utilisateur et l'authentification de l'utilisateur, elle constitue le point de départ du jeu. Une fois connecté, l'utilisateur peut créer un ou plusieurs personnages, modifier l'apparence, choisir parmi différentes catégories et attribuer un nom personnalisé. La déconnexion et reconnexion est possible à tout moment pour arrêter ou poursuivre l'expérience de jeu. D'autres éléments sont également proposés, par exemple des donjons, des métiers ou encore des quêtes. Cette diversité de contenu offre au joueur l'opportunité d'évoluer de manière significative dans le jeu. L'objectif global est de permettre aux joueurs de progresser en augmentant le niveau de leur personnage pour augmenter sa puissance et améliorer ses caractéristiques. Cette mécanique de progression offre une expérience immersive et gratifiante, incitant les joueurs à explorer toutes les facettes proposées par ces jeux.

2.2.2 Les fonctionnalités retenues

Dans l'application "OnlyQuest", l'inspiration provient de certaines de ces fonctionnalités jugées essentielles pour un jeu vidéo. De fait, ladite application offre la possibilité de créer un compte et de s'authentifier pour accéder au contenu interne. De plus, le joueur peut créer plusieurs personnages en choisissant un nom, une catégorie, et une compétence. Toutefois, en raison de l'absence d'un aspect graphique complexe, la modification de l'apparence du personnage n'est pas encore envisageable. Cependant, la liste de tous les personnages, désignés comme des "Héros", peut être consultée dans une page dédiée. Il existe également un journal affichant toutes les quêtes possibles à accomplir. Au-delà de ces fonctionnalités, un espace spécifique a été créé pour les administrateurs qui développeront le jeu. Une section dédiée à la gestion des catégories, des quêtes, et des compétences a été ajoutée pour faciliter la gestion de contenu.

L'exploration des différents cas d'utilisation sera effectuée du côté d'un utilisateur ordinaire, le joueur, mais également du point de vue d'un administrateur impliqué dans le développement du jeu. Cela permettra de visualiser de manière exhaustive les fonctionnalités offertes par l'application.

2.3 Cas d'utilisation du joueur

L'utilisateur, en tant que joueur, manifeste l'intention de découvrir l'application à travers le jeu. Son objectif principal est de se divertir et de passer du bon temps en explorant les diverses fonctionnalités.

Gestion des héros

La gestion des personnages permet au joueur de créer un héros en lui attribuant un nom, en choisissant sa catégorie (mage, guerrier, archer) et en sélectionnant une compétence initiale. Ensuite, le joueur peut visualiser ses héros en accédant à la liste de ses héros, depuis laquelle la suppression d'un ou plusieurs personnages est possible.

Visualisation des quêtes

Enfin, l'utilisateur a accès à la liste des quêtes existantes. Les détails et le déroulement des quêtes ne seront pas abordés dans ce travail, de ce fait aucune autre fonctionnalité n'est actuellement disponible à ce sujet.

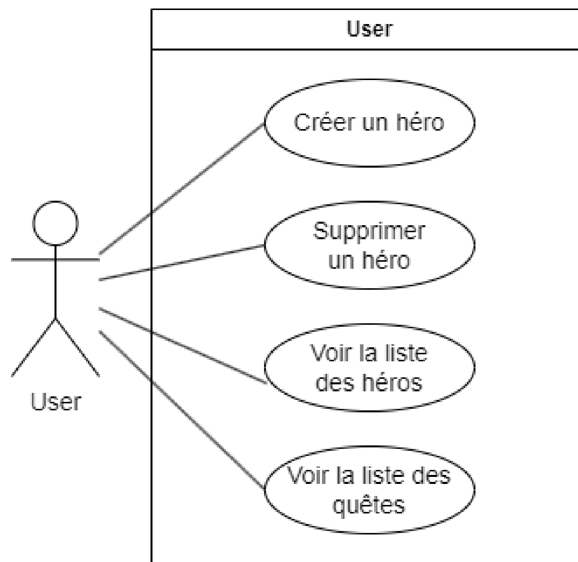

Figure 11 - Use cas utilisateur

2.4 Cas d'utilisation de l'administrateur

Un administrateur possède une autorisation élevée. Il peut alors accéder à l'intégralité des fonctionnalités de l'application qui restent cachées pour un utilisateur standard. Celles-ci visent à simplifier la création, la mise à jour, et l'extension du contenu du jeu.

Gestion des catégories

Les catégories définissent les caractéristiques (points de vie, dommage et énergie) initiales des héros et déterminent quelles compétences peuvent être choisies. La gestion des catégories propose deux fonctionnalités. La première permet de créer une nouvelle catégorie de héros en choisissant un nom, une description, et en initialisant les caractéristiques de base du héros. La seconde autorise la modification de l'un ou plusieurs de ces éléments. Si un administrateur estime qu'une catégorie n'est plus pertinente, il peut également décider de la supprimer depuis la liste, à condition qu'elle ne soit pas encore attribuée à un héros.

Gestion des compétences

La gestion des compétences est simple. Il est possible de créer une nouvelle compétence en lui attribuant un nom, une description et en précisant la catégorie de héros pouvant l'utiliser. Il est également envisageable de mettre à jour les différents champs en modifiant leurs valeurs. La liste des compétences permet de les voir et les supprimer à condition qu'aucun héros ne la possède déjà.

Gestion des quêtes

Enfin, l'administrateur peut gérer les quêtes, comme le mécanisme des quêtes n'est pas développé dans ce travail, la gestion se limite à la création, la modification, la visualisation et

la suppression de quêtes. Pour la création, l'utilisateur doit fournir un nom, une description, et la quantité de points d'expérience que fournit la quête. De plus, il faut choisir les conditions nécessaires pour effectuer la quête, ce qui regroupe le niveau, le nombre de points de vie, les dégâts et l'énergie. Tous ces champs sont modifiables depuis la liste des quêtes, et il est possible de supprimer une quête si aucun héros n'est en train de l'effectuer.

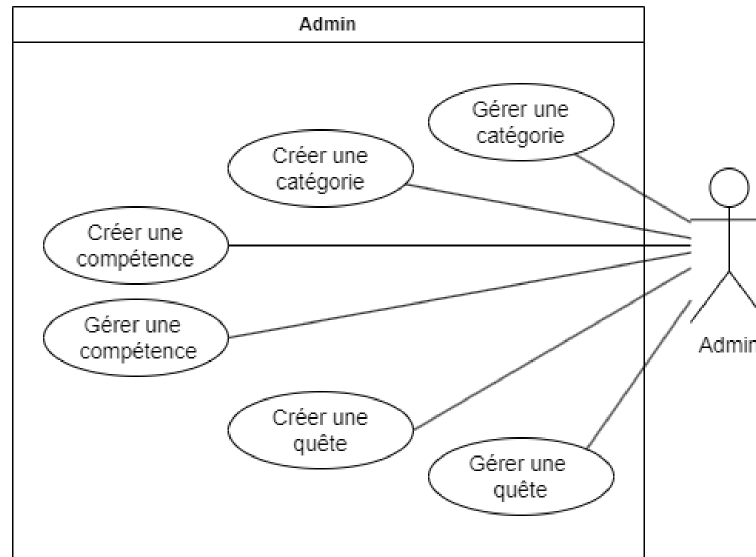

Figure 12 - Use case administrateur

Sur la Figure 12, "gérer" implique la suppression, la modification et la visualisation.

2.5 Cas d'utilisation partagé

Authentification et enregistrement

Le joueur et l'administrateur ont la possibilité de créer un compte ou de se connecter pour accéder à leur propre page utilisateur. L'authentification nécessite uniquement un nom d'utilisateur et un mot de passe, tandis que l'enregistrement implique également l'ajout d'une adresse email. Celle-ci pourrait servir à récupérer les informations d'identification et effectuer une potentielle vérification d'enregistrement.

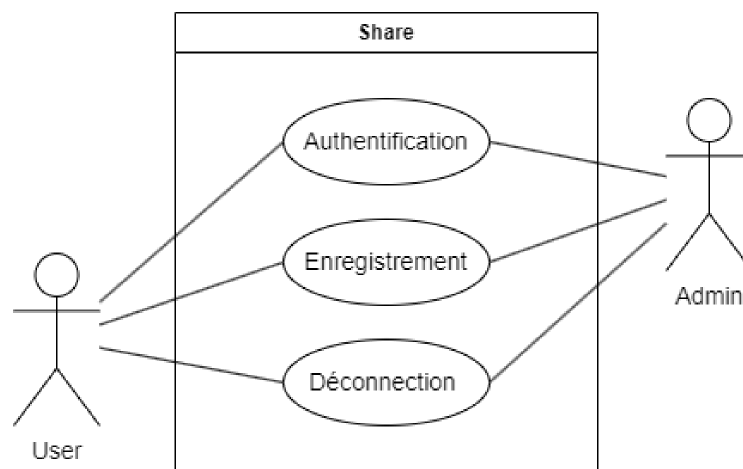

Figure 13 - Use case administrateur et utilisateur

2.6 Modélisation de la base de données

La modélisation des données est un travail important à effectuer avant d'implémenter une application. En effet, elle donne une vision globale sur l'ensemble des données et permet d'identifier les besoins métiers de l'application. Un modèle permet également de structurer de manière claire et logique les données en définissant les entités, leurs attributs et leurs relations. Ce processus fournit un plan bien défini qui facilite le développement logiciel et permet ainsi d'éviter des erreurs potentielles liées à l'ajout d'élément à postériori.

Dans le cadre de ce projet, un modèle relationnel a été utilisé. Il représente la base de données au moyen d'un diagramme Entité-Relation avec une notation de type "Modified Chen Notation" (notation MC) pour exprimer les cardinalités.

Avant de comprendre le schéma, il faut en définir la syntaxe et la sémantique afin de mieux l'interpréter. Du point de vue sémantique, les rectangles représentent des entités, c'est-à-dire des objets du monde réel, tels que des personnes, des lieux ou des événements. Les losanges sont eux utilisés pour symboliser les relations entre ces entités, décrivant les liens conceptuels entre elles. Les traits qui relient les entités aux relations indiquent des associations, décrivant comment les entités sont connectées les unes aux autres. La cardinalité est représentée par des symboles, généralement "1","c", "m" et "mc". Ces symboles sont placés aux extrémités des relations et indiquent le nombre minimum et maximum de fois qu'une entité peut être liée à une autre dans la relation [25] :

- Type 1 : "exactement un"
- Type c : "aucun ou un"
- Type m : "un ou plusieurs"
- Type mc : "aucun, un ou plusieurs"

Pour saisir la syntaxe, la Figure 14 illustre toutes les entités et les relations ce qui permet d'analyser plus facilement la construction grammaticale du schéma :

- Un utilisateur possède zéro, un ou plusieurs héros ; un héros est possédé par exactement un utilisateur.
- Un héros possède une ou plusieurs compétences ; une compétence peut être possédée par aucun, un ou plusieurs héros.
- Un héros appartient à exactement une catégorie ; une catégorie peut être attribuée à aucun, un ou plusieurs héros.
- Un héros n'effectue aucune, une ou plusieurs quêtes ; une quête est effectuée par aucun, un ou plusieurs héros.
- Une catégorie offre une ou plusieurs compétences ; une compétence est offerte par exactement une catégorie.

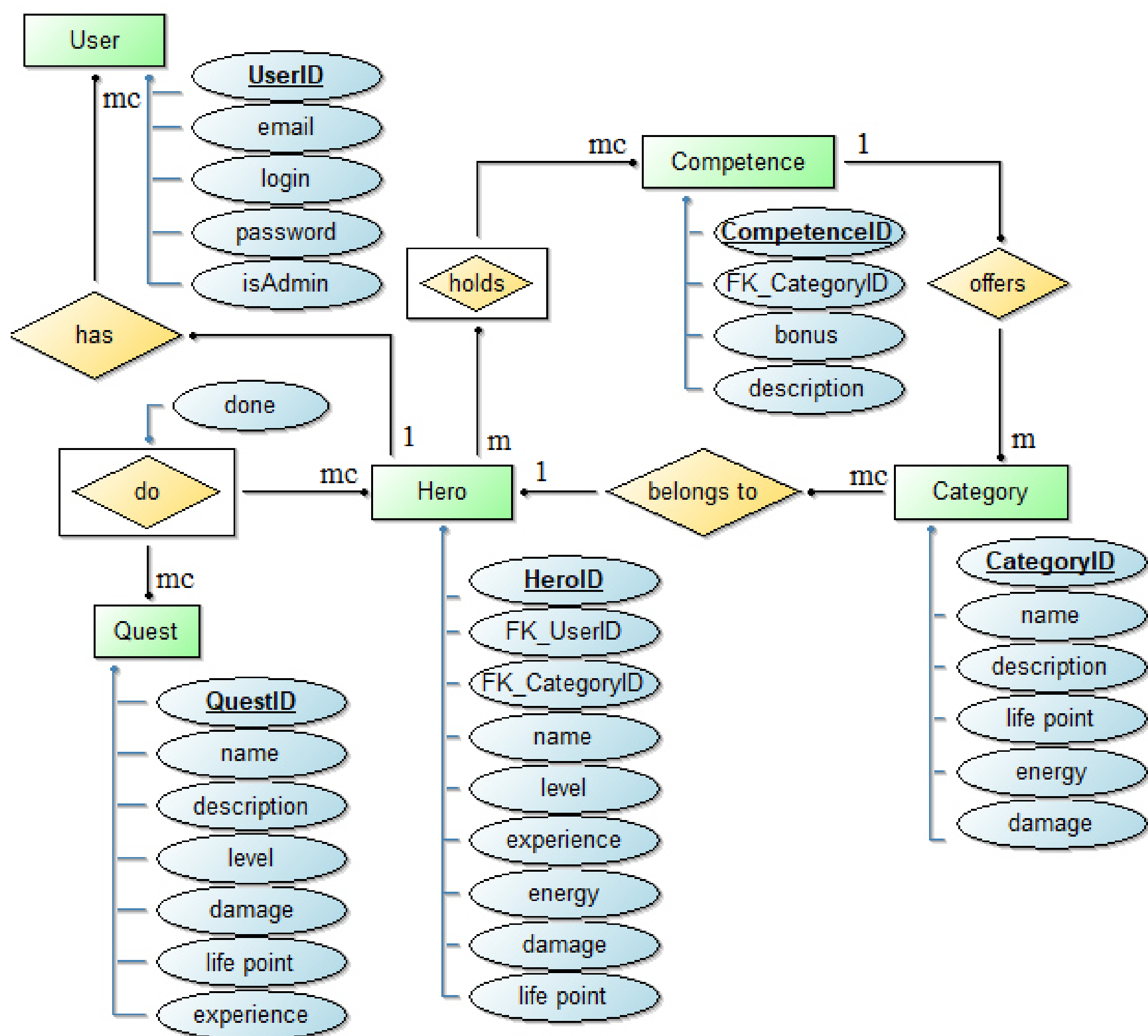

Figure 14 - Modèle entité-relation

3

Application du point de vue utilisateur

3.1 Généralité 15
3.2 Pages du client pour le joueur 17
3.3 Pages du client pour l'administrateur 20

3.1 Généralité

Pour refléter les différentes fonctionnalités, le point de vue d'un utilisateur sera adopté afin d'illustrer concrètement les cas d'utilisation par le biais du client. En informatique, le terme "client" est utilisé pour désigner une interface utilisateur, soit une application logicielle facilitant la communication avec les serveurs.

