Développement d'une application web en Java : Apprentissage et mise en pratique

Sarah KEGHIAN
MPCI-1A

Encadrant : François BRUCKER,
Laboratoire d'Informatique et des Systèmes (LIS)

juin-juillet 2024

Remerciements

Je tiens d'abord à remercier mon encadrant de stage, François Brucker, enseignant-chercheur au LIS, pour ses conseils et ses explications m'ayant permis de comprendre des notions importantes ainsi que de faire des décisions plus informées concernant l'évolution de ce stage. Merci également pour le temps passé à suivre régulièrement l'avancée du stage.

Je tiens aussi à remercier Damien Bourdette, directeur de projet et développeur au sein du groupe M6, pour avoir présenté son métier ainsi que d'avoir répondu aux questions concernant mon stage.

Enfin, merci à Arnaud Serres, stagiaire et co-créateur du projet final de ce stage pour son travail, sa disponibilité et son engagement dans le projet.

Table des matières

Pour assurer la qualité rédactionnelle de ce rapport, ChatGPT a été utilisé.

Glossaire

SQL (Structured Query Language) : Langage de programmation conçu spécifiquement pour interagir avec une base de données en vue de réaliser diverses opérations sur les données qu'elle contient.

Logique métier : Ensemble des règles et processus qui définissent comment traiter les données pour répondre aux besoins spécifiques de l'application.

Injection de dépendances : Méthode qui consiste à fournir les objets nécessaires à une classe depuis l'extérieur plutôt que de les créer à l'intérieur.

Stubbing : Technique utilisée dans les tests unitaires pour remplacer le comportement réel d'une méthode ou d'une fonction par un comportement simulé. Cela permet de tester une partie spécifique du code sans dépendre des autres composants.

Arbres B : Structure de données équilibrée utilisée pour organiser et accéder efficacement à de grandes quantités de données. Il est particulièrement utile dans les systèmes de gestion de bases de données et les systèmes de fichiers et garantie des recherches en temps logarithmique.

ResponseEntity : Classe générique de java qui prend deux arguments : le type de l'objet renvoyé en tant que corps de la réponse et un code de statut HTTP.

API (Application Programming Interface) : Ensemble de règles et de définitions qui permet à différentes applications ou services de communiquer entre eux.

1 Introduction

Une application web est un logiciel accessible via un navigateur internet. Dans le développement d'une application web, on distingue dans deux aspects : le front-end et le back-end. Le front-end correspond à la partie visible de l'application avec laquelle l'utilisateur interagit. Le back-end est la partie qui gère entièrement ou en partie la logique de l'application, c'est-à-dire son fonctionnement interne, ainsi que les interactions avec la base de données. Ces deux aspects, ayant des besoins différents, nécessitent souvent des langages de programmations et des compétences différentes. Ce stage porte sur l'apprentissage du développement back-end.

Le langage de programmation orienté objet nommé Java est fréquemment utilisé pour le développement back-end, notamment pour les applications de grande envergure. En effet, ce langage est connu pour sa fiabilité grâce sa vérification stricte des types, à sa gestion des exceptions et à sa gestion automatique de la mémoire qui réduisent les risques de bug et facilitent la maintenance à long terme des applications. Les versions LTS (Long-Term Support) de Java permettent aussi la stabilité à long terme d'une application en garantissant des mises à jour qui durent sur une longue période. De plus, il existe un grand écosystème de bibliothèques et de frameworks qui permettent d'accélérer développement web, notamment en répondant de manière optimisée à des problématiques courantes.

Cependant, le développement avec Java présente certains inconvénients, particulièrement pour les petites applications. La verbosité du langage, la consommation élevée de mémoire et le démarrage parfois lent des applications, en particulier lorsqu'elles utilisent des frameworks lourds, peuvent poser des problèmes. Ces aspects peuvent être moins adaptés aux projets de petite taille.

Comment optimiser l'apprentissage de Java et Spring Boot pour le développement efficace d'une application web, et quels sont les résultats de cet apprentissage ? Les objectifs de ce stage sont dans un premier temps, de proposer et d'analyser ma démarche d'apprentissage de Java ainsi que du framework de développement back-end Spring Boot. Puis dans un second temps, de détailler la mise en pratique de cet apprentissage dans la réalisation d'une application web en collaboration avec un stagiaire.

2 Présentation de l'organisme d'accueil

Le Laboratoire d'Informatique et des Systèmes (LIS) couvre divers domaines de recherche, organisés en quatre pôles : Calcul, Science des données, Analyse et Contrôle des systèmes, ainsi que Signal et Image. Le LIS se consacre à la fois à la recherche fondamentale dans ces domaines et à l'application concrète de ces connaissances dans divers secteurs industriels, notamment le transport, la santé, l'environnement, et la défense.

Le LIS est fortement connecté au milieu socio-économique grâce à son implication dans divers pôles de compétitivité comme le Pôle Mer, le Pôle SCS, et le Pôle Eurobiomed. Il est aussi membre de l'institut Carnot STAR et participe à des initiatives nationales telles que l'Institut Langage, Communication et Cerveau (ILCB) et le Centre Turing des Systèmes Vivants (Centuri).

De plus, les chercheurs et enseignants-chercheurs du LIS sont impliqués dans la formation des étudiants à l'informatique, notamment avec la Faculté d'Aix-Marseille et avec l'école Centrale Méditerranée, partenaire du LIS. C'est donc dans cette démarche d'éducation des étudiants que s'inscrit ce stage.

3 Apprentissage des outils nécessaires à la réalisation du projet

3.1 Apprentissage de Java

3.1.1 Caractéristiques et utilisations courantes de Java

Java est un langage de programmation orienté objet, développé par Sun Microsystems en 1995.

Les caractéristiques qui ont rendu le langage Java populaire sont nombreuses et variées. En voici quelques-unes :

Portabilité : Comme le suggère la devise "Write Once, Run Anywhere", Java est conçu pour être portable entre différentes plateformes. Le code Java est compilé en bytecode, qui est indépendant du système d'exploitation et du matériel. Ainsi, un fichier .java est compilé en fichier .class qui peut être exécuté sur diverses plateformes, à condition que chacune dispose d'une Machine Virtuelle Java (JVM). La JVM est responsable de la traduction du bytecode en code machine adapté au système d'exploitation et à l'architecture matérielle, comme le processeur. Par exemple, une application Java Web développée sur Windows peut être déployée et exécutée sur un serveur Linux sans modification du code. Cette caractéristique rend Java particulièrement adapté à ce stage.

Orienté Objet : Java utilise les concepts de la POO (Programmation Orientée Objet) comme l'encapsulation, l'héritage et le polymorphisme. Cela facilite la modularité, la réutilisation du code et une meilleure organisation du code. De plus, étant déjà familiarisée avec ces concepts, je pourrai apprendre Java plus rapidement, ce qui constitue un avantage considérable pour ce stage.