Le terminal qui peut être un ordinateur, un smartphone ou d'autres appareils, exécute l'application visible par l'utilisateur. Le client informatique utilise les différents services mis à disposition par le serveur en transmettant des requêtes de type HyperText Transfer Protocol (HTTP). Sa responsabilité consiste à préparer la réponse du serveur de manière à ce qu'elle soit correctement affichée sur le terminal demandeur.

Par exemple, un navigateur Web agit en tant que client informatique en envoyant des requêtes au serveur lorsqu'un utilisateur visite un site Internet. Le navigateur affiche ensuite la réponse envoyée par le serveur au format HTML et CSS dans une fenêtre. C'est le principe d'une architecture client-serveur, qui sera abordée plus en détail par la suite [30].

L'interface utilisateur de l'application a comme page principale le formulaire de connexion et d'enregistrement de l'utilisateur représenté sur la Figure 15 et la Figure 16. Elle constitue le point d'entrée pour le reste du contenu offert par le jeu. En effet, pour accéder à l'intégralité des fonctionnalités, l'utilisateur doit posséder un compte. La page est séparée en deux sections distinctes : la première est destinée aux utilisateurs déjà enregistrés, leur permettant de se connecter. La seconde section, accessible par l'onglet "Register" dans la barre de navigation, offre la possibilité de créer un nouveau compte.

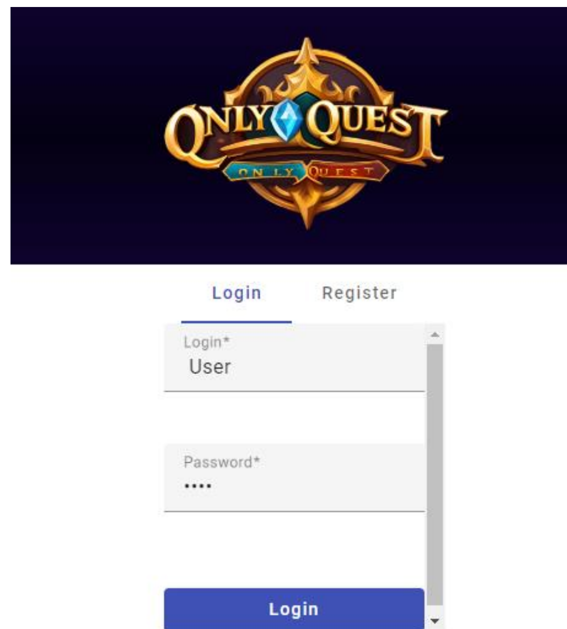

Figure 15 - Formulaire de connexion

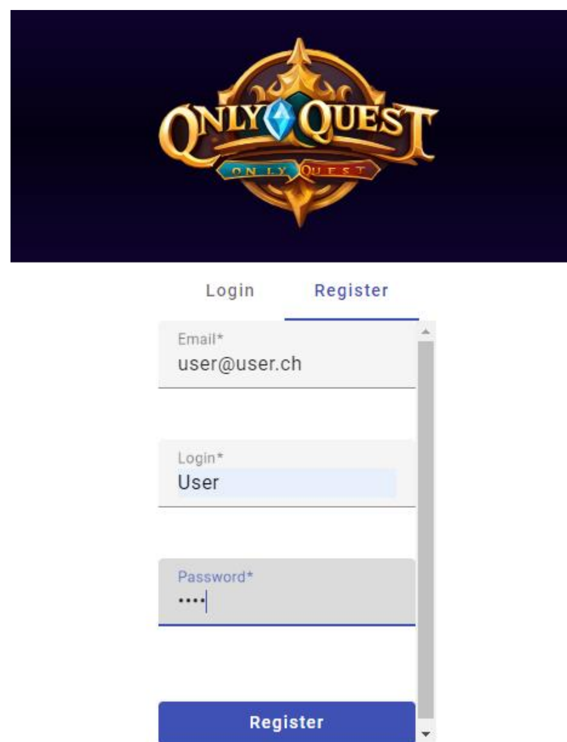

Figure 16 - Formulaire d'enregistrement

3.2 Pages du client pour le joueur

Lorsqu'un joueur se connecte, il est dirigé vers la page d'accueil du jeu illustré par la Figure 18. Celle-ci offre une vue d'ensemble des éléments du jeu, en effet, elle regroupe la liste des différentes catégories et compétences disponibles. Cette page revêt une importance cruciale dans le jeu, car elle permet aux joueurs de découvrir les détails concernant les personnages et aide à prendre une décision sur le choix de leurs héros. La page principale comprend également une barre de menu, regroupant divers éléments. À l'extrême gauche, un menu de navigation latéral est composé de trois icônes. La première icône permet à l'utilisateur de naviguer vers la page de création d'un héros, la deuxième offre la possibilité de visualiser tous les héros possédés par l'utilisateur, le cas échéant, et la dernière permet de sélectionner un héros pour l'envoyer en quête afin d'acquérir de l'expérience. À côté de ce menu se trouvent deux flèches, permettant de revenir à la page précédente ou de passer à la suivante. Sur la droite, la barre de menu est constituée de trois icônes. La première permet simplement de retourner à la page d'accueil, la deuxième offre la possibilité de modifier ses identifiants, et la troisième permet de se déconnecter.

Comme mentionné précédemment, chaque icône propose une fonctionnalité distincte. Par exemple, l'icône "New hero" de la Figure 18 donne accès à un formulaire composé de trois champs. Lorsqu'un utilisateur désire créer un nouveau héros, il doit lui donner un nom puis sélectionner une catégorie et une compétence initiale. Une fois tous les champs remplis, le bouton "Create Hero" devient vert, et il est possible de soumettre le formulaire pour concrétiser la création du personnage (voir Figure 17).

Figure 17 - Créer un nouveau héros

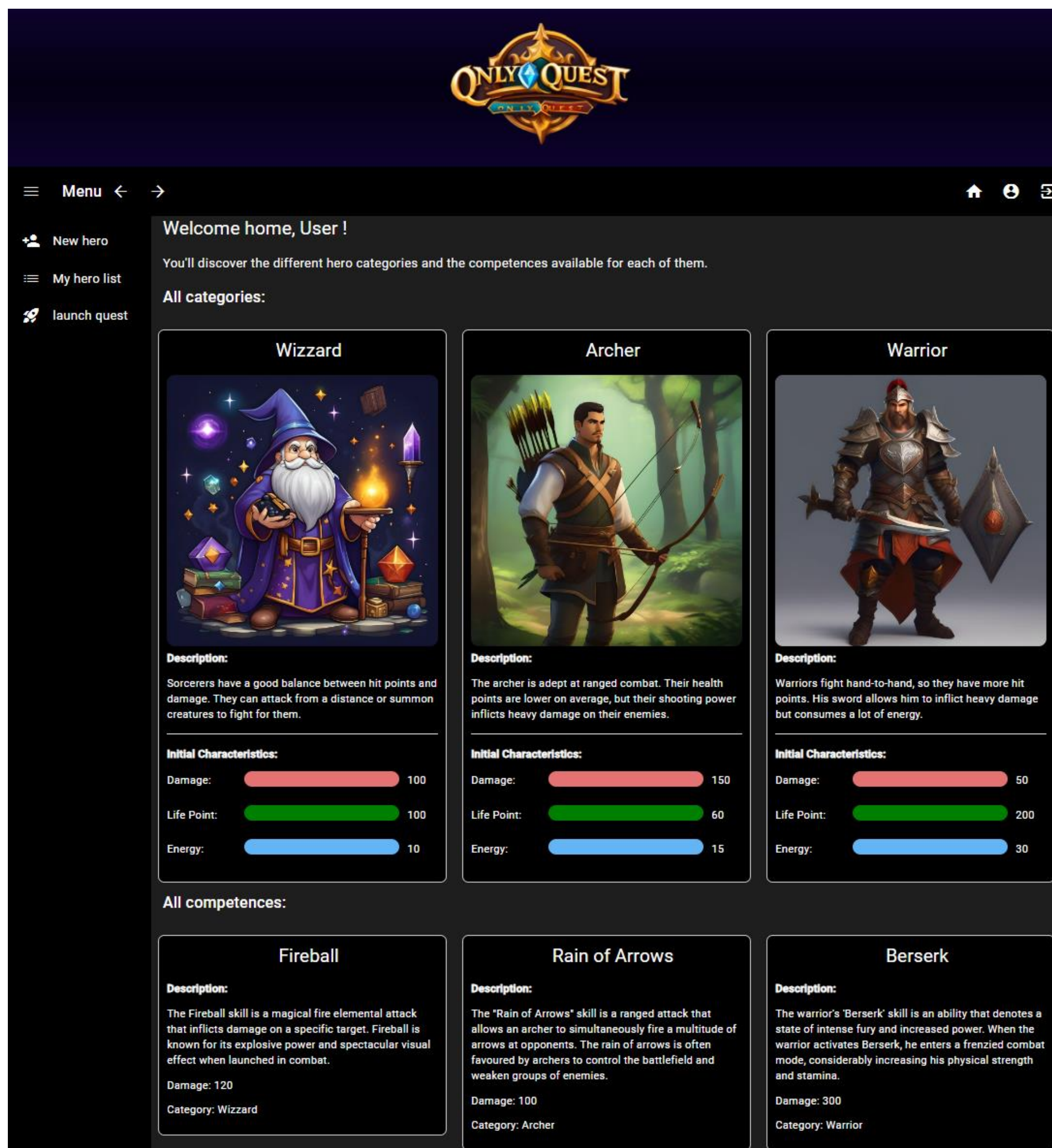

Figure 18 - Page d'accueil

Une fois que le formulaire de création de héros a été soumis, le personnage est ajouté à la liste des héros de l'utilisateur. Pour consulter cette liste, il suffit d'aller sur la page dédiée qui est accessible par l'icône intitulée "My hero list" de la Figure 18. Cette page présente chaque héros sous la forme de cartes, affichant le nom du personnage, son niveau, son expérience, ainsi que l'ensemble de ses caractéristiques. À partir de cette liste, l'utilisateur a également la possibilité de supprimer ses héros grâce au bouton "remove".

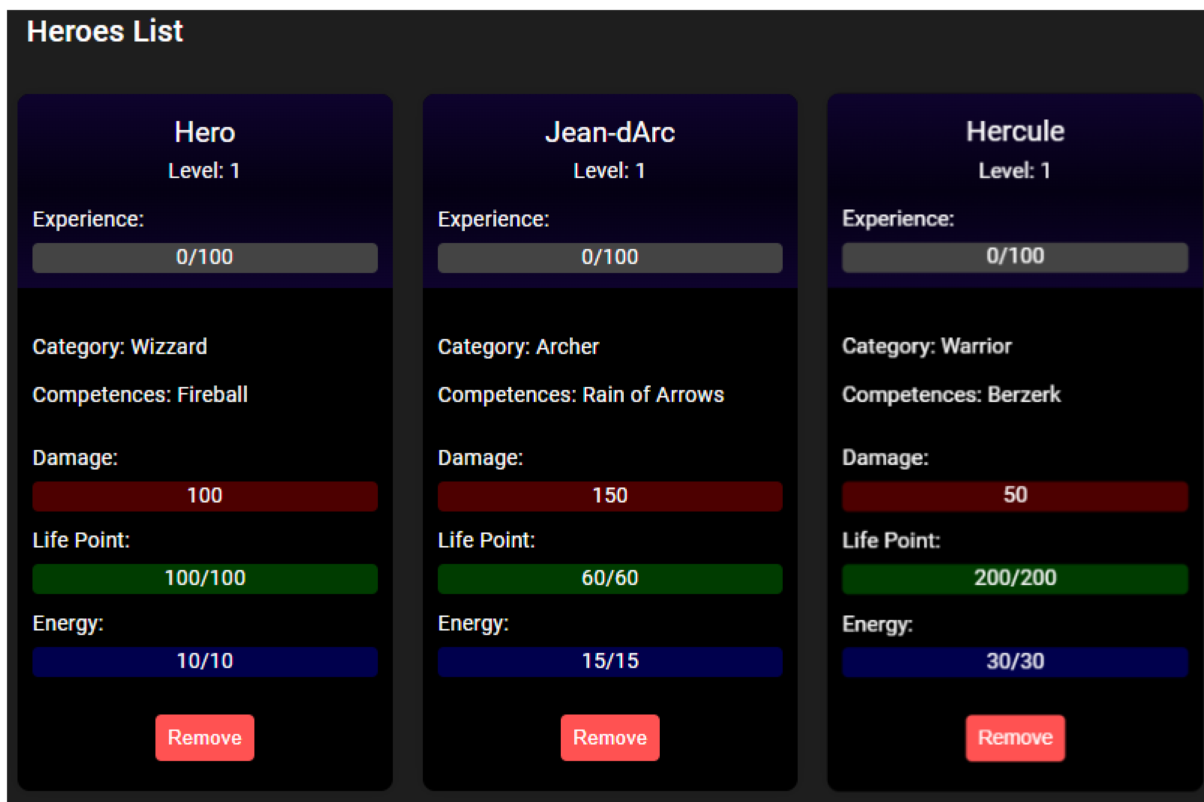

Figure 19 - Listes des héros

À ce stade, le joueur qui possède un ou plusieurs héros aura la possibilité de lancer des quêtes. Cependant, comme mentionné précédemment, le mécanisme des quêtes n'est pas abordé dans ce travail. Pour l'instant, la page partiellement illustrée dans la Figure 20 se contente de présenter la liste des quêtes et les conditions requises pour y participer.

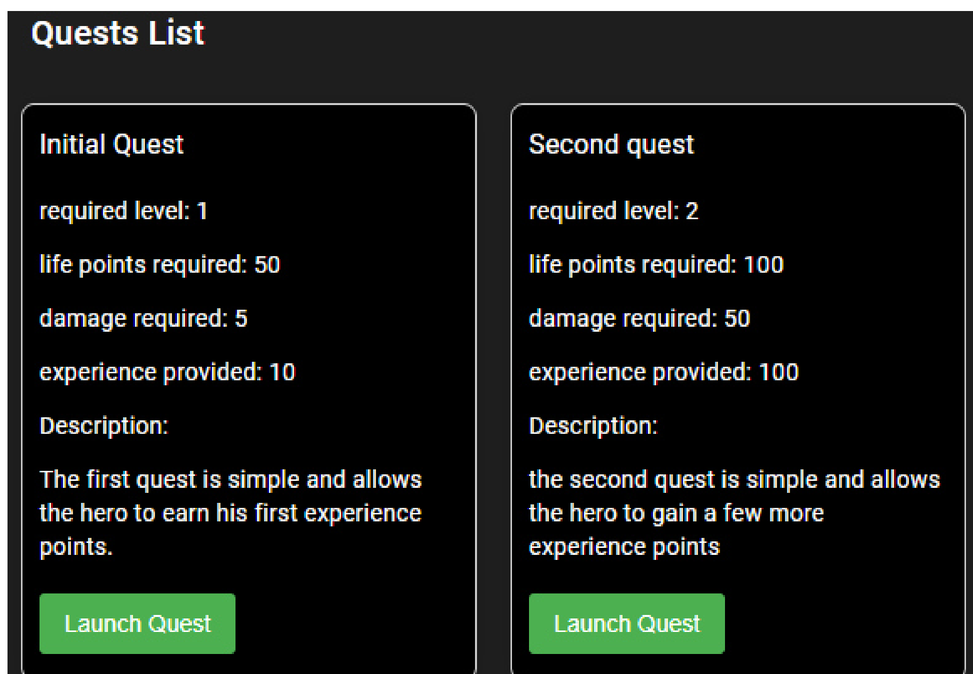

Figure 20 - Liste des quêtes

3.3 Pages du client pour l'administrateur

L'administrateur a accès au même contenu que le joueur et peut également participer à l'aventure en tant qu'utilisateur classique. Cependant, il bénéficie d'une icône supplémentaire dans sa barre de menu (voir Figure 21), qui n'est visible que si l'utilisateur est de type administrateur. En cliquant sur cette icône, un sous-menu s'affiche, proposant trois rubriques qui permettent d'accéder aux différentes pages. Grâce à ces options, l'administrateur a la possibilité de participer à la gestion des catégories, des compétences et des quêtes.

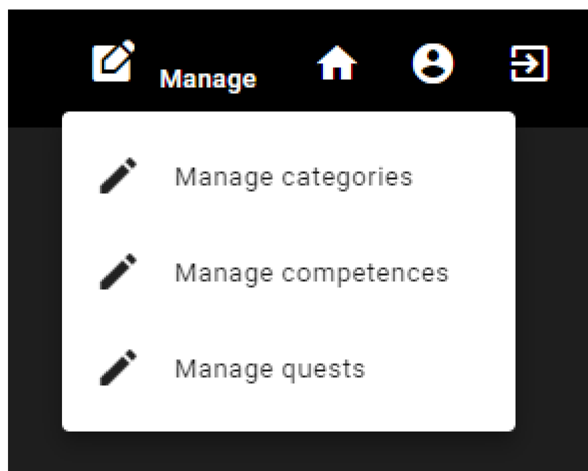

Figure 21 - Gestion du contenu (1)

Les trois pages accessibles par le sous-menu sont en tout point semblables, elles comportent un menu de navigation latéral qui permet d'accéder aux pages de création et de modification des différents éléments. La Figure 22 illustre uniquement les menus de navigations des trois pages.

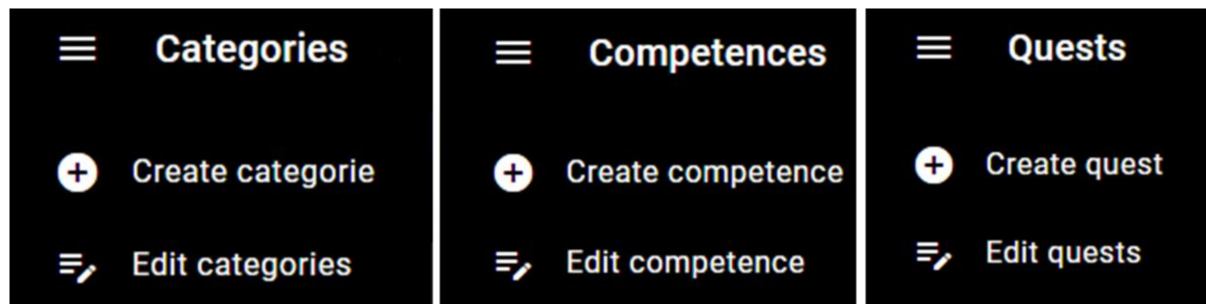

Figure 22 - Gestion du contenu (2)

La première section du sous-menu concerne la gestion des catégories. Pour créer une nouvelle catégorie, il est nécessaire de renseigner tous les champs du formulaire, ce qui comprend le nom, la description, ainsi que les caractéristiques de base de la classe (les points de vie, les dommages et l'énergie). Une fois le formulaire soumis, la catégorie est créée et apparaîtra à la fois sur la page d'accueil du joueur et dans la page dédiée à la modification des catégories accessibles par les administrateurs.

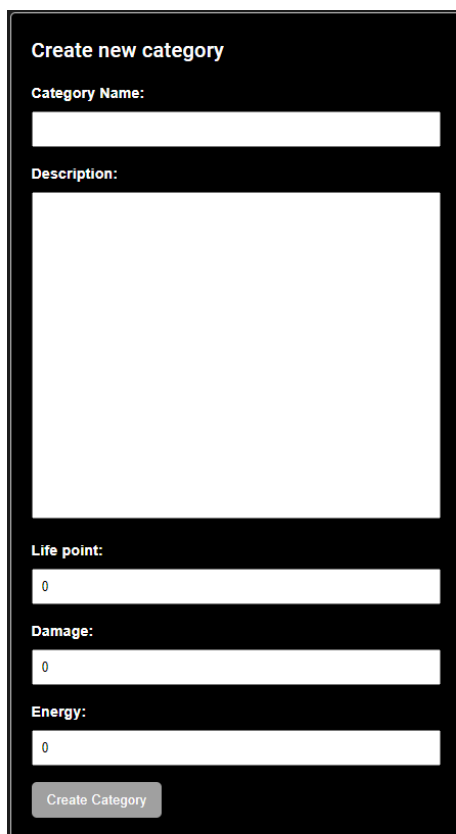

Figure 23 - Formulaire de création de catégories

Cette page présente la liste complète de toutes les catégories existantes sous forme de cartes avec leurs caractéristiques. Deux options s'offrent alors à l'administrateur : la modification et la suppression d'une catégorie existante.

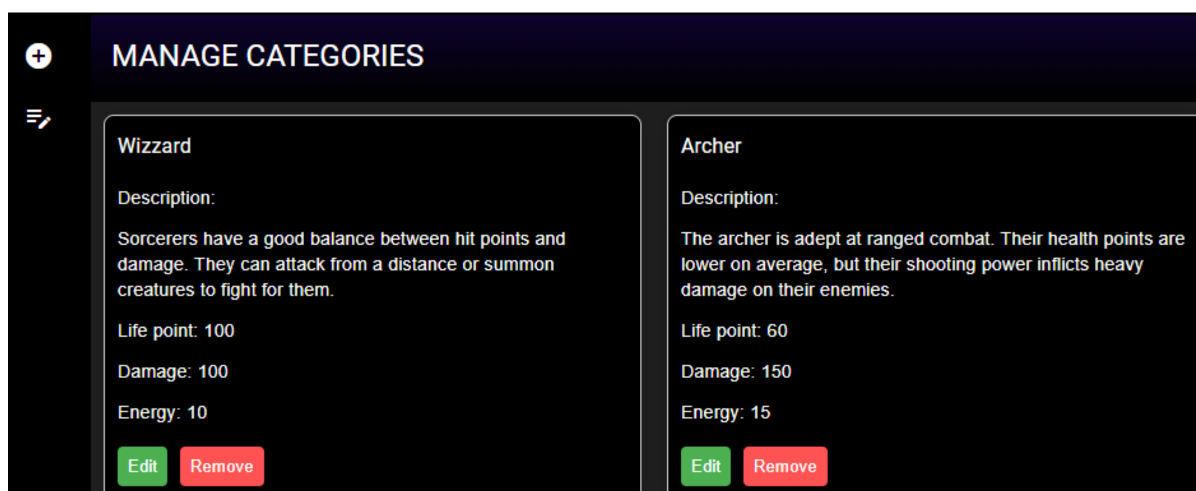

Figure 24 - Liste des catégories

Le bouton "Edit" est conçu pour ajuster une catégorie. Lors de la modification d'une catégorie, des champs apparaissent sous la carte, offrant la possibilité d'apporter des changements partiels ou totaux à la catégorie. Il est envisageable d'annuler les modifications en utilisant le bouton "Cancel" ou de les sauvegarder en appuyant sur le bouton "Save".

Figure 25 - Édition des catégories

Il est également possible de créer et de modifier les compétences ainsi que les quêtes en accédant aux pages correspondantes. Le principe demeure le même, la création se fait à chaque fois à travers un formulaire, et il est envisageable de modifier chacun des éléments de la même manière que pour les catégories.

4 Programmation

4.1 Architecture logicielle 23
4.1.1 Architecture REST 23
4.1.2 Full Stack Spring Boot Angular...24
4.2 Frontend 25
4.2.1 Angular 26
4.2.2 Implémentation de l'interface utilisateur...28
4.3 Backend 36
4.3.1 Spring boot..36
4.3.2 Implémentation REST API 37
4.4 Base de données 55

4.1 Architecture logicielle

L'architecture logicielle est un concept primordial du développement informatique. En effet, elle offre une représentation symbolique et schématique de la structure organisationnelle d'un système logiciel en fournissant des lignes directrices pour la mise en œuvre et la maintenance du logiciel, détaillant les éléments, leurs interrelations et leurs interactions. Contrairement aux spécifications de l'analyse fonctionnelle qui décrivent ce que le système doit réaliser, le modèle d'architecture, élaboré lors de la phase de conception, se concentre sur la manière dont le système doit être conçu pour répondre aux spécifications. En d'autres termes, alors que l'analyse stipule ce qui doit être fait, l'architecture logicielle se concentre sur la manière dont cela doit être réalisé [5].

4.1.1 Architecture REST

En 2000, Roy Fielding a formulé l'architecture REpresentational State Transfer (REST) dans le cadre de sa thèse de doctorat intitulée "Architectural Styles and the Design of Network-based Software Architectures" à l'université de Californie à Irvine [48]. REST est un ensemble de contraintes architecturales plutôt qu'un protocole ou une norme spécifique. Le terme "RESTful" est utilisé pour décrire une implémentation ou un service qui se conforme aux principes de REST. Ce modèle convient parfaitement à la création d'une interface de programmation

d'application. Une API est un ensemble de protocoles standardisés qui permettent à plusieurs applications informatiques de communiquer entre elles.

Dans le cadre d'une API RESTful, lorsqu'un client émet une requête, celle-ci transfère une représentation de l'état de la ressource au demandeur ou au point de terminaison (endpoints). Ce dernier se réfère à une URL (Uniform Resource Locator) spécifique ou à une ressource particulière que le client peut utiliser pour interagir avec le service. Cette représentation est transmise par le protocole HTTP dans des formats tels que JavaScript Object Notation (JSON), souvent privilégié en raison de sa lisibilité tant pour les humains que pour les machines.

Une telle API doit satisfaire certains critères, notamment :

- Une architecture client-serveur avec des clients, des serveurs et des ressources, avec gestion des requêtes par HTTP
- Des communications client-serveur sans état, où les informations du client ne sont pas stockées entre les requêtes GET, chaque requête étant traitée indépendamment.
- La possibilité de mettre en cache des données pour optimiser les interactions client-serveur.
- Une interface uniforme entre les composants, garantissant un transfert standardisé des informations.
- Un système à couches, invisible pour le client, permettant de hiérarchiser les serveurs pour des aspects tels que la sécurité et l'équilibrage de charge.
- Du code à la demande (optionnel), offrant la possibilité d'envoyer du code exécutable depuis le serveur vers le client sur demande, étendant ainsi les fonctionnalités du client [44].

4.1.2 Full Stack Spring Boot Angular

Pour mettre en œuvre une application, il est essentiel de prendre des décisions quant aux technologies qui seront utilisées. Elles forment ce qu'on appelle communément un "stack" (littéralement une "pile"). Dans le cadre du projet développé au cours de ce travail, le choix a été fait d'utiliser Spring Boot pour mettre en œuvre une API pour le backend, Angular pour créer l'interface utilisateur qui permet la communication client-seveur, et MySQL pour la base de données. Cette combinaison offre une approche robuste pour le développement d'applications Web. En effet, l'utilisation conjointe de Spring Boot et Angular crée un tandem puissant, parfaitement adapté au développement d'applications Web avec une complexité réduite [8]. Ces deux frameworks se révèlent non seulement intuitifs, mais également efficaces en raison des nombreuses fonctionnalités qu'ils offrent. Ils facilitent ainsi la programmation et la gestion de la communication entre le client et le serveur, ainsi que la manipulation de la base de données. Cette synergie entre Spring Boot, Angular et MySQL permet de tirer parti de leurs avantages respectifs, offrant une expérience de développement fluide et efficace pour l'ensemble du projet.

L'architecture logicielle illustrée par la Figure 26 expose les interactions entre différentes technologies. Le client Angular émet des requêtes HTTP vers le backend Spring Boot pour réaliser diverses opérations telles que la récupération (GET), la création (POST), la mise à jour (PUT/PATCH), ou la suppression de données (DELETE) [40]. Le backend Spring Boot expose

des points de terminaison, définis dans les Controllers, spécifiant les opérations autorisées pour le frontend. En retour, il renvoie des réponses sous forme d'objets JSON.

La coordination entre la base de données et le backend s'opère de manière automatisée grâce à l'utilisation de Spring Data Java Persistence API (JPA).

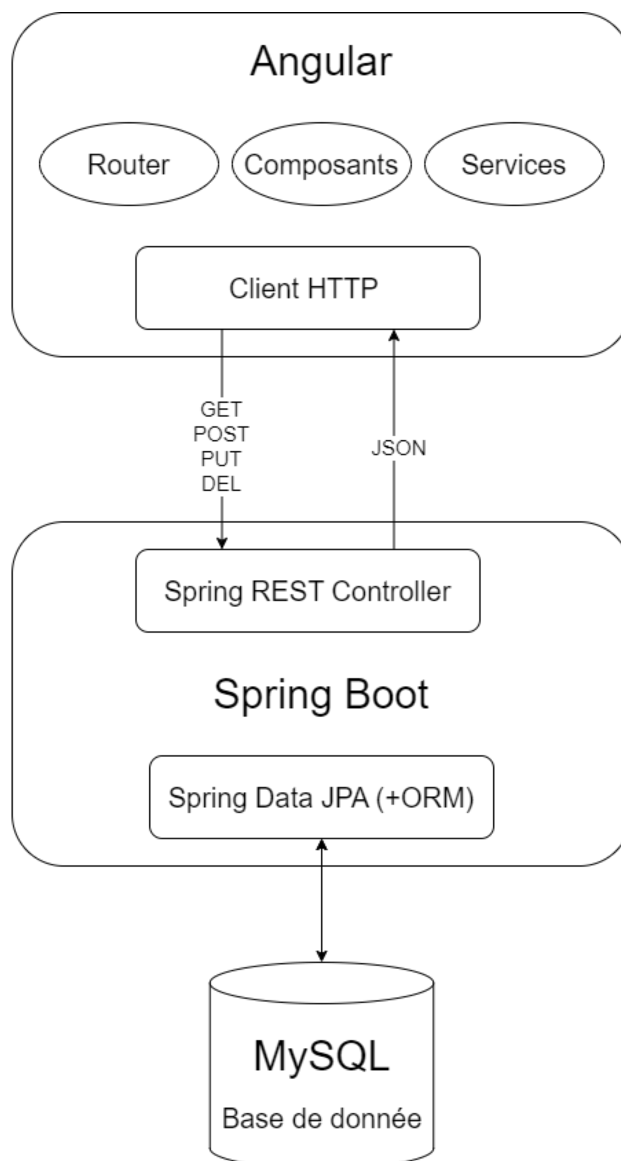

Figure 26 - Architecture logicielle

Dans la suite de ce document, chaque composant de cette architecture sera présenté, en s'appuyant sur son utilisation concrète au sein de ce projet. Cette démarche permettra de mieux comprendre la programmation du jeu, d'analyser le choix des technologiques et d'explorer plus en détail les avantages et les spécificités de cette architecture logicielle au sein de l'application.

4.2 Frontend

Le front-end dans le développement Web englobe la création des éléments visibles d'une page Internet ou d'une application, avec des fichiers HTML, CSS, et JavaScript. Concrètement, il permet à l'utilisateur d'interagir directement avec une interface compréhensible et ergonomique

regroupant une multitude de composants auxquels il a l'habitude d'être confronté. Étant donné l'évolution continue des outils et des techniques, il est essentiel de s'adapter constamment aux évolutions technologiques. La conception des sites Internet doit garantir une expérience utilisateur optimale en facilitant la navigation et l'accès à l'information. Cette tâche est particulièrement complexe étant donné la diversité des formats et tailles d'appareils. Ainsi, le développeur doit veiller à ce que le site s'affiche correctement sur tous les navigateurs Web et appareils disponibles [46]. Pour ce faire, il est courant d'utiliser des frameworks tels qu'Angular, qui s'adapte automatiquement aux différents supports et systèmes d'exploitation.