Performances : La JVM utilise la compilation Just-in-Time (JIT) pour optimiser le bytecode en code machine pendant l'exécution, ce qui améliore la performance des applications Java. Cela assure que le langage Java est performant pour le projet effectué pendant le stage.

Évolution continue : Java est maintenu et standardisé par Oracle et la communauté Java (via le Java Community Process), ce qui garantit des mises à jour régulières et une compatibilité à long terme. Ainsi, l'apprentissage de ce langage sera bénéfique, car ses évolutions régulières assureront sa pertinence pour l'avenir.

Simplicité et Clarté : Java conserve une syntaxe inspirée de C++, mais avec des simplifications qui éliminent certains des aspects les plus complexes et potentiellement dangereux, comme la gestion manuelle de la mémoire. Cette approche facilite l'apprentissage pour les nouveaux développeurs. Avec le temps limité dont je dispose pour apprendre un nouveau langage, cette simplicité fait de Java un choix idéal pour ce stage.

Les multiples domaines d'application de Java soulignent la polyvalence du langage. En effet, les applications Web bénéficient de Java pour sa stabilité et ses frameworks puissants comme Spring et Hibernate. Les applications mobiles, en particulier sur la plateforme Android, sont souvent développées en Java en raison de son intégration native avec le système Android. Les applications d'entreprise exploitent Java pour ses capacités à gérer des systèmes complexes et des transactions grâce à des technologies comme Java EE (Enterprise Edition). Enfin, Java est utilisé dans les systèmes embarqués et les applications serveur pour sa portabilité, sa sécurité, et ses performances fiables.

Dans ce contexte, le développement d'une application web, objectif principal de mon stage, représente une opportunité idéale pour appliquer Java de manière pratique. Cette approche assure également que les compétences acquises seront transférables à divers autres domaines d'application du langage.

3.1.2 Acquisition des bases de Java

Au cours de mon apprentissage du langage Java, j'ai suivi une série d'étapes structurées pour maîtriser les différents aspects de ce langage orienté objet. Mon approche a consisté à combiner des tutoriels fournis par Oracle, des exercices pratiques, et des projets que j'avais auparavant effectué en python. Le but final étant d'acquérir une compréhension de Java, de sa syntaxe et de ses fonctionnalités spécifiques.

Les tutoriels officiels d'Oracle, axés sur Java 8, introduisent des concepts tels que les expressions lambda [11] et les streams [10], qui ont marqué une transition majeure vers un Java plus moderne et fonctionnel. Java 8 a effectivement apporté des changements significatifs, modernisant le langage avec des fonctionnalités avancées. Cependant, Java a continué à évoluer jusqu'à la version actuelle, Java 22. Malgré ces évolutions, les tutoriels sur Java 8 restent pertinents, car ils couvrent les concepts fondamentaux du langage qui n'ont pas changé de manière significative. Ces bases sont essentielles pour comprendre le langage, même si des fonctionnalités avancées ont été ajoutées dans les versions ultérieures.

Découverte du Fonctionnement de Java : J'ai commencé par me familiariser avec les fondamentaux du fonctionnement de Java, notamment le rôle du compilateur et de la Java Virtual Machine (JVM). Cette compréhension de l'infrastructure sous-jacente est essentielle pour une maîtrise complète du langage. Les tutoriels Oracle ont fourni une introduction claire sur la manière de coder dans un fichier texte, puis de compiler et d'exécuter des programmes Java via le terminal [7]. Cette première étape m'a permis de saisir les bases du processus de compilation et d'exécution en Java.

Apprentissage des Bases du Langage J'ai poursuivi mon apprentissage en suivant les tutoriels Oracle sur les bases du langage Java. Étant donné mon expérience préalable avec Python, j'ai trouvé que la traduction de la syntaxe Python en Java était relativement directe. Cependant, j'ai dû accorder une attention particulière aux types primitifs de données et aux types de référence propres à Java. Des exercices pratiques réalisés en utilisant le terminal m'ont permis de renforcer ces concepts de manière concrète.

Maîtrise des Classes, Objets et Annotations La phase suivante de mon apprentissage a porté sur les concepts de classes, d'objets et d'annotations en Java. Bien que la programmation orientée objet (POO) soit un domaine déjà connu grâce à ma formation antérieure, les tutoriels Oracle m'ont permis de approfondir ma compréhension des spécificités de Java. J'ai ainsi appris les différences de syntaxe et les particularités de la POO en Java, notamment les types de classes tels que les interfaces et les énumérations.

Un aspect essentiel et distinctif de Java, par rapport à Python, est le contrôle d'accès aux variables et méthodes. En Java, ce contrôle est géré par les modificateurs d'accès, qui déterminent la visibilité des éléments de la classe. Par exemple, les modificateurs public, protected et private permettent de définir si une variable ou une méthode est accessible uniquement au sein de la même classe, par les classes du même package, ou par toutes les autres classes. Cette fonctionnalité offre un niveau de protection des données et une encapsulation plus stricts, favorisant ainsi une meilleure gestion de l'accès aux composants d'une classe.

Utilisation d'IntelliJ et Gradle Pour développer des compétences pratiques, j'ai appris à utiliser IntelliJ IDEA comme environnement de développement intégré (IDE) et Gradle comme outil de gestion de build, en y intégrant JUnit pour les tests. Cette phase m'a permis de m'acclimater à des outils essentiels pour le développement en Java, facilitant ainsi la gestion des projets et des dépendances. J'ai également décidé à partir d'ici d'utiliser Java 21, car c'est la version LTS (Long Term Support) la plus récente.

Exploration des Concepts Avancés J'ai approfondi mes connaissances en suivant les tutoriels Oracle sur les interfaces, les packages, et les principes de tri en Java. J'ai étudié l'utilisation des interfaces pour définir des contrats de comportement, ainsi que les concepts de tri en utilisant Comparator et Comparable [8, 9]. De plus, les concepts de packages, d'héritage, de visibilité, et de méthodes surchargées ont été examinés.

Réalisation de Projets Pratiques Pour mettre en pratique mes connaissances, j'ai réalisé plusieurs projets. Le premier projet a été un projet de création de dés très simple, mais qui m'a permis de bien intégrer la syntaxe Java (cf. annexe : 5.). Ensuite, j'ai travaillé sur un projet de cartes, qui m'a amené à me renseigner sur les interfaces Comparable et Comparator [8, 9] qui m'ont permis, dans ce projet, de trier les cartes selon des règles établies. Ce projet m'a également fait appliquer mes apprentissages sur les énumérations et la surcharge de méthodes (cf. annexe : 5.).

Pour compléter ma formation, j'ai réalisé un projet de bataille navale, qui a mis en œuvre des concepts avancés tels que la surcharge de méthodes, les énumérations, et l'héritage. Ce projet a également impliqué une réflexion approfondie sur l'organisation en packages pour une gestion efficace du code.