4.2.1 Angular

Angular est un framework open-source dédié au développement du client. Fondé sur TypeScript, un langage de programmation libre développé par Microsoft, Angular vise à améliorer et sécuriser la production de code JavaScript. TypeScript est un surensemble syntaxique strict de JavaScript, ce qui signifie que tout code JavaScript correct peut être utilisé avec TypeScript. Il présente l'avantage d'offrir un typage statique optionnel pour les variables et les fonctions, la possibilité de créer des classes et des interfaces, ainsi que d'importer des modules, tout en conservant l'approche non contraignante de JavaScript.

Angular représente une refonte complète d'AngularJS, un framework antérieur développé par la même équipe. Cette nouvelle itération offre des améliorations significatives, tirant parti des meilleures pratiques de développement et des évolutions technologiques pour fournir une expérience de développement moderne et performante. Ce framework est spécialement conçu pour la création d'applications Web, en mettant particulièrement l'accent sur les applications monopages (SPA Single Page Applications). Ces dernières offrent une expérience utilisateur fluide en permettant l'accès à différentes fonctionnalités sans nécessiter le rechargement de la page à chaque nouvelle action. Angular repose sur une architecture de type Modèle-Vue-Contrôleur (MVC), permettant ainsi une séparation claire des données, de l'aspect visuel, et des actions. Cette approche architecturale offre une excellente maintenabilité du code et améliore la collaboration au sein des équipes de développement [45][49].

Initialisation du framework

Avant d'entamer le développement concret de l'application avec Angular, il est préalablement requis d'installer Node.js et npm (Node Package Manager), un gestionnaire de paquets (voir Listing 1). Node.js représente un environnement d'exécution JavaScript open source et multiplateforme [19]. Une fois ces prérequis installés, l'utilisation de l'invite de commande, également appelée terminal, est nécessaire pour exécuter plusieurs commandes essentielles à l'installation du framework et à la création d'un projet.

```
6     npm install -g npm
```

Listing 1 - Installer npm

Dans un premier temps, il convient d'installer Angular CLI (Command-Line Interface), qui sera utilisé pour initialiser, développer et maintenir des applications Angular directement depuis l'invite de commande.

```
7      npm install -g @angular/cli
```

Listing 2 - Installer angular/cli

Pour amorcer la création d’un projet Angular, il faut ouvrir un terminal et naviguer jusqu’au répertoire qui contiendra le projet généré. Une fois que le terminal se trouve dans le répertoire adéquat, la commande "ng new nom-du-projet" initialisera un nouveau projet Angular.

```
8      ng new nom-du-projet
```

Listing 3 - Générer un projet Angular

Cette commande générera un nouveau dossier intitulé ici "nom-du-projet" dans le répertoire actuel, puis procédera au téléchargement et à l’installation de tous les fichiers et dépendances d’un projet Angular. Une fois les opérations finies, il est possible d’accéder à son dossier en utilisant la commande "cd nom-du-projet" puis démarrer l’application avec la commande "ng serve". Cette dernière initiera un serveur local sur l’ordinateur et ouvrira l’application dans le navigateur par défaut.

4.2.2 Implémentation de l'interface utilisateur

Lors du développement d'une application avec Angular, il est essentiel d'établir une structure claire pour faciliter la compréhension du code et l'utilisation des différents éléments. Une approche pertinente consiste à recourir à des composants et des services. Les composants contiennent les parties du code utiles à l'affichage alors que les services encapsulent les opérations courantes indispensables à une application pour séparer les fonctionnalités et les données. Une fois injectés, ils deviennent des dépendances pour les composants qui ne peuvent pas fonctionner de manière autonome. De plus, c'est au sein des services que sont utilisés les points de terminaison (endpoints), agissant comme des intermédiaires entre les composants et le backend de l'application [43] [23].

Angular Services

Dans le service dédié aux utilisateurs, représenté dans le Listing 4, plusieurs éléments clés sont présents. Tout d'abord, l'annotation @Injectable agit comme un décorateur signalant que la classe est conçue pour être injectée ailleurs dans l'application, par exemple dans un composant ou un autre service [1]. Dans de la section de déclaration des variables, l'URL de l'API est le chemin général qui va permettre aux méthodes du service de communiquer avec le backend par divers endpoints comme "/user" ou encore "/logout".

L'injection de dépendance permet d'utiliser des objets ou des services de sources externes sans avoir à les coder en dur dans l'application ce qui favorise la réutilisabilité et la modularité du code. Le constructeur de la classe en réalise deux . La première concerne l'instance de la classe HttpClient, facilitant l'envoi de requêtes HTTP vers un serveur distant. Cela est particulièrement utile pour des opérations telles que les requêtes GET, POST, et autres. La seconde injection concerne l'instance de la classe "Router" qui permet la gestion de la navigation au sein de l'application, notamment après une connexion ou une déconnexion réussie.

Les méthodes "login" et "logout" du service tirent parti des fonctionnalités d'Observable. Ce modèle de conception (design pattern) est un concept central en programmation réactive qui permet le traitement des flux de données asynchrones. Lorsqu'une requête de connexion est initiée avec la méthode login, elle retourne un Observable émettant des événements au fil de l'évolution de la requête HTTP, offrant une gestion efficace des réponses du serveur et des erreurs éventuelles [28].

```
1     // les importations sont omises
2
3     @Injectable({providedIn: 'root'})
4     export class UserService {
5       private apiURL = "http://localhost:8081/api";
6       private userId: string | null = null;
7       private isLoggedIn: boolean = false;
8       private _admin: boolean = false;
9
10      constructor(private http: HttpClient, private router: Router)
      {}
11
12      // GETTER et SETTER des champs sont omis
13
14      // Obtient les données de l’utilisateur actuellement connecté
15      getUserData(): Observable<any> {
16        if (this.userId) {
17          const url = `${this.apiURL}/user/${this.userId}’;
18          return this.http.get(url);
19        } else {
20          // Gère le cas où l’ID actuel n’est pas défini
21          return throwError('Current user ID is not set.');
22        }
23      }
24
25      // Obtient le login de l’utilisateur à partir de son ID
26      getUserLogin(userId: string): Observable<string> {
27        const url = `${this.apiURL}/user/${userId}’;
28        return this.http.get<any>(url);
29      }
30
31      // Enregistre un nouvel utilisateur
32      register(user: any): Observable<any> {
33        return this.http.post(’${this.apiURL}/user’, user);
34      }
35
36      // Effectue la connexion de l’utilisateur
37      login(credentials: { login: string, password: string }):
      Observable<any> {
38        return this.http.post(’${this.apiURL}/login’, credentials);
39      }
40
41      // Déconnecte l’utilisateur
42      logout(): Observable<string> {
43        this.isLoggedIn = false;
44        this.userId = null;
45        this.router.navigate(['/login']);
46        return this.http.post<string>(’${this.apiURL}/logout’, {});
47      }
48
49      isAuthenticated(): boolean {
50        return this.isLoggedIn;
51      }
52    }
```

Listing 4 - Services Utilisateur

Angular components

Le composant Angular appelé AuthFormComponent (voir Listing 5) représente la partie de l'interface utilisateur dédiée à l'inscription et à la connexion. Il s'agit d'un fichier TypeScript de type composant, c'est pourquoi il est décoré avec @Component qui accepte plusieurs paramètres. Le premier de ces paramètres est le sélecteur, qui détermine comment le composant sera identifié dans d'autres composants ou dans l'application en général. Ce sélecteur agit comme une balise personnalisée pouvant être utilisée dans le code HTML pour inclure le composant dans une page. Les deuxièmes et troisièmes paramètres indiquent respectivement les fichiers HTML et CSS liés au composant. L'ensemble de ces éléments concourt à la création du contenu de la page visible par l'utilisateur [3].

Ce composant utilise également le décorateur @ViewChild pour obtenir l'instance du composant MatTabGroup, facilitant ainsi la manipulation des tabulations. Par exemple, cela permet de passer directement à la tabulation "Login" une fois que l'utilisateur a fini son enregistrement (voir la Figure 16). Les propriétés "user" et "credentials" stockent respectivement les données de l'utilisateur et les informations d'identification.

Le constructeur de ce composant injecte les services nécessaires, tels que UserService pour la gestion des utilisateurs, "Router" pour la navigation, et MatSnackBar pour les notifications. Le service utilisateur est particulièrement crucial, car il permet l'utilisation des différentes méthodes mentionnées dans le Listing 5. Les méthodes "register" et "login", qui gèrent les opérations d'inscription et de connexion, utilisent ce service. Celles-ci utilisent la méthode "subscribe" pour écouter la réponse asynchrone du backend, permettant également de traiter les éventuelles erreurs survenues lors de la connexion ou de l'enregistrement d'un utilisateur.

```
1     // les importations sont omises
2
3     @Component({
4       selector: 'app-auth-form',
5       templateUrl: './auth-form.component.html',
6       styleUrls: ['./auth-form.component.css']
7     })
8     export class AuthFormComponent {
9       @ViewChild(MatTabGroup) tabGroup!: MatTabGroup;
10
11      user = { email: '', login: '', password: '' };
12      credentials = { login: '', password: '' };
13
14      constructor(
15        private userService: UserService,
16        private router: Router,
17        private snackBar: MatSnackBar) {}
18
19
20      register() {
21        // Vérifie si tous les champs sont remplis
22        if (this.user.email && this.user.login && this.user.password)
      {
23          this.userService.register(this.user).subscribe(
24            response => {
25              console.log('User registered successfully:',
      response);
26              // Affiche une notification pour une inscription
      réussie
27              this.snackBar.open('Successful registration, you can
      login', 'Close', {
28                duration: 3000,
29              });
30              // Change l'onglet actif à l'onglet de connexion après
      une inscription réussie
31              this.tabGroup.selectedIndex = 0;
32            },
33            error => {
34              console.error('Error registering user:', error);
35              // Gère les erreurs lors de l'inscription
36              if (error.status === 400 && error.error) {
37                this.snackBar.open(error.error, 'Close', {
38                  duration: 9000,
39                });
40              }
41            }
42          );
43        } else {
44          console.error('All fields (email, login, password) are
      required.');
45          // Affiche une notification en cas de champs manquants
46          this.snackBar.open('Please fill in all fields.', 'Close', {
47            duration: 5000,
48          });
49        }
50      }
51
52      login() {
```

```
53        this.userService.login(this.credentials).subscribe(response
    => {
54          console.log('Login successful:', response);
55          const userId = response.id;
56          // Définir le statut administrateur et l'ID de
    l'utilisateur connecté
57          this.userService.admin = response.admin;
58          this.userService.setUserId(userId);
59          this.userService.setLoggedIn(true);
60          // Redirige vers la page utilisateur après une connexion
    réussie
61          this.router.navigate(['/user']);
62        },
63        error => {
64          // Gère les erreurs de connexion
65          console.error('Error login user:', error);
66          if (error.status === 401) {
67            // Utilisateur non trouvé dans la base de données
68            this.snackBar.open('Incorrect login or password.',
    'Close', {
69              duration: 5000,
70            });
71          } else {
72            // Autre erreur de connexion
73            this.snackBar.open('An error has occurred during
    connection.', 'Close', {
74              duration: 5000,
75            });
76          }
77        });
78      }
79    }
```

Listing 5 - Composant de connexion et d'authentification

Bien que similaire dans sa structure, l'implémentation du composant utilisateur diffère quelque peu des autres éléments. C'est pourquoi une analyse succincte du service et du composant de l'entité catégorie est également réalisée. Afin d'éviter la redondance, le Listing 6 ne présente que les méthodes sans inclure l'intégralité de la classe.

Tout comme dans le service utilisateur, la classe HttpClient est employée pour chaque méthode qui renvoie un Observable. La différence réside dans le fait que le service catégorie utilise toutes les requêtes HTTP pour communiquer avec le backend à l'exception de PUT. En effet, les méthodes "createCategory", "getAllCategories" et "getCategoryById", "deleteCategory", "patchCategory" utilisent respectivement les requêtes POST, GET, DELETE et PATCH.

```
1       createCategory(category: any): Observable<any> {
2         return this.http.post(this.apiURL, category);
3       }
4
5       getAllCategories(): Observable<any[]> {
6         return this.http.get<any[]>(this.apiURL);
7       }
8
9       getCategoryById(categoryId: number): Observable<any> {
10        const url = `${this.apiURL}/${categoryId}`;
11        return this.http.get<any>(url);
12      }
13
14      deleteCategory(categoryId: number): Observable<void> {
15        const url = `${this.apiURL}/${categoryId}`;
16        return this.http.delete<void>(url);
17      }
18
19      patchCategory(categoryId: number, categoryUpdates: any):
        Observable<any> {
20        const url = `${this.apiURL}/${categoryId}`;
21        return this.http.patch(url, categoryUpdates);
22      }
```

Listing 6 - Méthodes CRUD du service catégorie

L'utilisation de ce service est faite dans deux fichiers TypeScript distincts. Le premier (voir Listing 7) initialise le formulaire représenté sur la Figure 23 qui permet la création d'une nouvelle catégorie en utilisant la méthode "createCategory" du service associé. Avec Angular, la méthode "ngOnInit" peut être utilisée par toutes les classes qui implémentent OnInit. Il s'agit d'une méthode de rappel qui est invoquée une seule fois lors de la création du composant. C'est un endroit idéal pour effectuer des initialisations, des appels à des services ou d'autres opérations nécessaires au démarrage du composant. Utiliser "ngOnInit" garantit que ces opérations sont effectuées au bon moment, juste après la création du composant et avant toute autre vérification de changement [29].

```
1     // les importations et annotations @Component sont omises
2
3     export class CreateCategoriesComponent implements OnInit {
4       categoryForm!: FormGroup;
5
6       constructor(private fb: FormBuilder, private categoryService:
      ManageCategoriesService) {}
7
8       ngOnInit(): void {
9         this.categoryForm = this.fb.group({
10          categoryName: ['', [Validators.required]],
11          description: ['', [Validators.required]],
12          lp: [0, [Validators.required, Validators.min(1)]],
13          dps: [0, [Validators.required, Validators.min(1)]],
14          energy: [0, [Validators.required, Validators.min(1)]],
15
16        });
17      }
18
19      onSubmit(): void {
20        if (this.categoryForm.valid) {
21          const newCategory = this.categoryForm.value;
22          this.categoryService.createCategory(newCategory).subscribe(
23            (response) => {
24              console.log('Category created successfully:',
      response);
25              // Réinitialisez le formulaire après la création
26              this.categoryForm.reset();
27            },
28            (error) => {
29              console.error('Error creating category:', error);
30            }
31          );
32        }
33      }
34    }
```

Listing 7 - Composant de création de catégories

Le second fichier représenté sur le Listing 8 fait usage de toutes les autres méthodes du service. La méthode "loadCategories" utilise la méthode du service "getAllCategories" pour afficher toutes les catégories. La méthode "removeCategory" utilise le service pour supprimer une catégorie, puis appelle "loadCategories" pour afficher uniquement les catégories qui n'ont pas été supprimées (voir la Figure 24).