Parallèlement, j'ai exploré des fonctionnalités spécifiques telles que les HashMaps (dictionnaires), les records, et les principes de l'interface Stream pour la gestion des flux de données, en utilisant des tutoriels Youtubes (cf. annexe : 1.). Les tutoriels Oracle sur les génériques, ont été des ressources précieuses pour comprendre et utiliser ces concepts.

Après avoir terminé ces tutoriels, j'ai préparé un retour pour mon encadrant, en mettant en avant les aspects essentiels à retenir et ceux qui pourraient être omis, afin d'optimiser les futurs apprentissages (cf. annexe : 1.). J'ai également précisé les sources complémentaires qui m'ont été utiles à l'apprentissage de Java, Spring Boot et IntelliJ IDEA. De plus, j'ai fourni une correction détaillée des QCM du GEI sur Java (cf. annexe : 2.), ce qui m'a permis d'identifier certaines attentes et objectifs d'un apprentissage de ce langage, et de constater qu'ils étaient globalement atteints.

J'ai constaté que la structure fournie par les tutoriels d'Oracle a grandement bénéficié à mon apprentissage. Bien qu'ils soient parfois longs et détaillés, voire excessivement détaillés, ils ont offert une base solide pour comprendre Java. De plus, il est toujours possible de passer les parties qui ne nous sont pas utiles au moment de l'apprentissage. En parallèle, les projets pratiques ont joué un rôle crucial en permettant de concrétiser et d'ancrer ces connaissances dans des contextes réels. Cependant si je pouvais recommencer, je ferais des changements à mon apprentissage initial.

Je commencerais par intégrer l'étude de l'organisation en packages en même temps que l'exploration des concepts de classes et d'objets. Cela permettrait de mieux structurer les projets dès le début et d'améliorer la gestion et la modularité du code.

De plus, je mettrais en œuvre des projets ou des programmes courts immédiatement après chaque leçon pour appliquer les concepts appris. Cette approche pratique renforcerait ma compréhension et mon application des notions. Je veillerais également à inclure un projet spécifique sur les interfaces, afin de mieux appréhender et appliquer ce concept essentiel.

Enfin, je commencerais l'étude des Generics plus tôt, idéalement lors du projet de cartes. Cela me permettrait de comprendre et d'appliquer les Generics dès le début, facilitant ainsi la gestion des types de données dans des projets plus complexes.

En ajustant ces aspects, je pourrais améliorer l'efficacité de mon apprentissage et la pratique des concepts de Java.

Voyons maintenant comment l'apprentissage de Spring Boot a différé ou ressemblé à celui de Java.

3.2 Développer une application web avec Spring Boot

Comme mentionné précédemment, Java offre une multitude d'outils pour faciliter le développement web. Parmi eux, Spring Boot se distingue comme un framework open-source basé sur le framework Java Spring. Il a été développé pour simplifier le processus de développement des applications Java.

Les caractéristiques de Spring Boot qui en font un choix intéressant pour ce projet sont :

— L'Auto-configuration : Spring Boot configure automatiquement votre application en fonction des dépendances ajoutées. Par exemple, en ajoutant une dépendance de base de données, Spring Boot configurera automatiquement une connexion à la base de données.

— Emballage autonome : Il permet de créer des applications autonomes avec un serveur embarqué (comme Tomcat), ce qui signifie qu'il n'y a pas besoin de déployer l'application dans un serveur séparé. C'est le serveur embarqué qui gère la communication avec le client. Cette fonctionnalité est indispensable pour ce stage puisque nous n'avons pas de serveur à disposition pour héberger l'application. L'application est ainsi contenue dans un fichier JAR prêt à l'exécution.

— Starter POMs : Spring Boot fournit des "starters" qui sont des ensembles de dépendances préconfigurées pour différentes fonctionnalités (web, JPA, sécurité, etc.), ce qui simplifie encore la configuration.

Pour comprendre comment ces caractéristiques sont mises en œuvre, examinons l'architecture typique d'une application Spring Boot.

3.2.1 Architecture d'une application web Spring Boot

Cette architecture se décompose en trois couches, la couche de persistance , la couche de service et la couche de présentation (ou controller sur Spring Boot).

La couche de persistance est chargée de la gestion des interactions avec la base de données, notamment les opérations CRUD (Create, Read, `update`, Delete) pour les entités de l'application. Cette gestion peut être implémentée en suivant différents design patterns, le plus couramment utilisé étant le pattern Repository. Le Repository fournit des méthodes permettant d'effectuer ces opérations dans le langage de programmation utilisé, sans nécessiter l'utilisation directe du **SQL**. Ainsi, lorsque des opérations plus complexes sur les données sont nécessaires au niveau de la couche de service, il est possible de les réaliser sans écrire de requêtes **SQL** explicites, en utilisant simplement les méthodes exposées par le Repository. Cette approche favorise l'abstraction des détails de la base de données et améliore la maintenabilité du code. Avec Spring Boot, la gestion des opérations de persistance est simplifiée grâce à l'interface JpaRepository, qui inclut déjà des méthodes prédéfinies pour ces opérations courantes.

La couche service est responsable de la **logique métier** de l'application, elle utilise les méthodes fournies par les Repository pour accomplir les opérations nécessaires à la réalisation des fonctionnalités de l'application. Elle agit comme une interface entre la couche de persistance et la couche de présentation. Cela permet une séparation claire des responsabilités, facilite les tests unitaires et améliore la maintenabilité du code en centralisant la **logique métier**.

Enfin, la couche de présentation a pour but de récupérer les données traitées et les résultats fournis par la couche service pour les apporter au front-end. Sur Spring Boot, la couche de présentation est typiquement représentée par les classes controller, les @RestController sont utilisés pour exposer des données au format JSON, qui sont ensuite consommées par le frontend (comme des applications JavaScript). En revanche, les @Controllers gèrent les requêtes des utilisateurs pour les pages web. Ils renvoient des pages web complètes (HTML) ou des fichiers statiques (comme JavaScript, CSS) au client pour affichage dans un navigateur.

Avec cette compréhension de l'architecture d'une application Spring Boot, je vais maintenant détailler la démarche que j'ai suivie pour apprendre et appliquer Spring Boot dans le contexte de mon projet.

3.2.2 Apprentissage des bases de Spring Boot

Ma démarche d'apprentissage pour Spring Boot a été guidée par les besoins de mon projet.

J'ai d'abord appris à démarrer un projet en utilisant le Spring Initializr et à ajouter des dépendances nécessaires. Par la suite, je me suis intéressée à la construction d'applications web avec Spring Boot. Bien que le tutoriel officiel [15] introduise les @Controller et fournisse des informations sur le sens de annotations, il manque d'explications plus générales et reste donc limité pour la création d'une application pleinement fonctionnelle.

Pour approfondir mes connaissances et acquérir des compétences pratiques, j'ai exploré plusieurs tutoriels YouTube [12, 3]. Ces ressources m'ont permis de créer des applications web simples. Leur approche pédagogique, à la fois théorique et concrète, a été très bénéfique. J'ai ainsi pu comprendre des concepts clés comme l'**injection de dépendances** [16, 3] (34 :52-49 :39) et l'architecture couramment utilisée dans les applications web (cf. section 3.2.1) [3] (29 :55-33 :14).

Après avoir assimilé les aspects théoriques, j'ai suivi plusieurs tutoriels pratiques visant à développer une application web CRUD fonctionnelle. Le premier projet [12] (cf. annexe : 7.) consistait à créer une application de bibliothèque permettant de stocker des livres avec leur titre et leur auteur. Ce projet m'a permis d'apprendre à connecter une base de données à l'application et à utiliser les méthodes couramment employées du Repository. J'y ai également appris l'utilisation de la classe @RestController pour échanger des données au format JSON avec le front-end, ainsi que l'application des annotations @GetMapping, @PostMapping, et @DeleteMapping selon les types de requêtes HTTP (GET, POST, DELETE, etc.). De plus, j'ai pu appréhender une première approche de la gestion des erreurs côté front-end en renvoyant des réponses HTTP appropriées, telles que l'erreur 404 (not found) lorsque le livre n'est pas trouvé, ou la réponse 200 OK lorsque la requête est réussie par exemple. Enfin, j'ai simulé l'utilisation de cette application grâce au logiciel Postman. Cependant, ce tutoriel ne mettait pas en oeuvre l'architecture complète car il n'utilisait pas de couche service. J'ai donc regardé d'autres tutoriels pour comprendre cela [2, 4].

Par la suite, en prévision de la réalisation de mon projet final, un jeu de dés, j'ai vérifié ma compréhension en créant une application CRUD permettant de stocker des dés et leurs positions (cf. annexe : 7.). Ce projet m'a également permis d'apprendre à effectuer des tests unitaires sur la couche service de l'application en utilisant la bibliothèque Mockito, qui repose sur le principe du **stubbing** [13, 14].

Pour compléter cet apprentissage initial, j'ai développé une application testée intégrant des fonctionnalités au-delà des simples opérations CRUD (cf. annexe : 7.). J'ai ainsi conçu une application permettant de stocker des noms de joueurs : si un joueur n'existe pas, il est ajouté, et s'il existe déjà, l'application renvoie une erreur 400 - Bad Request

Mon apprentissage initial s'est arrêté à ce stade, mais j'ai poursuivi l'approfondissement de mes compétences en Spring Boot en travaillant sur le projet final (cf. section 4), notamment en apprenant à servir les fichiers statiques du front-end (cf. section 4.3.3).

On peut remarquer que mon apprentissage de Spring Boot est plus spécifique que celui de Java, car il a été orienté par les besoins particuliers de mon projet. Il est donc possible que certaines connaissances aient été abordées de manière moins approfondie.

4 Projet d'application web : jeu du Décathlon

Les dépendances Spring Boot choisies pour cette application sont :

- Spring starter Web : starter Spring Boot pour développer des applications web, incluant Tomcat comme serveur embarqué
- Spring Starter JPA : starter Spring Boot facilitant l'intégration de la persistance des données (processus de sauvegarde des données de manière durable) avec JPA, fournissant une configuration automatique pour les bases de données.
- H2 Database : base de donnée en fichier ou en mémoire, gratuite et opensource demandant une configuration minimale [5]
- Lombok : bibliothèque Java permettant de réduire le code répétitif

Le Décathlon (cf. règles [6]) est un jeu de dés composé de plusieurs mini-jeux. Cette application web

sera dédiée au mini-jeu du saut en longueur.

Cependant, pour adapter une expérience de jeu multijoueur à une application web, il est nécessaire de modifier certains aspects pour maintenir le principe de concurrence entre les joueurs. Nous avons donc décidé de mettre en place un classement des meilleurs joueurs en fonction de leurs scores, lequel sera consultable après chaque partie. Cet ajout nécessite l'utilisation d'une base de données.

4.1 Base de données de l'application

4.1.1 Théorie rudimentaire sur les bases de données

Les bases de données sont des structures conçues pour stocker des informations de manière organisée. Elles sont typiquement structurées sous forme de tables, où chaque table représente un type d'entité. En programmation orientée objet, cette notion d'entité se traduit par une classe et les colonnes ou champs de ces tables correspondent alors aux attributs de la classe. Chaque ligne de la table décrit alors une instance différente de la classe. Pour différencier et accéder facilement à chaque ligne, une clé primaire est associée à chaque enregistrement. Cette clé (ou id) est unique et est généralement un entier et de type Long en java. Voici une table de donnée stockant des joueurs (Figure 1).

SELECT * FROM JOUEUR;

ID	NOM
1	sarah
33	Joueur
34	test
35	Simon
36	moi
37	salomé

FIGURE 1 – Table de données de Partie

On observe bien une colonne contenant les clés primaires (ID) et une colonne contenant une information sur le joueur, son nom.

Les tables peuvent être stockées soit dans des fichiers sur disque local ou sur un serveur dédié, soit dans la mémoire vive, en fonction du type de base de données utilisée. L'avantage du stockage en mémoire vive est que l'accès aux données est très rapide, mais cela présente l'inconvénient que si l'application plante ou doit redémarrer, les données seront effacées. Comme cette application est destinée à un entraînement et n'a pas pour but de fonctionner en permanence, la base de donnée sera de type "in-file", c'est-à-dire que les données seront stockées dans un fichier sur le disque dur du PC hôte.

Avec cette compréhension des concepts fondamentaux des bases de données, nous pouvons maintenant examiner comment ces principes ont été appliqués dans la conception de la base de données pour notre jeu du Décathlon.

4.1.2 Implémention de la base de donnée pour les besoins du jeu

Pour déterminer quelles informations stocker, considérons le déroulement d'une partie. Tout d'abord, il est nécessaire d'enregistrer le joueur qui a lancé le jeu. Ce joueur est associé à une ou plusieurs parties, chaque partie étant elle-même associée à trois essais. Les résultats de ces essais sont déterminés par les lancers de dés. Nous pouvons donc prévoir de stocker les données suivantes : Joueur, Partie, Essais, et Dé.
Cependant, la création de l'entité Essais peut être discutable, car il n'est pas strictement nécessaire de conserver les scores des trois essais. Cependant, si l'on souhaite établir un historique plus détaillé des parties ou réaliser des statistiques sur les performances des joueurs, il pourrait être pertinent de conserver ces informations. De plus, la segmentation du code en créant des entités distinctes comme "Essais" facilite les tests unitaires et la maintenance du code. En encapsulant les essais dans une entité distincte,

la complexité de la classe Partie est réduite. En revanche, l'ajout d'une nouvelle entité peut aussi entraîner une complexité supplémentaire dans la gestion de la base de données et des relations entre les entités.

De manière générale, il est judicieux de minimiser le nombre de requêtes entre le front-end et le back-end pour améliorer les performances de l'application. Pour ce projet, où le front-end doit recevoir plusieurs dés à chaque partie, une solution efficace est de créer une entité GroupeDés. Cette entité représente un ensemble de 5 dés, chaque dé étant associé à un même identifiant de groupe (idGroupe). Ce regroupement permet de rattacher plusieurs dés à un groupe, optimisant ainsi les échanges de données entre le back-end et le front-end.