Pour la modification, la méthode "patchCategory" du service est utilisée dans "saveEditedCategory". Cette dernière enregistre les modifications apportées à chaque attribut de la catégorie. Ces modifications sont préalablement enregistrées dans l'ensemble des champs de la classe ("editedCategoryName", "editedCategoryLP", etc.) lorsque le formulaire de la Figure 25 est sauvegardé. Ensuite, la liste est rechargée pour refléter ces modifications.

```
35    export class AllCategoriesComponent implements OnInit {
36      categories: any[] = [];
37      editingCategoryId: number | null = null;
38      editedCategoryName: string = '';
39      editedCategoryDescription: string = '';
40      editedCategoryLP: number = 0;
41      editedCategoryDPS: number = 0;
42      editedCategoryEnergy: number = 0;
43
44      constructor(private categoriesService: ManageCategoriesService)
      {}
45
46      ngOnInit(): void {this.loadCategories();}
47
48      loadCategories(): void {
49        this.categoriesService.getAllCategories().subscribe(
50          (categories) => {
51            this.categories = categories;
52          },
53          (error) => {
54            console.error('Error loading categories:', error);
55          }
56        );
57      }
58
59      removeCategory(categoryId: number) : void {
60        this.categoriesService.deleteCategory(categoryId).subscribe(
61          () => {
62            // Rechargez la liste des catégories après la suppression
63            this.loadCategories() ;
64          },
65          (error) => {
66            console.error('Error removing category:', error) ;
67          }
68        ) ;
69      }
70
71      confirmRemoveCategory(categoryId: number) : void {
72        const confirmation = window.confirm(
73          "Do you really want to delete the category?" +
74            "The deletion will be permanent."
75        );
76        if (confirmation) {
77          this.removeCategory(categoryId);
78        }
79      }
80
81      // Activez le mode d'édition pour cette catégorie
82      editCategory(category: any) : void {
83        this.editingCategoryId = category.id;
84        this.editedCategoryName = category.categoryName;
85        this.editedCategoryDescription = category.description;
86        this.editedCategoryLP = category.lp;
87        this.editedCategoryDPS = category.dps;
88        this.editedCategoryEnergy = category.energy;
89      }
90
91      saveEditedCategory(categoryId: number, categoryUpdates: any):
      void {
```

```
92        this.categoriesService.patchCategory(categoryId,
      categoryUpdates).subscribe(
93          (updatedCategory) => {
94            // Mettre à jour la liste des catégories après avoir
      appliqué les modifications
95            this.loadCategories();
96            this.cancelCategoryEdit();
97          },
98          (error) => {
99            console.error('Error updating category:', error);
100         }
101       );
102     }
103
104     cancelCategoryEdit(): void {
105       // Désactiver le mode d'édition et efface les champs du
      formulaire
106       this.editingCategoryId = null;
107       this.editedCategoryDescription = '';
108       this.editedCategoryName = '';
109       this.editedCategoryDPS = 0;
110       this.editedCategoryLP = 0;
111       this.editedCategoryEnergy = 0;
112     }
113   }
```

Listing 8 - Composant d'affichage et modification de catégorie

4.3 Backend

4.3.1 Spring boot

Le Java Spring Framework est une infrastructure d'entreprise open source largement utilisée, conçue pour développer des applications autonomes de production opérant sur la machine virtuelle Java (JVM). Le Framework Spring propose une injection de dépendances, permettant aux objets de définir leurs propres dépendances que le conteneur Spring injecte par la suite. Cette fonctionnalité favorise la création d'applications modulaires avec des composants faiblement couplés, idéales pour créer des petits services Web autonomes (microservices). Bien que Spring Framework soit robuste, la configuration, l'installation et le déploiement d'applications Spring peuvent être chronophages et complexes.

C'est là que l'extension Spring Boot intervient, elle simplifie et accélère le développement d'applications Web permettant aux développeurs de se concentrer davantage sur le développement de fonctionnalités plutôt que sur la configuration technique. Voici les trois fonctionnalités clés offertes par l'extension :

- Une configuration automatique : Spring Boot offre une configuration automatique intelligente, minimisant la nécessité d'une configuration manuelle fastidieuse. L'outil identifie et configure automatiquement les éléments nécessaires à l'application.
- Une approche directive de la configuration : Plutôt que de demander aux développeurs de prendre toutes les décisions de configuration, Spring Boot utilise une approche directive. Il choisit les packages à installer et les valeurs par défaut à utiliser en fonction des besoins du projet.

- La création d'applications autonomes : Spring Boot permet de créer des applications autonomes, réduisant ainsi les efforts nécessaires pour la configuration, l'installation et le déploiement des applications Spring.

Ces fonctionnalités s'harmonisent pour simplifier le processus de configuration d'une application Spring avec une intervention minimale [31].

4.3.2 Implémentation REST API

Initialisation de Spring Boot

Pour initialiser un projet Spring Boot, il existe un service en ligne qui propose la création d'un projet préconfiguré de manière simple. En sélectionnant le langage, le type de projet, la version de Spring Boot, les métadonnées et optionnellement certaines dépendances, il est possible de démarrer rapidement un projet. Par la suite, l'ajout de nouvelles dépendances est fait dans le fichier pom.xml (Maven). La configuration initiale utilisée pour l'application "OnlyQuest" est illustrée dans la Figure 27.

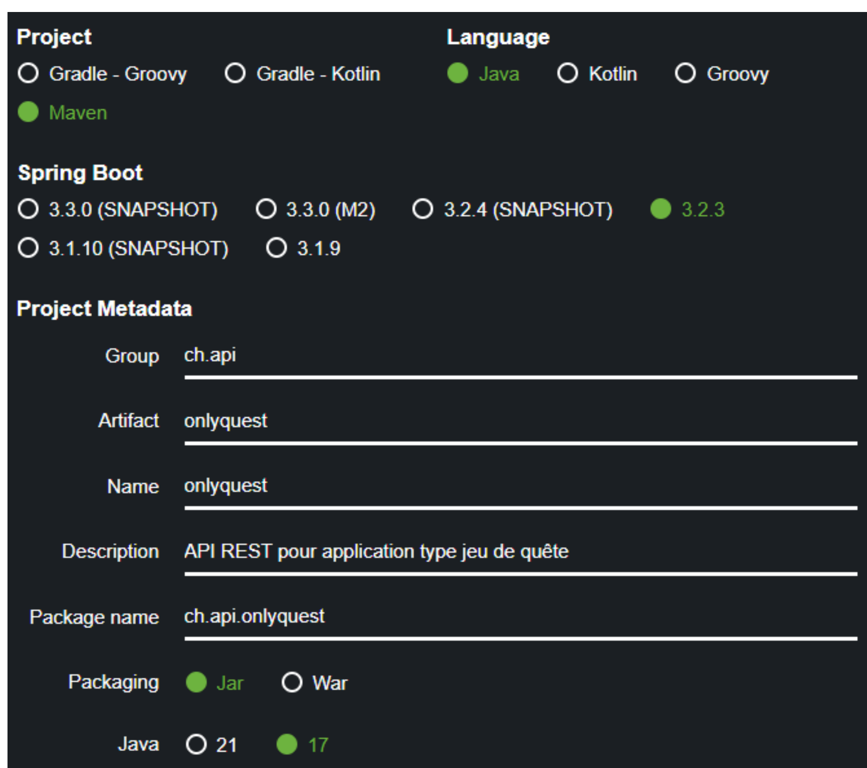

Figure 27 - Initialiseur de projet Spring Boot

L'utilisation de Spring Initializr génère un fichier ZIP contenant les éléments nécessaires pour le projet. Il suffit de le décompresser dans un environnement de travail pour s'apercevoir qu'il contient plusieurs fichiers. L'un d'entre eux se nomme "pom.xml", il joue un rôle crucial dans la configuration du projet. Il permet de spécifier diverses informations telles que les dépendances du projet, les plugins, les objectifs, les profils de construction, la version du projet, la description, les développeurs, les listes de diffusion, et d'autres paramètres essentiels à la gestion du projet. Ainsi, le pom.xml devient une pièce maîtresse, orchestrant la structure et le comportement du projet grâce à la standardisation offerte par Maven [20].

Un autre fichier important est créé, il s'agit de "application.properties" qui permet de spécifier certaines propriétés de l'application. La configuration Spring Boot présentée par le Listing 9 inclut l'activation de Swagger UI pour la documentation de l'API, les détails de la base de données MySQL, la stratégie Hibernate pour la gestion du schéma de base de données et la spécification du port du serveur intégré.

```
1    springfox.documentation.swagger-ui.enabled=true
2    ## Spring DATASOURCE
3    spring.datasource.url=jdbc:mysql://localhost:3306/TB?createDataba
     seIfNotExist=true
4    spring.datasource.username=root
5    spring.datasource.password=root
6    spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
7    ## Hibernate Properties
8    spring.jpa.hibernate.ddl-auto=update
9    spring.jpa.show-sql=true
10   spring.jpa.open-in-view=true
11   server.port=8081
```

Listing 9 - Propriété de l'API

L'initialisation produit également une classe avec une méthode "main", destinée à exécuter l'application. Cette classe utilise l'annotation @SpringBootApplication qui active le mécanisme de configuration automatique, le scan des packages de l'application et d'autres configurations [41]. C'est également ici que la configuration de Swagger est faite par openAPI.

```
1    // les importations sont omises
2
3    @SpringBootApplication
4    @OpenAPIDefinition(
5            info = @Info(
6                    title = "OnlyQuest",
7                    version = "1.0",
8                    description = "RESTful API for Quest game",
9                    termsOfService = "",
10                   contact = @Contact(
11                           name = "Mr. Lambert",
12                           email = "victor.lambert@unifr.ch"
13                   )
14           )
15   )
16   public class OnlyquestApplication {
17       public static void main(String[] args) {
18           SpringApplication.run(OnlyquestApplication.class, args);
19       }
20
21   }
```

Listing 10 - Classe main

Modèle-vue-contrôleur

Lors du développement d'une API REST avec Spring Boot et Angular, l'utilisation d'une architecture de type modèle-vue-contrôleur (MVC) est courante.

Le Modèle représente la structure de données et la logique métier de l'application. Dans Spring Boot, le modèle peut être composé d'objets Java simples ou d'entités persistantes représentant les données manipulées par l'application.

Le Contrôleur agit comme un intermédiaire entre le modèle et la vue. Il intercepte les requêtes de l'utilisateur qui seront ensuite traitées en fonction de la logique métier définie dans le modèle. Ensuite, il sélectionne la vue appropriée pour afficher les résultats. En Spring Boot, les contrôleurs sont généralement des classes annotées avec @Controller ou @RestController et contiennent des méthodes annotées avec @RequestMapping ou ses dérivées.

Ces deux couches sont représentées par deux packages distincts dans l'application, "model" et "controller", qui contiennent toutes les classes représentant les entités (utilisateur, héros, catégorie, etc.) ainsi que leur contrôleur respectif. Un package supplémentaire, souvent appelé "repository", sert de dépôt de données (data repositories).

Spring Repository

Ces repositories sont des interfaces propres à chaque entité qui étendent JpaRepository. Cette interface hérite de PagingAndSortingRepository, qui à son tour hérite de CrudRepository [11].

- CrudRepository est une interface de base offrant des méthodes standards pour effectuer des opérations create, read, update, delete (CRUD) sur une entité dans un référentiel Spring Data. Ces opérations se concentrent sur la manipulation des données en utilisant les verbes d'action HTTP standardisés tels que POST, PUT, GET et DELETE [35].
- PagingAndSortingRepository fournit des méthodes pour effectuer la pagination et le tri des enregistrements selon des critères spécifiés.
- JpaRepository offre des méthodes liées à JPA, dont le vidage du contexte de persistance. Cette opération synchronise les modifications apportées aux entités gérées avec la base de données, garantissant que toutes les modifications en attente sont effectivement écrites dans la base de données. De plus, JpaRepository propose des fonctionnalités telles que la suppression de plusieurs enregistrements en lot [22].

L'utilisation de JpaRepository dans une API est cruciale, car elle simplifie l'implémentation des méthodes CRUD et d'autres opérations sur les données. Par exemple, dans le Listing 11, UserRepository étend JpaRepository<User, Long>, ce qui offre l'accès à toutes les méthodes héritées de la classe parente. Ici, User représente le type d'entité manipulée par le repository, tandis que Long indique le type de la clé primaire associée à l'entité User.

```
1     //les importations sont omises
2
3     @Repository
4     public interface UserRepository extends JpaRepository<User,Long>
      {
5         Optional<User> findByLogin(String login);
6         boolean existsByEmailOrLogin(String email, String login);
7     }
```

Listing 11 - Dépôts utilisateur

L'annotation @Repository est utilisée pour indiquer que la classe joue le rôle d'intermédiaire dans la communication avec une source de données, par exemple MySQL.

Les méthodes définies dans la classe UserRepository sont basées sur Spring Data JPA et visent à simplifier l'accès aux données de l'entité User dans la base de données. Elles suivent une convention de nommage spécifique à Spring Data JPA, permettant la génération automatique des requêtes Structured Query Language (SQL) correspondantes. Par exemple, la méthode "findByLogin" recherche un utilisateur dans la base de données en utilisant le champ "login". La signature de la méthode indique que le résultat peut être encapsulé dans un objet de type Optional<User>. L'utilisation d'Optional facilite la vérification de la présence de la valeur à l'aide des méthodes "isPresent" et "isEmpty" [18].

La méthode "existsByEmailOrLogin" vérifie simplement l'existence d'un utilisateur en utilisant une combinaison d'adresse email (email) ou de login (login).

Certaines interfaces, comme HeroRepository sont vides. En effet, les méthodes héritées de JpaRepository sont suffisantes pour effectuer les opérations nécessaires.

```
1    //les importations sont omises
2
3    @Repository
4    public interface HeroRepository extends JpaRepository<Hero,Long>
     { }
```

Listing 12 - Dépôts héros

Comme évoqué précédemment, ces interfaces sont étroitement liées aux contrôleurs, car c'est dans ces derniers qu'elles sont utilisées. Elles facilitent ainsi considérablement la mise en œuvre des points de terminaison grâce aux opérations CRUD. Avant d'aborder l'implémentation des contrôleurs, il est essentiel d'aborder le sujet du CORS (Cross-Origin Resource Sharing) ou partage des ressources entre origines multiples.

Configuration CORS

Lors du développement d'une application Web, il est courant de séparer le frontend et le backend de l'application. Cependant, les navigateurs Web imposent une politique de "même origine" sur toutes les applications Web. Cette politique stipule qu'un client sur un navigateur ne peut interagir qu'avec des serveurs ayant la même origine que le client. Ainsi, dans le cas d'une application découplée précédemment mentionnée, si le front-end tente de demander une ressource au serveur, le navigateur générera une erreur. Comme ils ont des origines différentes, ils ne sont pas autorisés à partager des ressources entre eux, provoquant ainsi une erreur de type CORS. Par conséquent, il est nécessaire d'indiquer au navigateur d'autoriser le client et le serveur à partager des ressources tout en provenant d'origines différentes. En d'autres termes, le partage de ressources entre origines multiples (CORS), doit être activé dans l'application [4]. Cette activation est réalisée dans une classe distincte du back-end, ici nommée "WebConfig", illustrée dans le Listing 13.

```
1     // les importations sont omises
2
3     @Configuration
4     public class WebConfig {
5         @Bean
6         public WebMvcConfigurer corsConfig(){
7             return new WebMvcConfigurer() {
8                 @Override
9                 public void addCorsMappings(CorsRegistry registry) {
10                    registry.addMapping("/**")
11                            .allowedOrigins("http://localhost:4200")
12                            .allowedMethods(
13                                    HttpMethod.GET.name(),
14                                    HttpMethod.POST.name(),
15                                    HttpMethod.DELETE.name(),
16                                    HttpMethod.PATCH.name(),
17                                    HttpMethod.PUT.name())
18                            .allowedHeaders(
19                                    HttpHeaders.CONTENT_TYPE,
20                                    HttpHeaders.AUTHORIZATION);
21                }
22            };
23        }
24    }
```

Listing 13 - CORS config.