Une question primordiale concernant l'implémentation de la base de données est : comment faire le lien entre les données ? Une réponse à cette question peut être la modélisation relationnelle. Elle repose sur l'utilisation de clés primaires et étrangères pour structurer les données et leurs relations. Les clés primaires identifient de manière unique chaque enregistrement dans une table, tandis que les clés étrangères établissent des liens entre les tables en faisant référence aux clés primaires d'autres tables. Les schémas UML (tableaux 1, 2, 3, 4, et 5) illustrent ce concept, on observe que la classe Essais est associée à la classe Partie par une clé étrangère idPartie, et la classe Partie est associée à la classe Joueur par une clé étrangère idJoueur. Ce modèle relationnel assure une structure claire et cohérente des données, permettant des opérations efficaces et une gestion simplifiée des relations entre les entités. ce schéma permet également d'envoyer des groupes d'informations, comme les dés, dans une seule requête, économisant des ressources et réduisant le temps de transfert des données.

De
- id : Long - idGroupe : Long - position : Integer - frozen : boolean
+ noArgsConstructor() : De + allArgsConstructor() : De + setter() : void + getter() : De + equals() : boolean + hashcode() : int

Table 1 – Schéma UML de la classe De

GroupeDes
- id : long - listeDes : list<Long>
+ noArgsConstructor() : GroupeDes + allArgsConstructor() : GroupeDes + setter() : void + getter() : GroupeDes + equals() : boolean + hashcode() : int

Table 2 – Schéma UML de la classe GroupeDes

Pour gérer le comportement différencié des dés en fonction de leur état de gel, plusieurs approches ont été envisagées. Initialement, il a été proposé d'ajouter un attribut booléen nommé `frozen` à la classe `De`. Cet attribut indiquerait si le dé est figé ou non. Lorsqu'un dé est marqué comme figé, il ne changera pas de position lors du lancement du groupe de dés.

Avant de mettre en œuvre cette solution, d'autres possibilités ont été explorées. Une option aurait été de créer une table distincte pour stocker les dés gelés, nécessitant ainsi une opération supplémentaire pour vérifier si un dé appartient à cette liste avant chaque lancer. Cette approche ajouterait une complexité supplémentaire au code en introduisant des vérifications constantes et en augmentant le nombre de tables dans la base de données.

Une autre option envisagée était la création d'une sous-classe `DeFreeze` héritant de la classe `De`, avec une méthode `lancer()` différente. Cependant, bien que cette solution soit théoriquement valable, elle introduirait une complexité additionnelle pour une différence qui pourrait être gérée de manière plus simple.

Ainsi, après avoir examiné ces alternatives, il a été décidé de maintenir l'approche initiale avec l'attribut `frozen`. Cette solution est plus simple à implémenter et à maintenir, tout en répondant efficacement aux besoins fonctionnels du projet.

Enfin, dans l'application, l'accès aux données sera géré par des classes d'interface qui étendent `JpaRepository`. Ces interfaces fournissent des méthodes permettant d'effectuer diverses opérations sur les données, telles que les opérations CRUD.

Essais
- id : Long - idPartie : Long - score1 : Integer - score2 : Integer - score3 : Integer
+ noArgsConstructor() : Essais + allArgsConstructor() : Essais + setter() : void + getter() : Essais + equals() : boolean + hashcode() : int

TABLE 3 – Schéma UML de la classe Essais

Partie
- id : Long - idJoueur : Long - scoreFinal : Integer
+ noArgsConstructor() : Partie + allArgsConstructor() : Partie + setter() : void + getter() : Partie + equals() : boolean + hashcode() : int

TABLE 4 – Schéma UML de la classe Partie

Joueur
- id : Long - nom : String
+ noArgsConstructor() : Joueur + allArgsConstructor() : Joueur + setter() : void + getter() : Joueur + equals() : boolean + hashcode() : int

TABLE 5 – Schéma UML de la classe Joueur

Avec la structure de la base de données en place, nous pouvons maintenant aborder la gestion des données via la couche service. Nous verrons comment la **logique métier** est intégrée dans cette couche et comment les opérations CRUD sont mises en œuvre pour optimiser les interactions entre le front-end et le back-end.

4.2 Création de la couche service : logique métier et choix de conceptions

Afin de fluidifier le jeu et d'améliorer l'expérience utilisateur, une grande partie de la **logique métier** du jeu sera gérée côté front-end à l'aide de JavaScript. Cela permettra de réduire la charge sur le serveur et d'offrir une meilleure réactivité en traitant les interactions des utilisateurs directement dans le navigateur, sans nécessiter des allers-retours constants avec le back-end. En conséquence, la **logique métier** sera moins présente dans la couche service du back-end, ce qui simplifie la gestion du côté serveur.

A chaque entité de la base de donnée est associée une classe service à part entière. J'ai d'abord créé les méthodes pour les opérations CRUD de chaque classe.

4.2.1 Méthodes `create`

La méthode `create` repose principalement sur l'utilisation de la méthode `save()` du `JpaRepository`, qui permet d'enregistrer un objet dans la base de données. Cependant, pour certaines entités, la création nécessite de récupérer des informations supplémentaires.

Afin de réduire le nombre de requêtes effectuées par le front-end, j'ai décidé que les méthodes `create` non seulement créent l'objet, mais le renvoient également en réponse à la méthode.

De : Pour créer un objet `De`, la méthode prend en argument l'identifiant du `GroupeDes` associé, lequel est ensuite enregistré dans l'attribut `idGroupe`. Par ailleurs, l'attribut `frozen` est initialisé à `False`. Enfin, l'objet `De` est enregistré dans la base de données via la méthode `save()`.

GroupeDes : La création d'un `GroupeDes` nécessite la génération de plusieurs objets `De` et la récupération de leurs identifiants (ID). Initialement, j'avais envisagé de parcourir la liste des dés existants pour

identifier ceux appartenant au `GroupeDes` en cours de création en fonction de leur `idGroupe`. Cependant, j'ai rapidement réalisé que cette approche augmentait inutilement la complexité du code, puisqu'elle nécessitait de parcourir l'ensemble des dés déjà créés.

Après avoir révisé cette approche, j'ai opté pour une méthode plus efficace : récupérer les identifiants des dés au fur et à mesure de leur création directement dans la méthode `createGroupeDes`, en utilisant la méthode `createDe()` du `DeService`. Cette solution rend superflu l'attribut `idGroupe` initialement prévu, que j'aurais dû supprimer pour simplifier et optimiser mon code. Ce choix améliore à la fois la lisibilité et l'efficacité du processus de création d'un `GroupeDes`.