@Configuration signale à Spring que cette classe participe à la configuration de l'application. Ses méthodes seront utilisées pour créer et configurer des beans Spring. L'annotation @Bean appliquée à la méthode "addCorsMappings" indique à Spring de créer un bean de type WebMvcConfigurer.

La méthode "addCorsMappings" définie dans la classe du Listing 13 est une implémentation de l'interface WebMvcConfigurer. Elle offre une personnalisation de divers aspects de la configuration MVC, notamment la gestion des CORS. Dans ce contexte, elle donne l'accès à toutes les URL depuis le domaine "http://localhost:4200" et autorise l'utilisation des méthodes HTTP spécifiées ainsi que certains entêtes.

Une fois la configuration terminée, les fonctionnalités inter-origine sont pleinement exploitables, permettant aux services Angular de consommer l'API grâce aux contrôleurs Spring. Pour éviter les redondances, l'analyse des contrôleurs se concentrera exclusivement sur l'entité "User".

REST Controller

```
1       //les importations sont omises
2
3       @RestController
4       @RequestMapping("/api")
5       public class UserController {
6
7           @Autowired
8           private UserRepository userRepository;
9
10          @PostMapping(value = "/user", consumes =
        {MediaType.APPLICATION_JSON_VALUE})
11          public ResponseEntity<?> createUser(@RequestBody User user) {
12              user.setAdmin(false);
13              if (userRepository.existsByEmailOrLogin(user.getEmail(),
        user.getLogin())) {
14                  return ResponseEntity.status(HttpStatus.BAD_REQUEST)
15                          .body("Login or Email already used.");
16              }
17              if (!isValidEmail(user.getEmail())) {
18                  return new ResponseEntity<>("Invalid email format.",
        HttpStatus.BAD_REQUEST);
19              }
20
21              User createdUser = userRepository.save(user);
22
23              return new ResponseEntity<>(createdUser,
        HttpStatus.CREATED);
24          }
25
26          private boolean isValidEmail(String email) {
27              String emailRegex = "^[a-zA-Z0-9_+&*-]+(?:\\.[a-zA-Z0-
        9_+&*-]+)*@(?:[a-zA-Z0-9-]+\\.)+[a-zA-Z]{2,7}$";
28              Pattern pattern = Pattern.compile(emailRegex);
29              Matcher matcher = pattern.matcher(email);
30              return matcher.matches();
31          }
32
33          @PostMapping("/login")
34          public ResponseEntity<?> login(@RequestBody @Valid
        LoginRequest loginRequest) {
35              Optional<User> userOptional =
        userRepository.findByLogin(loginRequest.getLogin());
36
37              if (userOptional.isPresent()) {
38                  User user = userOptional.get();
39
40                  if
        (user.getPassword().equals(loginRequest.getPassword())) {
41                      // L'authentification réussie
42                      return ResponseEntity.ok(user);
43                  } else {
44                      // Échec de l'authentification
45                      return
        ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Invalid
        credentials");
46                  }
47              } else {
```

```
48                // L'utilisateur n'existe pas
49                return
     ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("User not
     found");
50            }
51        }
52
53        @PostMapping("/logout")
54        public ResponseEntity<String> logout() {
55            return ResponseEntity.ok().header("Location",
     "/login").body("Logout successful");
56        }
57
58        @GetMapping("/user")
59        public List<User> getAllUsers() {
60            return userRepository.findAll();
61        }
62
63        @GetMapping("/user/{id}")
64        public ResponseEntity<User> getUserById(@PathVariable Long
     id) {
65            Optional<User> user = userRepository.findById(id);
66            return user.map(value -> new ResponseEntity<>(value,
     HttpStatus.OK))
67                    .orElseGet(() -> new
     ResponseEntity<>(HttpStatus.NOT_FOUND));
68        }
69
70        @PutMapping("/user/{id}")
71        public ResponseEntity<User> updateUser(@PathVariable Long id,
     @RequestBody User updatedUser) {
72            return userRepository.findById(id)
73                    .map(user -> {
74                        user.setEmail(updatedUser.getEmail());
75                        user.setLogin(updatedUser.getLogin());
76                        user.setPassword(updatedUser.getPassword());
77                        userRepository.save(user);
78                        return new ResponseEntity<>(user,
     HttpStatus.OK);
79                    })
80                    .orElseGet(() -> new
     ResponseEntity<>(HttpStatus.NOT_FOUND));
81        }
82
83        @PatchMapping("/user/{id}")
84        public ResponseEntity<User> partialUpdateUser(@PathVariable
     Long id, @RequestBody User userUpdates) {
85            return userRepository.findById(id)
86                    .map(user -> {
87                        if (userUpdates.getEmail() != null) {
88                            user.setEmail(userUpdates.getEmail());
89                        }
90                        if (userUpdates.getLogin() != null) {
91                            user.setLogin(userUpdates.getLogin());
92                        }
93                        if (userUpdates.getPassword() != null) {
94
     user.setPassword(userUpdates.getPassword());
95                        }
96                        //uniquement les admin peuvent changer
     isAdmin
```

```
97                          if(userUpdates.isAdmin() == true){
98                              user.setAdmin(userUpdates.isAdmin());
99                          }
100                         userRepository.save(user);
101                         return new ResponseEntity<>(user,
      HttpStatus.OK);
102                     })
103                     .orElseGet(() -> new
      ResponseEntity<>(HttpStatus.NOT_FOUND));
104         }
105
106         @DeleteMapping("/user/{id}")
107         public ResponseEntity<Void> deleteUser(@PathVariable Long id)
      {
108             userRepository.deleteById(id);
109             return new ResponseEntity<>(HttpStatus.NO_CONTENT);
110         }
111     }
```

Listing 14 - Contrôleurs pour utilisateur

Dans le Listing 13, diverses annotations jouent un rôle important dans la définition et la structuration du contrôleur. L'annotation @RestController identifie la classe en tant que contrôleur Spring, chargé de gérer les requêtes HTTP et de retourner des réponses HTTP. Cette annotation englobe également @Controller, qui est une spécialisation de la classe @Component. @ResponseBody, intégré dans @RestController, permet la sérialisation automatique des objets de retour dans HttpResponse [38].

L'annotation @RequestMapping("/api") créée un mappage, autrement dit une association entre les requêtes HTTP du frontend et les endpoints du backend. Elle définit également le chemin de base pour toutes les méthodes du contrôleur, ajoutant "/api" comme préfixe à chaque route définie. Cette annotation peut être employée soit au niveau de la classe pour définir des mappages partagés, soit au niveau de la méthode pour restreindre un mapping à un endpoint spécifique. Des variantes de raccourci, telles que @GetMapping, @PostMapping, @PutMapping, @PatchMapping et @DeleteMapping, sont également disponibles pour spécifier le type de requête HTTP associé à chaque méthode [24].

La méthode "createUser" est annotée avec @PostMapping(), indiquant qu'elle répond aux requêtes HTTP de type POST. Elle consomme des données au format JSON grâce à l'annotation @RequestBody, qui lie le corps de la requête à l'objet "User" passé en paramètre [37].

L'annotation @Autowired est utilisée pour injecter automatiquement une instance de "UserRepository" dans la variable "userRepository". Cette injection peut également être réalisée par le biais d'un constructeur.

Ces annotations simplifient la gestion des opérations CRUD sur les utilisateurs dans l'API, en fournissant des points de terminaison structurés et définis pour interagir avec la base de données.

Documentation Swagger et endpoint

Chaque méthode de mappage dans une application crée des points d'entrée. Ces endpoints représentent les différentes fonctionnalités offertes par l'application, telles que la récupération de données, la création de nouvelles ressources, la mise à jour ou la suppression d'éléments existants, etc.

L’un des défis lors du développement d’une API est de les documenter efficacement afin que les utilisateurs puissent comprendre comment les utiliser correctement. C’est là qu’intervient Swagger, une plateforme open source pour la documentation, la conception et le test d’API. Swagger permet de générer automatiquement une documentation claire et détaillée pour les endpoints d’une API à partir des informations fournies dans le code source. Les développeurs peuvent ainsi l’explorer pour comprendre les fonctionnalités offertes par une API, savoir comment les appeler correctement, quels paramètres fournir et quels types de réponses ils peuvent attendre.

La Figure 28 représente tous les endpoints correspondant aux fonctionnalités des utilisateurs de l’application OnlyQuest. Il existe deux endpoints de type GET, "/api/user" et "api/user/{id}". Le premier permet de récupérer la liste de tous les utilisateurs et le second de trouver un utilisateur en indiquant son ID. DELETE permet de supprimer un utilisateur, PUT est utilisé pour modifier l’ensemble des attributs d’un utilisateur et PATCH pour changer un ou plusieurs attributs individuellement. Ces différentes méthodes sont présentes pour chaque entité illustrée dans les figures Figure 31 à Figure 34 et fonctionnent de manière équivalente.

En plus des fonctionnalités précédemment mentionnées, il existe deux endpoints supplémentaires de type POST. Le "login" permet à l’utilisateur de s’authentifier auprès de l’API en envoyant ses informations d’identification qui seront ensuite comparées avec les données stockées dans la base de données. La ressource qui en résulte est une nouvelle session. Le "logout" permet de se déconnecter et d’ajouter des actions spécifiques, par exemple supprimer les informations de connexion, vider le cache ou supprimer le jeton d’authentification.

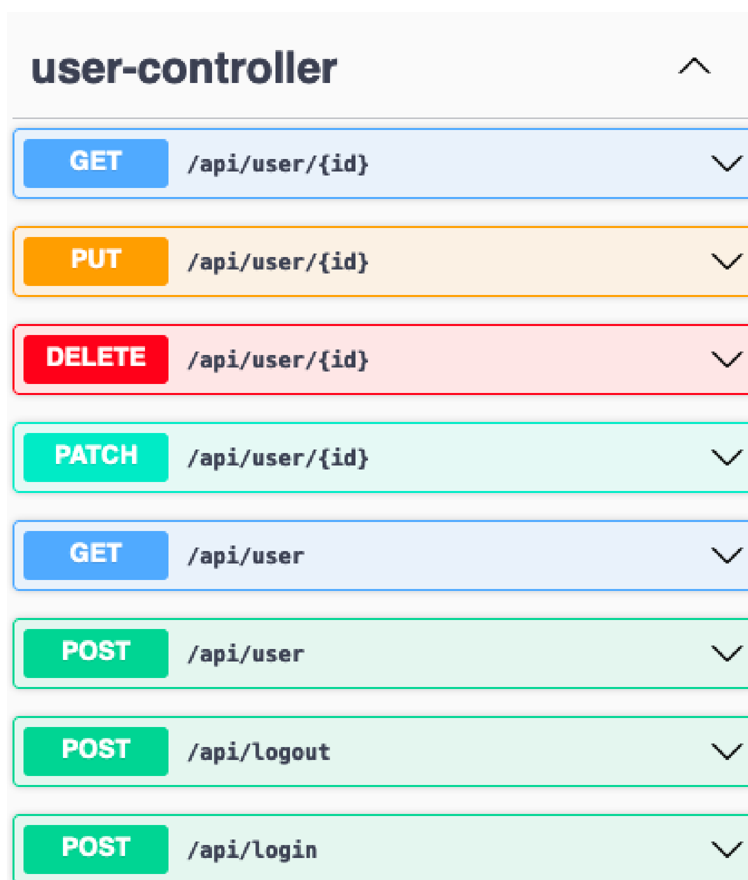

Figure 28 - Endpoint (user)

La Figure 29 présente le schéma de données pour les utilisateurs, détaillant pour chaque attribut le type qui lui est associé. Le type "integer" représente un nombre entier, "string" une chaine de caractères et "boolean" une valeur binaire (1 = vrai, 0 = faux). Le champs "heroes" est plus complexe, car il représente une liste qui regroupe plusieurs attributs et d'autres listes concernant les relations de l'entité héros.

Figure 29 - Schéma de données (user)

En plus de la documentation, Swagger offre la possibilité de tester les requêtes afin de vérifier que celles-ci fonctionnent correctement. La Figure 30 montre une requête de type POST et la réponse envoyée par le serveur. La première rubrique affiche le corps de la requête en format JSON qui sert à créer un nouvel utilisateur. Le type de chaque attribut doit correspondre aux types de données définies dans l'API. La deuxième rubrique affiche les différents éléments de réponse du serveur, dont le code 201 "created" qui indique que la requête a bien fonctionné et que la ressource a été créée avec succès.

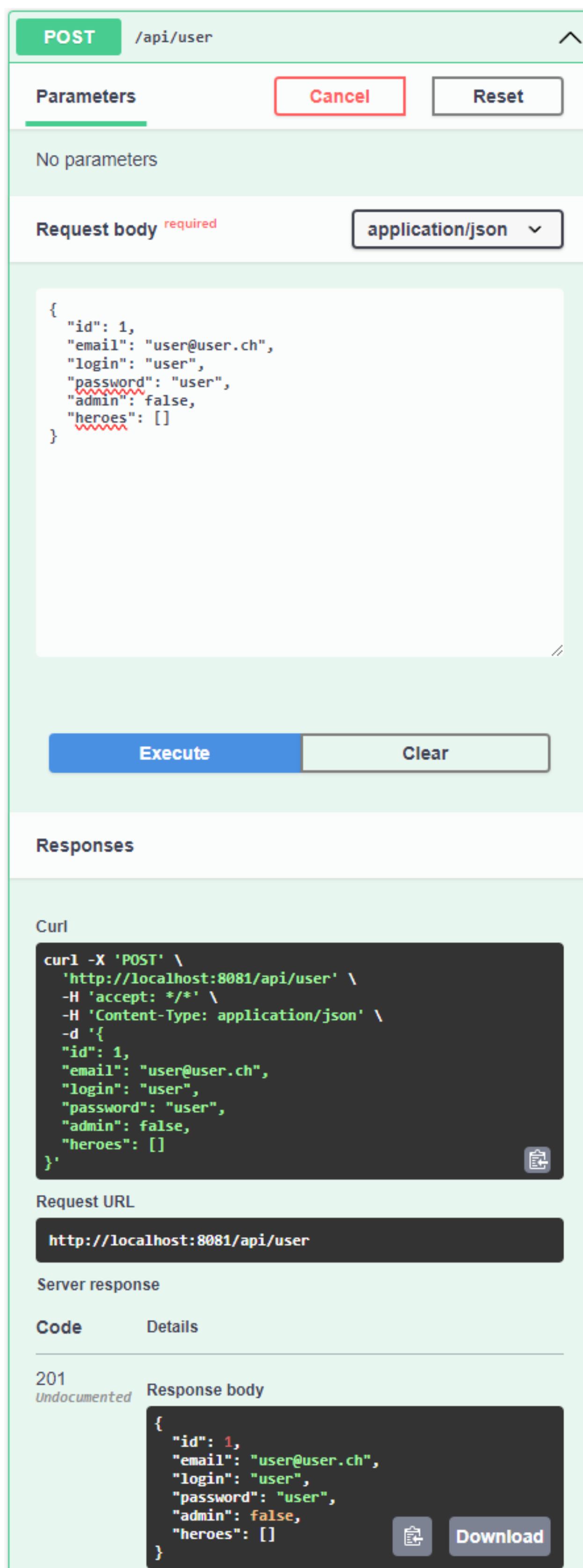

Figure 30 - Requête de création d'utilisateur avec Swagger

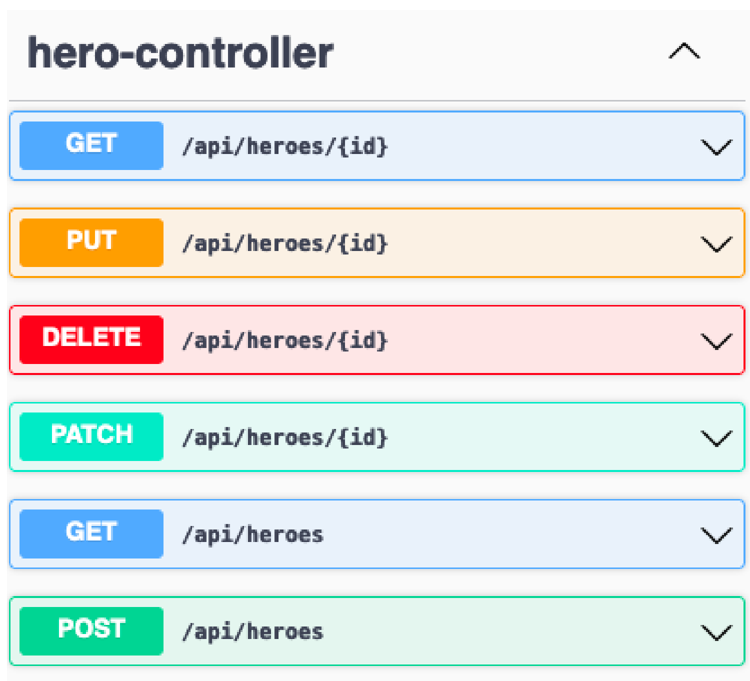

Figure 31 - Endpoint (hero)

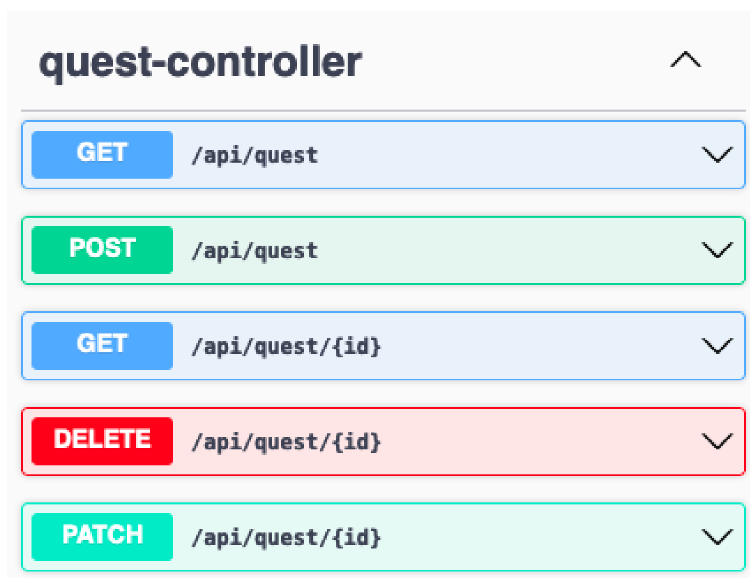

Figure 32 - Endpoint (quest)

category-controller

GET /api/category/{id}

PUT /api/category/{id}

DELETE /api/category/{id}

PATCH /api/category/{id}

GET /api/category

POST /api/category

Figure 33 - Endpoint (category)

competence-controller

GET /api/competence

POST /api/competence

GET /api/competence/{id}

DELETE /api/competence/{id}

PATCH /api/competence/{id}

Figure 34 - Endpoint (competence)

REST Entity (ORM et JPA)

Avant d'aborder le sujet des entités, il est crucial de comprendre un concept étroitement lié à celles-ci, à savoir le mappage objet-relation (ORM).

Le mappage objet-relation (ORM) représente une technique ou un modèle de conception utilisé pour accéder à une base de données relationnelles à partir d'un langage orienté objet. Spring-ORM englobe plusieurs technologies de persistance, notamment JPA (Java Persistence API), largement utilisé pour maintenir la cohérence des données entre les objets Java et les bases de données relationnelles. Cette approche agit comme un pont entre les modèles de domaine orientés objet et les systèmes de bases de données relationnelles. Hibernate, un framework Java, simplifie également le développement d'applications Java interagissant avec une base de données [34].

L'ORM offre un soutien significatif pour la gestion des ressources, l'accès aux données (DAO), les implémentations de mises en œuvre et les stratégies de transaction. Spring apporte des améliorations notables à la couche ORM lors de la création d'applications, facilitant ainsi l'intégration et réduisant l'effort, le coût, et les risques associés à la construction d'une infrastructure similaire en interne [26].

À l'aide de Hibernate, le schéma est créé de manière automatique à partir du modèle de domaine. Il n'est donc pas nécessaire de fournir manuellement des instructions SQL pour définir les tables et les relations entre les entités, ce qui réduit considérablement la charge de travail lors du développement. Pour illustrer concrètement le bon fonctionnement de ce mécanisme, la Figure 35 présente les tables générées par Hibernate lors de l'exécution de l'application. Ce schéma a été directement fait à l'aide d'un outil de conception offert par PhpMyAdmin.

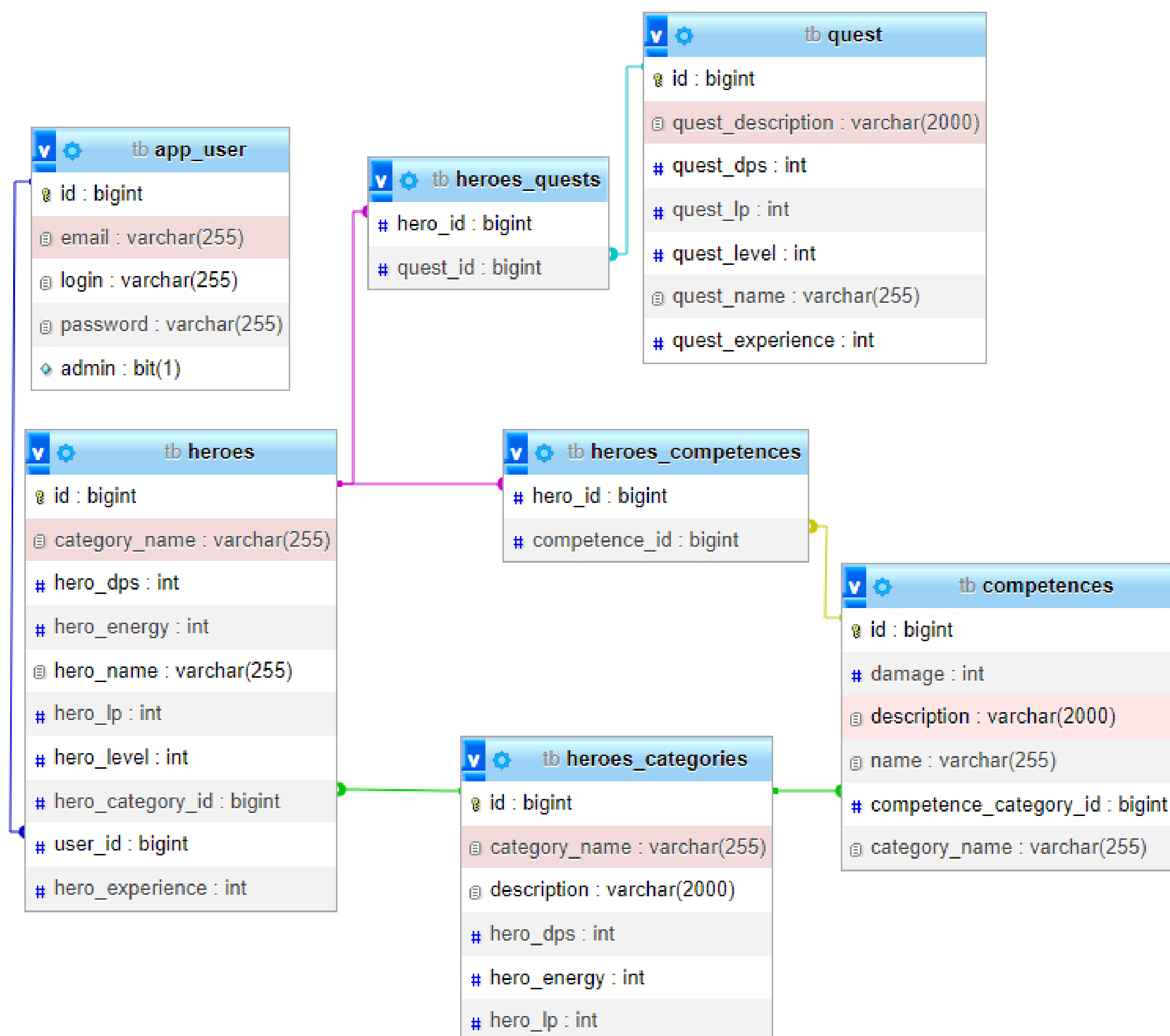

Figure 35 - PhpMyAdmin concepteur de table

La comparaison la Figure 35 et la Figure 14 montre que les deux représentations coïncident. Chaque entité représentée par un rectangle sur le diagramme modèle entité-relation est également présente sur celui généré par PhpMyAdmin. Les relations many-to-many ont créé des tables distinctes.

Pour implémenter une entité basée sur JPA, l'annotation @Entity demeure essentielle car elle indique explicitement qu'il s'agit d'une entité JPA. L'annotation @Table permet simplement de choisir le nom de la table [2]. À titre d'exemple, une analyse sera effectuée sur l'entité "Hero", car elle englobe l'ensemble du contenu essentiel nécessaire à la compréhension des modèles.

```
1     // les importations sont omises
2
3     @EqualsAndHashCode
4     @AllArgsConstructor
5     @NoArgsConstructor
6     @Builder
7     @Data
8     @Entity
9     @Table(name = "heroes")
10    public class Hero  {
11
12        @Id
13        @GeneratedValue(strategy = GenerationType.IDENTITY)
14        private Long id;
15
16        @Column(name = "hero_name", nullable = false)
17         String heroName;
18        @Column(name = "hero_level", nullable = false)
19         int lvl;
20        @Column(name = "hero_experience", nullable = false)
21        private int experience;
22
23        @Column(name = "hero_LP", nullable = false)
24        private int lp;
25        @Column(name = "hero_dps", nullable = false)
26        private int dps;
27        @Column(name = "hero_energy", nullable = false)
28        private int energy;
29        private String categoryName;
30
31        //foreign key
32        @ManyToOne
33        @JoinColumn(name = "hero_category_id")
34        private Category heroCategory;
35        //foreign key
36        @ManyToOne
37        @JoinColumn(name = "user_id")
38        private User userHeroes;
39
40
41        @ManyToMany(
42                fetch = FetchType.LAZY,
43                cascade = CascadeType.MERGE)
44        // les colonnes qui vont être créées dans la table MANY-MANY
45        @JoinTable(
46                joinColumns = @JoinColumn(name = "hero_id"),
47                inverseJoinColumns = @JoinColumn(name = "quest_id"))
48        //annotation pour éviter une boucle infinie
49        @JsonIgnoreProperties("heroes")
50        private List<Quest> quests = new ArrayList<>();
51
52        @ManyToMany(
53                fetch = FetchType.LAZY,
54                cascade = CascadeType.MERGE)
55        @JoinTable(
56                joinColumns = @JoinColumn(name = "hero_id"),
57                inverseJoinColumns = @JoinColumn(name =
      "competence_id"))
58        @JsonIgnoreProperties("heroes")
59        private List<Competence> competences = new ArrayList<>();
```

```
60
61
62        @JsonBackReference(value = "HeroesInCategory")
63        public Category getHeroCategory() {
64            return heroCategory;
65        }
66
67
68        @JsonBackReference(value = "HeroesUser")
69        public User getUserHeroes() {
70            return userHeroes;
71        }
72
```

Listing 15 - Entité (héros)

Cette classe possède plusieurs annotations, telles que @AllArgsConstructor qui génère automatiquement un constructeur prenant en compte tous les champs de la classe comme paramètres. D'autre part, @NoArgsConstructor génère automatiquement un constructeur sans paramètre [10].

Ensuite, @Builder se charge de créer automatiquement un constructeur de type "builder", facilitant la création d'instances avec une initialisation sélective des champs avec des méthodes chaînées. Cette approche se montre particulièrement avantageuse lorsque de nombreux champs sont optionnels [7].

@Data est une annotation composite regroupant plusieurs autres annotations telles que @ToString, @EqualsAndHashCode, @Getter, @Setter, et @RequiredArgsConstructor. Elle génère automatiquement les méthodes "toString", "equals", "hashCode", ainsi que les getters et les setters pour tous les champs de la classe. Cette annotation simplifie la création de classes en éliminant le besoin d'écrire ces méthodes manuellement [12].

Les champs de la classe sont également annotés pour définir leurs propriétés. La première annotation, @Id, permet à JPA de reconnaître le champ comme l'identifiant unique de la table. En complément, @GeneratedValue garantit que la valeur de cet identifiant est automatiquement générée [https://spring.io/guides/gs/accessing-data-jpa]. Quant à l'annotation @Column, elle permet de spécifier certaines caractéristiques des colonnes de la table, telles que le nom ou la possibilité d'accepter des valeurs nulles.

```
1     //représentation partielle de l'entité "User"
2
3     @OneToMany(mappedBy = "userHeroes")
4     private List<Hero> heroes;
5
6     @JsonManagedReference(value = "HeroesUser")
7     public List<Hero> getHeroes() {
8          return heroes;
9     }
```

Listing 16 - Entité User (code partiel)

```
1    //représentation partielle de l'entité "Competence"
2
3    @ManyToMany(
4            mappedBy = "competences",
5            fetch = FetchType.LAZY,
6            cascade = CascadeType.MERGE)
7    //annotation pour éviter une boucle infinie
8    @JsonIgnoreProperties("competences")
9    private List<Hero> heroes = new ArrayList<>();
10
```

Listing 17 - Entité Competence (code partiel)

Ensuite viennent les annotations de relation. Tout comme en SQL, JPA propose quatre types de relations. Dans le cadre de ce projet, uniquement trois d'entre elles sont utilisées.

Les annotations @OneToMany et @ManyToOne fonctionnent de concert, comme le montrent le Listing 15 (lignes 36-38) et le Listing 16, illustrant ce type de relation. L'entité User utilise "mappedBy" au sein de l'annotation @OneToMany pour indiquer que la liste de héros fait référence aux champs "UserHeroes" définis dans la classe Hero du Listing 15. En se référant à la Figure 35, la représentation de cette référence est symbolisée par le lien bleu foncé reliant "id" de la table User et "UserId" de la table Heroes.

Un utilisateur peut posséder plusieurs héros, tandis qu'un héros ne peut appartenir qu'à un seul utilisateur, créant ainsi une relation bidirectionnelle. Cette configuration permet d'obtenir des informations sur les deux entités liées en interrogeant chacune d'entre elles de manière indépendante [6].

Les relations bidirectionnelles permettent une navigation dans un graphe d'objets dans les deux sens. Cependant, la sérialisation de ces relations dans des formats tels que JSON pose certaines difficultés. Le problème survient lors de la conversion de ces objets interconnectés en un format sérialisé, tel que JSON. Prenons l'exemple d'un utilisateur :

Lors de la sérialisation, la propriété "id" de l'utilisateur est d'abord traitée, puis la liste des héros associés à l'utilisateur commence à être sérialisée. Cependant, le premier héros de la liste fait référence à l'utilisateur d'origine. Cette situation crée un cycle infini où l'utilisateur est associé à un héros qui renvoie à l'utilisateur lui-même. Cette boucle continue indéfiniment et provoque une erreur de débordement de pile en raison d'une récursivité infinie.