Essais : Pour créer un objet `Essais`, il suffit de fournir l'identifiant de la partie en argument. Les trois scores sont initialisés à `-1` pour différencier les scores vides des scores nuls, ce qui est nécessaire pour le bon fonctionnement de la méthode `update` (cf. section 4.2.3) de cet objet. Enfin, l'objet est enregistré dans la base de données.

Partie : Pour créer un objet `Partie`, on fournit en argument l'identifiant du joueur, puis on initialise le score à `0`. Contrairement à `Essais`, il n'est pas nécessaire de mettre `-1` en score initial.

Joueur : Pour créer un objet `Joueur`, on fournit en argument le nom du joueur. Celui-ci est enregistré dans l'attribut `nom` à l'aide du `setter`, puis l'objet est enregistré dans la base de données.

4.2.2 Méthodes `read`

Les méthodes `read` renvoient simplement l'objet spécifié à partir de son identifiant en utilisant la méthode `findById()` du `JpaRepository`.

La seule exception est la méthode `read` de `Joueur`, qui, pour répondre aux besoins du front-end, renvoie l'objet `Joueur` à partir de la chaîne de caractères correspondant à son nom. Pour ce faire, la méthode parcourt la liste des joueurs dans la base de données et recherche celui dont le nom correspond. Cette méthode présente une complexité linéaire en fonction du nombre de joueurs dans la base de données car elle parcourt l'ensemble des enregistrements. Cela pourrait poser problème en cas d'utilisation fréquente (ce qui n'est pas le cas dans ce projet) ou si le nombre de joueurs devenait très important. Alors, j'aurais la possibilité d'indexer la colonne `nom` dans la base de donnée pour que la recherche soit gérée par la base de donnée. Grâce à l'utilisation d'**arbres B**, la complexité de recherche serait alors logarithmique.

Cependant, un point d'ombre subsiste : le jeu, en l'état, ne prend pas en compte la possibilité que deux joueurs aient le même nom. En effet, Si un joueur saisit un nom déjà existant, il sera assimilé au joueur ayant précédemment enregistré ce même nom. Une solution pour traiter ce problème, mais que nous n'avons pas eu le temps de mettre en place, est discutée dans la conclusion (cf. section 5).

4.2.3 Méthodes `update`

Les méthodes udpate peuvent avoir des fonctionnalités très différentes en fonctions de l'objet auquel elle s'applique.

De : L'`update` peut se faire par 3 méthodes différentes qui mettent à jour les attributs et prennent en argument l'identifiant du dé.
La méthode `throwDe()`, modifie l'attribut `position` de l'objet en lui assignant un entier aléatoire entre 1 et 6, simulant un lancer de dé si le dé n'est pas figé.
La méthode `freezeDe()` fixe l'attribut `frozen` à `True`, ce qui signifie que le dé est figé et ne sera pas modifié lors des prochains lancers.
La méthode `unfreezeDe()` le fixe à `False`, permettant ainsi au dé de participer à nouveau aux lancers et d'être modifié si le joueur change d'avis sur le dé à figer.
J'ai envisagé la création d'une seule méthode `freezeUnfreezeDe()` pour alterner la valeur de l'attribut `frozen`, mais j'ai finalement opté pour des méthodes séparées. Cette approche offre une plus grande clarté, car chaque méthode a une fonctionnalité spécifique clairement définie. De plus, en cas d'erreur ou de situation particulière, comme la tentative de figer un dé déjà figé, il est préférable d'avoir des méthodes distinctes pour éviter des effets inattendus ou des erreurs difficiles à diagnostiquer.

GroupeDes : Les méthodes d'`update` pour GroupeDes sont similaires à celles utilisées pour De. La méthode `throwGroupe()` prend en argument l'identifiant du groupe de dés et applique la méthode `throwDe()` à chaque dé du groupe. Ainsi, tous les dés non figés seront lancés.
Les méthodes `freezeDeGroupe()` et `unfreezeDeGroupe()` prennent deux arguments : l'identifiant du groupe et l'identifiant du dé à modifier. Elles permettent respectivement de figer ou de dégeler un dé spécifique au sein du groupe. La mise en place de ces méthodes est discutable car elle présente des avantages et inconvénients. D'un côté, elles demandent deux arguments, ce qui peut compliquer les appels aux méthodes et introduire des risques d'erreurs. Mais d'un autre, elles permettent de centraliser la gestion des états des dés dans une seule classe de service, simplifiant ainsi l'interface pour le front-end.

Essais : La méthode `update` pour Essais, `addEssai()` prend en argument l'identifiant de l'objet Essais et un score sous forme d'entier. Elle identifie le premier emplacement de score disponible (score1, score2, ou score3) dont la valeur est encore à -1, puis y enregistre le score et sauvegarde l'objet. Si tous les emplacements de score sont déjà remplis, l'objet est retourné sans modification. Une amélioration potentielle serait de renvoyer une exception lorsque tous les emplacements sont occupés.
Cette approche permet d'ajouter les scores progressivement au cours de la partie, nécessitant moins d'arguments que si l'on devait remplir les trois scores en une seule fois. Elle simplifie également l'interaction avec le front-end, qui n'a pas besoin de gérer les trois scores simultanément.

Partie : La méthode `addScoreFinalPartie()` est conçue pour mettre à jour le score final d'une partie. Elle prend en argument l'identifiant de la partie et le score final sous forme d'entier. La méthode utilise le setter de la classe Partie pour attribuer le score final à l'objet et sauvegarde ensuite ce dernier. Le score final est déterminé comme étant le maximum des trois scores des essais, une opération effectuée par le front-end avant l'appel de cette méthode.

Joueur : Dans cette application, la fonctionnalité de modification du nom d'un joueur n'est pas implémentée. Le nom d'un joueur est fixé lors de la création de l'objet et ne peut pas être modifié ultérieurement.

4.2.4 Méthodes `delete`

Les méthodes `delete` sont simples et utilisent la méthode `deleteById()` du JpaRepository, prenant en argument l'identifiant de l'objet à supprimer. Habituellement, ces méthodes ne renvoient rien, mais j'ai choisi de les faire renvoyer l'objet supprimé. Cette approche permet de conserver une trace des suppressions, ce qui est utile en cas d'erreur ou pour des besoins de journalisation.
Ces méthodes sont utilisées par le front-end à la fin de chaque partie pour supprimer les dés, les groupes de dés, et les essais, afin d'éviter l'accumulation de données obsolètes dans la base de données. En revanche, les classes PartieService et JoueurService ne disposent pas de méthodes de suppression. Cela est dû au fait que les entités Partie et Joueur sont conservées pour permettre le classement des parties et maintenir un historique complet des jeux, ce qui est essentiel pour le reporting et la cohérence des statistiques.

4.2.5 Méthodes particulières

Certaines méthodes ne correspondent pas aux opérations CRUD classiques. Voici un aperçu de ces méthodes spécifiques.

La méthode `classerParties()` est utilisée en interne pour trier les parties en fonction de leur score final, dans un ordre décroissant. Ce tri est réalisé en utilisant l'interface `Comparable<>` [8] de Java, à travers l'implémentation de la méthode `compareTo()` dans la classe `Partie`. La méthode `sort()` de l'interface Comparable est ensuite appliquée pour effectuer le tri.

La méthode `convertPartieToDto()` prend en argument l'identifiant d'une partie et utilise la méthode `classerParties()` pour déterminer et attribuer une place dans le classement à cette partie.

Ensuite, la méthode `convertToLeaderDto()` rassemble les informations nécessaires à l'affichage du classement. Elle prend en argument l'identifiant d'une partie et récupère le nom du joueur associé (à partir de l'identifiant du joueur et de la liste des instances de `Joueur`) ainsi que la position de la partie dans le classement.

Enfin, la méthode `getLeadersParties()` renvoie la liste des 15 meilleures parties, en affichant uniquement le nom du joueur et sa place dans le classement.

Je détaillerai plus loin l'utilisation du pattern DTO (Data Transfer Object) dans ce projet, tel que mentionné dans ces méthodes (cf. section 4.3).

4.3 Création de la couche controller et gestion des erreurs

Après avoir élaboré la couche service, qui gère la logique métier et les opérations CRUD pour chaque entité, il est essentiel de créer la couche controller pour exposer ces services à travers des endpoints accessibles via HTTP. Cette couche réceptionne les requêtes des clients, invoque les services correspondants et renvoie les réponses appropriées

4.3.1 Gestion des erreurs : classe Optional<>

Dans cette section, nous allons explorer comment j'ai mis en place une gestion efficace des erreurs pour garantir une communication fluide et fiable entre le client et le serveur.

Un objet de type `Optional<NomDeClasse>` est une classe contenue dans le framework Java qui sert à encapsuler un objet pouvant potentiellement être null. Cette approche permet de gérer de manière plus sûre les valeurs nulles, réduisant ainsi le risque d'erreurs comme les NullPointerException.

En d'autres termes, lorsqu'une méthode du JpaRepository renvoie un Optional<NomDeClasse>, cela signifie que le résultat de la requête peut soit contenir un objet de la classe spécifiée (si un objet correspondant à l'identifiant fourni est trouvé dans la base de données), soit être vide (si aucun objet correspondant n'est trouvé).

Cela permet de traiter plus explicitement les cas où un objet n'est pas trouvé, en forçant le développeur à vérifier la présence de l'objet avant de l'utiliser, et d'éviter ainsi des comportements inattendus liés aux valeurs nulles.

Dans un premier temps, j'avais envisagé de gérer les erreurs en utilisant des blocs `try-catch` pour capturer les exceptions, que je transmettais ensuite jusqu'au `Controller`, lequel renvoyait l'erreur au front-end. Par exemple, si le front-end fournissait un identifiant inexistant, le code levait une exception de type `IllegalArgumentException` avec le message `"Aucun Joueur n'a l'id : " + id`. Toutefois, cette approche présentait un inconvénient majeur : elle ne permettait pas de distinguer clairement les erreurs provenant du client de celles provenant du serveur. Ainsi, une telle erreur se manifestait par un code 500, suggérant à tort une responsabilité du serveur, alors que l'erreur était en réalité due à une mauvaise requête du client.

Pour résoudre ce problème, j'ai opté pour une autre stratégie en tirant parti des objets `Optional<>` renvoyés par les méthodes du `JpaRepository`. Désormais, les méthodes de la couche service qui nécessitent un identifiant renvoient un `Optional<nomDeClasse>`. La vérification de la présence ou non de l'objet se fait alors au niveau des méthodes du `RestController`. Si l'`Optional` est vide, la méthode renvoie une `ResponseEntity` avec le code HTTP 404 : Not Found, signalant ainsi que l'erreur provient d'une mauvaise requête du client. De cette manière, le front-end peut identifier clairement que l'erreur vient de son côté et non du serveur, facilitant ainsi le processus de débogage et de correction. Cette méthode de gestion des erreurs est simple et répond aux besoins de ce projet.

Cependant, pour une gestion des erreurs plus détaillée, il serait possible d'utiliser les annotations @ResponseStatus [1]. Cette approche permettrait de centraliser la gestion des erreurs en associant directement les exceptions à des codes de réponse HTTP spécifiques ce qui pourrait offrir une plus grande flexibilité et une meilleure séparation des responsabilités entre la **logique métier** et la gestion des erreurs dans le controller.

4.3.2 Pattern DTO

En parallèle avec la gestion des erreurs, l'utilisation du pattern Data Transfer Object (DTO) peut jouer un rôle important dans l'organisation des données échangées entre le front-end et le back-end. Le pattern Data Transfer Object (DTO) est un patron de conception utilisé pour transférer des données entre différentes couches d'une application, ou entre différentes applications. L'objectif principal des DTO est de regrouper des données en un seul objet afin de simplifier et d'optimiser le transfert de ces données, notamment lors des appels de services ou de la communication entre le front-end et le back-end. Le

pattern DTO permet également d'isoler les couches de l'application, ce qui facilite les modifications et les évolutions des données échangées sans impacter directement la **logique métier**.

Pour chaque entité de l'application, j'ai créé une version DTO correspondante, ce qui a nécessité la mise en place de méthodes convertObjetToDto() et convertDtoToObjet() pour chaque classe. Cependant, je réalise maintenant que cette approche a introduit une complexité supplémentaire, avec un code plus lourd et des tests supplémentaires, pour un bénéfice relativement limité.
Néanmoins, l'utilisation du pattern DTO s'est révélée justifiée pour l'objet LeaderDto. Ce DTO est spécialement conçu pour regrouper le nom du joueur et le score d'une partie en un seul objet, ce qui simplifie la transmission de ces informations au contrôleur et au front-end. L'emploi de LeaderDto s'avère pertinent, car il permet de structurer et de présenter efficacement le classement du jeu, facilitant ainsi l'affichage des résultats et la gestion des données au niveau du front-end.

4.3.3 Définition des chemins d'URL et gestion des ressources statiques

Avec une gestion efficace du transfert des données grâce aux DTOs, il est également crucial de définir clairement les chemins d'URL pour assurer une communication fluide entre le front-end et le back-end. La définition des chemins d'URL et la gestion des ressources statiques sont des aspects essentiels pour organiser l'architecture des routes de l'application.

Pour la gestion des routes de l'application (ou mapping), chaque méthode est associée à un chemin URL spécifique. Les URLs destinées aux échanges de données avec le front-end, via les classes `RestController`, commencent par `/api` (**Application Programming Interface**), suivies du nom de l'objet et du nom de la méthode. Par exemple, pour invoquer la méthode `createPartie()`, le front-end en JavaScript utilise l'URL suivante : `http://localhost:8080/api/Partie/create`.

En revanche, pour les ressources statiques telles que les pages web, les URLs débutent par `/static`.

Cette convention est à la fois standard et intuitive, favorisant la cohérence et la clarté du code pour les développeurs. De plus, elle permet une extension facile de l'application : l'ajout de nouvelles fonctionnalités peut se faire en suivant cette même structure, sans perturber les routes déjà définies.