La résolution de ce problème est proposée par deux annotations :

L'annotation @JsonManagedReference concerne la partie parente ou "avant" de la relation. Elle indique à Jackson de gérer la sérialisation de la propriété associée, et le JSON résultant inclura les détails de cette propriété. Quant à l'annotation @JsonBackReference, elle désigne le côté enfant dit "retour" de la relation. Jackson la reconnaîtra et évitera de sérialiser cette propriété, empêchant ainsi la récursivité infinie mentionnée précédemment. Ces annotations fournissent une solution pour gérer les relations bidirectionnelles lors de la sérialisation dans des formats tels que JSON [39].

La relation entre les entités "Hero" et "Competence" est de type multiple-multiple. Cela signifie qu'un héros peut posséder plusieurs compétences et qu'une compétence peut être détenue par plusieurs héros. Dans ce cas, Spring Boot propose l'annotation @ManyToMany qui doit être utilisée des deux côtés des entités concernées. La spécification de la table de jointure se fait du côté du propriétaire grâce à l'annotation @JoinTable.

Dans le cas d'une relation bidirectionnelle, chaque côté peut être désigné comme étant le propriétaire. Dans ce cas précis, le choix a été fait de désigner l'entité héros comme étant prioritaire. Si la relation est bidirectionnelle, le côté qui n'est pas propriétaire doit utiliser l'attribut mappedBy de l'annotation @ManyToMany pour indiquer le champ de relation du côté propriétaire, en l'occurrence le champ "compétences".

Dans ce type de relation, l'utilisation de l'annotation @JsonIgnoreProperties permet d'éviter les problèmes de débordement de pile lors de la sérialisation. Cela garantit que les propriétés spécifiées ne sont pas sérialisées dans le JSON, ce qui prévient les boucles infinies.

4.4 Base de données

La base de données est hébergée en local sur localhost:3306 par un serveur fourni par WampServer. Cet environnement de développement Web conçu pour créer des applications Web dynamiques propose un serveur Apache2 qui est un serveur Web open source utilisé pour héberger des sites Web et des applications Web. Il fonctionne avec le langage de scripts PHP et une base de données MySQL. Pour améliorer la gestion de la base de données, il est équipé d'une interface PHPMyAdmin. Le backend utilise le mappage objet-relationnel, il offre alors une compatibilité parfaite avec MySQL, qui est un système pour la gestion de bases de données relationnelles [42][47]. Ainsi, aucune implémentation ni requête SQL n'est nécessaire pour créer la base de données. Lors de l'exécution de l'API, la base de données est automatiquement créée à partir des informations fournies dans le Listing 9. Toutes les entités deviennent des tables et tous les champs de ces entités deviennent des colonnes. Les relations many-to-many créent également des tables à part entière.

Table	Action					
app_user	Parcourir	Structure	Rechercher	Insérer	Vider	Supprimer
competences	Parcourir	Structure	Rechercher	Insérer	Vider	Supprimer
heroes	Parcourir	Structure	Rechercher	Insérer	Vider	Supprimer
heroes_categories	Parcourir	Structure	Rechercher	Insérer	Vider	Supprimer
heroes_competences	Parcourir	Structure	Rechercher	Insérer	Vider	Supprimer
heroes_quests	Parcourir	Structure	Rechercher	Insérer	Vider	Supprimer
quest	Parcourir	Structure	Rechercher	Insérer	Vider	Supprimer
7 tables	Somme					

Figure 36 - Toutes les tables

←T→			id	category_name	description	hero_dps	hero_energy	hero_lp
Éditer	Copier	Supprimer	1	Wizzard	(...)	100	100	100
Éditer	Copier	Supprimer	2	Archer	(...)	150	15	60
Éditer	Copier	Supprimer	3	Warrior	(...)	50	30	200

Figure 37 - Table heroes-categories

5
Conclusion

5.1 Résultat 56
5.2 Amélioration 57
5.3 Perspectives....57

5.1 Résultats

Développement de l'API avec Spring Boot

L'objectif initial de ce projet était de concevoir une API pour simplifier le développement, la maintenance et l'expansion d'un jeu. Pour ce faire, j'ai choisi d'utiliser Spring Boot pour l'implémentation côté serveur. Ce framework s'est avéré être un choix judicieux car il offre une interface agréable pour la configuration d'un projet complet. Grâce à Spring Boot, j'ai pu mettre en place rapidement et efficacement les fonctionnalités nécessaires à mon application.

L'utilisation conjointe de Spring Boot et d'une base de données MySQL a été particulièrement efficace. Effectivement, elle a permis de créer une structure robuste et autonome en ce qui concerne la gestion des données. L'utilisation d'un ORM a permis de mapper les objets Java aux entités de la base de données de manière transparente, ce qui a simplifié la manipulation et la persistance des données. Cette approche a grandement contribué à la fiabilité et à la pérennité de l'application.

Utilisation d'Angular pour le client :

Angular a été un choix naturel pour le développement du client qui consomme cette API. Grâce à ses nombreuses extensions et fonctionnalités, j'ai pu créer une interface utilisateur riche et interactive qui offre aux utilisateurs la possibilité d'interagir avec l'application de manière intuitive. L'écosystème d'Angular a fourni les outils nécessaires pour concevoir une interface utilisateur esthétique et ergonomique, tout en garantissant une expérience utilisateur fluide et agréable.

Réussite de l'objectif du projet :

À travers cette implémentation complète, j'ai pu atteindre l'objectif principal du projet qui était de simplifier et d'optimiser le développement d'un jeu vidéo. La combinaison efficace des technologies m'a permis de créer une application cohérente et performante, prête à être utilisée par des utilisateurs.

5.2 Améliorations

Améliorations potentielles de la sécurité

Bien que l'application remplisse son objectif fonctionnel, un aspect crucial qui n'a pas été abordé est celui de la sécurité. Actuellement, les mots de passe sont stockés en clair dans la base de données, ce qui représente une vulnérabilité majeure. Une amélioration essentielle consisterait à crypter les mots de passe afin de renforcer la sécurité des utilisateurs. De plus, l'ajout d'une validation d'email permettrait de garantir l'authenticité des informations fournies par les joueurs, tout en facilitant la récupération ou la modification des identifiants en cas de besoin. Intégrer des jetons (tokens) via Spring Security constituerait une solution efficace pour contrôler l'accès à l'application et vérifier les privilèges des utilisateurs, notamment s'ils sont des administrateurs. Enfin, pour une sécurité renforcée, l'utilisation du protocole HTTPS serait indispensable pour chiffrer les échanges de données entre le client et le serveur, garantissant ainsi la confidentialité et l'intégrité des informations.

Possibilités d'améliorations futures

Actuellement, l'application est uniquement centrée sur la gestion de contenu du jeu. Certaines fonctionnalités clés telles que le système de quêtes, le gameplay et l'aspect graphique du jeu n'ont pas été développées dans le cadre de ce projet. Ces aspects pourraient faire l'objet d'un travail ultérieur pour enrichir l'expérience utilisateur et offrir un jeu complet et attrayant.

Déploiement sur serveur ou Cloud

Enfin, pour rendre l'application accessible à un plus large public et assurer sa disponibilité continue, le déploiement sur un serveur ou un service de Cloud computing serait une étape logique. Cela permettrait aux utilisateurs d'accéder à l'application depuis n'importe quel appareil connecté à Internet, offrant ainsi une plus grande flexibilité et une meilleure expérience utilisateur. De plus, le déploiement offrirait également des avantages en termes de gestion des ressources, de sauvegarde des données et de scalabilité, garantissant ainsi la stabilité et la performance de l'application à long terme.

5.3 Perspectives

En conclusion, ce projet a mis en lumière l'importance cruciale de concevoir des outils et des processus adaptés pour soutenir le développement de jeux vidéo. En fournissant une infrastructure solide et des interfaces conviviales, de nouvelles possibilités d'innovation et d'amélioration sont ouvertes dans ce domaine en constante évolution. À l'image des grands Frameworks comme Unity ou Unreal Engin, ce projet représente une étape importante dans la création d'applications de jeu modernes. Il démontre également le potentiel des technologies telles que Spring Boot et Angular pour répondre aux besoins de l'industrie du jeu vidéo.

References

[1] À quoi sert @Injectable dans Angular ? https://angular.fr/services/injectable.html (dernière consultation le Mars 15, 2024)

[2] Accessing Data with JPA https://spring.io/guides/gs/accessing-data-jpa (dernière consultation le Mars 26, 2024)

[3] Angular components overview https://angular.io/guide/component-overview (dernière consultation le Mars 15, 2024)

[4] Angular CORS Guide https://www.stackhawk.com/blog/angular-cors-guide-examples-and-how-to-enable-it/ (dernière consultation le Mars 20, 2024)

[5] Architecture logicielle - Définition https://www.techno-science.net/glossaire-definition/Architecture-logicielle.html (dernière consultation le Mars 10, 2024)

[6] Bidirectional Relationship Using @OneToMany/@ManyToOne Annotation In Spring Boot https://medium.com/@arijit83work/bidirectional-relationship-using-onetomany-manytoone-annotation-in-spring-boot-3b91980ca222 (dernière consultation le Avril 02, 2024)

[7] Builder https://projectlombok.org/features/Builder (dernière consultation le Mars 26, 2024)

[8] Building a Web Application With Spring Boot and Angular https://www.baeldung.com/spring-boot-angular-web (dernière consultation le Mars 13, 2024)

[9] Comment installer Angular ? https://angular.fr/get_started/installation.html (dernière consultation le Mars 15, 2024)

[10] Constructor https://projectlombok.org/features/constructor (dernière consultation le Mars 26, 2024)

[11] CrudRepository, JpaRepository https://www.baeldung.com/spring-data-repositories (dernière consultation le Mars 18, 2024)

[12] Data https://projectlombok.org/features/Data (dernière consultation le Mars 26, 2024)

[13] Dofus - site officiel https://www.dofus.com/fr/mmorpg/decouvrir (dernière consultation le Fevrier 29, 2024)

[14] GitHub - backend https://github.com/VicoLambico/TB-backend.git

[15] GitHub - frontend https://github.com/VicoLambico/TB-frontend.git

[16] Google play store Dofus https://play.google.com/store/apps/details?id=com.ankama.dofustouch&hl=fr_CH&gl=US (dernière consultation le Fevrier 29, 2024)

[17] Google play store Shakes & Fidget https://play.google.com/store/apps/details?id=com.playagames.shakesfidget&hl=fr_CH&gl=US (dernière consultation le Fevrier 29, 2024)

[18] Guide To Java 8 Optional https://www.baeldung.com/java-optional (dernière consultation le Mars 18, 2024)

[19] Introduction to Node.js https://nodejs.org/en/learn/getting-started/introduction-to-nodejs (dernière consultation le Mars 15, 2024)

[20] Introduction to the POM https://maven.apache.org/guides/introduction/introduction-to-the-pom.html (dernière consultation le Mars 18, 2024)

[21] Java Web Project | Create Login and Register Form From Scratch with, Java11, Spring MVC, PostgreSQL https://www.youtube.com/watch?v=x_nfnVU0wAI (dernière consultation le Octobre 10, 2023)

[22] JPA/Hibernate Persistence https://www.baeldung.com/jpa-hibernate-persistence-context (dernière consultation le Mars 18, 2024)

[23] Lesson 09: Angular services https://angular.io/tutorial/first-app/first-app-lesson-09 (dernière consultation le Mars 15, 2024)

[24] Mapping Requests https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-requestmapping.html (dernière consultation le Mars 20, 2024)

[25] Meier, A. (2006). *Introduction pratique aux bases de données relationnelles* (Deuxième édition.) page 20-25.

[26] Object Relational Mapping (ORM) Data Access https://docs.spring.io/spring-framework/docs/3.0.x/spring-framework-reference/html/orm.html (dernière consultation le Mars 22, 2024)

[27] Object Relational Mapping (ORM) Data Access https://docs.spring.io/spring-framework/docs/3.0.x/spring-framework-reference/html/orm.html (dernière consultation le Avril 09, 2024)

[28] Observables in Angular https://angular.io/guide/observables-in-angular (dernière consultation le Mars 15, 2024)

[29] OnInit https://angular.io/api/core/OnInit (dernière consultation le Mars 15, 2024)

[30] Qu'est qu'un client informatique ? https://www.ionos.fr/digitalguide/serveur/know-how/quest-ce-quun-client-informatique/#:~:text=Un%20client%20informatique%20est%20un,%C3%A9lectronique%20ou%20les%20navigateurs%20Web (dernière consultation le Mars 04, 2024)

[31] Qu'est-ce que Java Spring Boot ? https://www.ibm.com/fr-fr/topics/java-spring-boot (dernière consultation le Mars 18, 2024)

[32] Shakes & Fidget - site officiel https://sfgame.net/ (dernière consultation le Fevrier 29, 2024).

[33] Shakes & Fidget - The Game https://www.jeuxonline.info/jeu/Shakes_&_Fidget_The_Game#:~:text=Shakes%20%26%20Fidget%20est%20un%20jeu,une%20des%20trois%20classes%20disponibles. (dernière consultation le Fevrier 29, 2024)

[34] Spring - ORM Framework https://www.geeksforgeeks.org/spring-orm-framework (dernière consultation le Mars 22, 2024)

[35] Spring Boot CRUD Operations https://www.javatpoint.com/spring-boot-crud-operations (dernière consultation le Mars 20, 2024)

[36] Spring security and Angular https://spring.io/guides/tutorials/spring-security-and-angular-js (dernière consultation le Mars 14, 2024)

[37] Spring's RequestBody https://www.baeldung.com/spring-request-response-body (dernière consultation le Mars 20, 2024)

[38] The Spring @Controller https://www.baeldung.com/spring-controller-vs-restcontroller (dernière consultation le Mars 20, 2024)

[39] Understanding Spring's @JsonBackReference and @JsonManagedReference Annotations https://medium.com/@AlexanderObregon/understanding-springs-jsonbackreference-and-jsonmanagedreference-annotations-783090468572 (dernière consultation le Avril 02, 2024)

[40] Using HTTP Methods for RESTful Services https://www.restapitutorial.com/lessons/httpmethods.html (dernière consultation le Mars 12, 2024)

[41] Using the @SpringBootApplication Annotation https://docs.spring.io/spring-boot/docs/2.0.x/reference/html/using-boot-using-springbootapplication-annotation.html (dernière consultation le Mars 18, 2024)

[42] WampServer https://www.wampserver.com/ (dernière consultation le Avril 02, 2024)

[43] What are CRUD operations in a REST API? https://www.basedash.com/blog/what-are-crud-operations-in-a-rest-api (dernière consultation le Mars 15, 2024)

[44] What is rest api ? https://www.redhat.com/fr/topics/api/what-is-a-rest-api (dernière consultation le Mars 10, 2024)

[45] Wikipedia - Angular https://fr.wikipedia.org/wiki/Angular (dernière consultation le Mars 15, 2024)

[46] Wikipedia - Développement Web frontal https://fr.wikipedia.org/wiki/D%C3%A9veloppement_web_frontal (dernière consultation le Mars 15, 2024)

[47] Wikipedia - MySQL https://fr.wikipedia.org/wiki/MySQL (dernière consultation le Avril 09, 2024)

[48] Wikipedia - Representationaal state transfer https://fr.wikipedia.org/wiki/Representational_state_transfer (dernière consultation le Mars 10, 2024)

[49] Wikipedia - TypeScript https://fr.wikipedia.org/wiki/TypeScript (dernière consultation le Mars 15, 2024)