La gestion des fichiers statiques dans Spring Boot est facilitée par la configuration d'une méthode dans une classe `Controller` qui retourne une chaîne de caractères `"redirect:/static/index.html"`. Le préfixe `redirect:` indique à Spring Boot que la réponse doit être une redirection HTTP. Lorsque cette méthode est invoquée, Spring Boot génère une réponse au client avec un code de statut HTTP 302 (Found) et l'en-tête `Location` pointant vers `/static/index.html`. Ce chemin représente l'emplacement du fichier à partir du répertoire `resources/static` dans le projet Spring Boot.

5 Conclusion

En conclusion, ce stage a montré qu'un apprentissage autonome et basique de Java est suffisant pour développer une application web simple avec Spring Boot. Toutefois, une approche enrichie, combinant les tutoriels Oracle, les tutoriels plus récents et des projets pratiques supplémentaires, serait plus bénéfique pour approfondir les compétences en programmation.

Nous avons également observé que Spring Boot est un framework accessible aux débutants grâce à son système de gestion des dépendances, permettant de créer une application fonctionnelle avec peu de connaissances préalables en développement web. Cependant, cette facilité d'utilisation peut également présenter un inconvénient : elle ne pousse pas toujours à une exploration approfondie des concepts sous-jacents, ce qui peut limiter le développement futur des compétences de l'apprenant, à moins que ce dernier ne fasse l'effort de se renseigner de manière autonome. De plus, bien que de nombreuses ressources soient disponibles pour approfondir ces connaissances, certaines peuvent manquer d'explications détaillées, nécessitant ainsi une recherche et une vérification supplémentaires pour garantir une compréhension complète des principes fondamentaux du développement web.

Enfin, la mise en pratique des apprentissages à travers le développement d'une application web a permis de renforcer et de tester les compétences acquises. Cependant, elle a également mis en évidence certaines limites, telles que la nécessité d'optimiser la gestion des entités de la base de données et le temps perdu sur des concepts moins pertinents, comme le pattern DTO dans ce cas particulier. Notons tout de même que le temps limité alloué à ce projet a restreint notre capacité à mettre en œuvre toutes les idées envisagées et à optimiser certains aspects du développement. Les perspectives d'amélioration

du projet sont alors les suivantes.

Pour renforcer la sécurité et la personnalisation des utilisateurs, il serait pertinent d'intégrer un système d'authentification des joueurs. Cela nécessiterait l'ajout d'un attribut pour le mot de passe dans l'objet `Joueur`, avec une contrainte d'unicité sur les noms des joueurs afin de garantir des comptes distincts.

Une autre amélioration importante serait d'ajouter la possibilité de rechercher l'historique des parties d'un joueur en utilisant son nom. Cette fonctionnalité permettrait de consulter non seulement les parties jouées, mais aussi des statistiques telles que le score moyen, offrant ainsi une vue d'ensemble des performances des joueurs.

De plus, il serait bénéfique d'enrichir les statistiques en incluant les scores des trois essais de chaque partie. Cela nécessiterait de conserver les entités `Essais` même après la fin des parties, ce qui entraînerait un volume de données plus important à stocker mais offrirait une analyse plus détaillée des performances des joueurs.

Enfin, pour enrichir les données disponibles, il serait utile d'inclure la date de création des parties. Cela permettrait non seulement de mieux suivre les sessions de jeu dans le temps, mais aussi d'analyser les tendances et les évolutions du jeu.

Bibliographie

Références

[1] Baeldung. *Spring ResponseStatusException*. https://www.baeldung.com/spring-response-status-exception. 2024.

[2] DevByteSchool. *Mastering Spring Boot - Implementing the Service Layer for Business Logic*. https://www.youtube.com/watch?v=HaYTMB4bj6O&list=PLkL-U3zEyYfmU-DtlYnkgV6CFTW7kvS9x. 2023.

[3] Devtiro. *The ULTIMATE Guide to Spring Boot : Spring Boot for Beginners*. https://www.youtube.com/watch?v=Nv2DERaMx-4&list=PLkL-U3zEyYfmU-DtlYnkgV6CFTW7kvS9x. 2023.

[4] Java Guides. *Spring MVC Tutorial with Spring Boot*. https://www.youtube.com/watch?v=Ku3gsv7_bCc&list=PLkL-U3zEyYfmU-DtlYnkgV6CFTW7kvS9x. 2021.

[5] h2. *H2 Database : Quickstart*. https://www.h2database.com/html/quickstart.html.

[6] Reiner Knizia. *Décathlon : Règles officielles*. https://www.knizia.de/wp-content/uploads/reiner/freebies/Website-Decathlon.pdf. 1990.

[7] Oracle. *Getting Started*. https://docs.oracle.com/javase/tutorial/getStarted/index.html.

[8] Oracle. *Interface Comparable<T>*. https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html.

[9] Oracle. *Interface Comparator<T>*. https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html.

[10] Oracle. *Interface Stream*. https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html.

[11] Oracle. *Lambda Expressions*. https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html.

[12] Prashant Sharma. *CRUD App in 30 mins*. https://www.youtube.com/watch?v=ZZTYQIUd_uY&list=PLkL-U3zEyYfmU-DtlYnkgV6CFTW7kvS9x. 2022.

[13] Teddy Smith. *Spring Boot Unit Testing With Mockito - 1. First Unit Test*. https://www.youtube.com/watch?v=jqwZthuBmZY&list=PL82C6-O4XrHcg8sNwpoDDhcxUCbFy855E. 2023.

[14] Teddy Smith. *Spring Boot Unit Testing With Mockito - Mocking Explained*. https://www.youtube.com/watch?v=4l3EFprMqpU&list=PL82C6-O4XrHcg8sNwpoDDhcxUCbFy855E&index=5. 2023.

[15] Spring. *Building an Application with Spring Boot*. https://spring.io/guides/gs/spring-boot.

[16] Telusko. *What is Dependency Injection*. https://www.youtube.com/watch?v=Eqi-hYX50MI&list=PLkL-U3zEyYfmU-DtlYnkgV6CFTW7kvS9x. 2018.

Annexe

1. Compte rendu destiné à l'encadrant : apprentissage de Java et Spring Boot + avis sur les tutoriaux d'Oracle
2. Correction détaillée des QCM GEI sur Java destinée à l'encadrant
3. Pour exécuter le projet sur IntelliJ IDEA : ouvrir dans IntelliJ, exécuter le fichier `long_jump_serveur/long_jump_serveur/src/main/java/fr/stageLIS/long_jump_serveur/LongJumpApplication.java`, attendre que le serveur se lance, entrer `http://localhost:8080` dans un navigateur, jouer !
4. Projet Final : application web Decathlon Long-Jump
5. Projets d'apprentissage de java (dés, cartes, bataille navale)
6. Projets d'apprentissage de Spring Boot (joueurs, livres, dés)