PROJET TUTORÉ

**Rapport Final**

ligne horizontale

# 

ligne horizontale

**Responsable:** POUZIN Pierre-Emmanuel Groupe n°1

**Responsable Qualité:** GIE Jimmy

**Responsable Données:** STIZ Romain

**Responsable Développeurs:** GUIBARD Théo

**Responsable DevOps:** ROUSSEAU Emile

**Développeurs:** GOURMAND Nicolas, TURIN Maxime, REMBERT Lucas, VOUETTE Maxence, POULET Alexandre

# 

Table des Matières

[**1) Rappel du contexte du projet et des objectifs 4**](#_uzg6dnxged34)

[1.1 Rappel fonctionnalités & choix techniques associés 4](#_q0xugimnpoc7)

[1.1.1 Front 4](#_po9lf5adgc3k)

[1.1.2 Back 4](#_29uyp3ndpr4b)

[1.1.3 ETL 5](#_bhc6ce4zrpic)

[1.2 Schéma de dépendances global 6](#_sczprnbrozp)

[**2) Documentation 6**](#_a70n7o5hkt5l)

[2.1 Plan de projet 6](#_qxoljlesi8ad)

[2.2 README 7](#_jx8zntzgnytk)

[2.3 DOXYGEN 7](#_qvl3ycjpwiid)

[**3) Fonctionnalités réalisées 7**](#_17569gtbkdkz)

[3.1 Front 7](#_2w5l2c7d51lq)

[3.1.1 Descriptif fonctionnel 7](#_hqvqihzovv)

[3.1.2 Descriptif technique 8](#_f5i7i1x9nfz4)

[3.1.2.1 Structure des pages 9](#_ed7g9wm3g214)

[3.1.2.2 Authentification 11](#_3ml7ggg7tmve)

[3.1.2.3 Parties 12](#_y80azrhrl1un)

[3.1.3 Tâche/Répartition 12](#_gtrh9yemzps1)

[3.1.4 Technologie 13](#_4hxinr8ws6v1)

[3.1.4.1 React 13](#_sj3dmg3zdd8x)

[3.1.4.2 TypeScript 13](#_6gaj3jafi1hf)

[3.1.4.3 Vite 14](#_b4jb8g895vvm)

[3.1.5 Justification 14](#_xrtjlbroz008)

[3.2 Back 15](#_eotoxlxh3kas)

[3.2.1 Introduction 15](#_s4d9zw4cw69l)

[3.2.2 Descriptif fonctionnel 15](#_r91rv8oeinxp)

[3.2.3 Descriptif technique 16](#_nti0vsslbq0g)

[3.2.3.1 Création de la base de données 16](#_d0dpdihafkim)

[3.2.3.2 Mapping des objets java avec Spring data JPA et Springboot 16](#_pw4fwi9yq1hm)

[3.2.3.3 Routes 17](#_2tt12apgk2ru)

[3.2.3.4 Algorithme Minimax 19](#_i0nhnu1p8kzl)

[3.2.3.5 Statistiques 21](#_emkutqbfg9nu)

[3.2.4 Tâche/Répartition 22](#_bjww3w5uio6h)

[3.2.5 Justification 23](#_i1zhjy4qkbvn)

[3.3 ETL 23](#_88lylyegerc0)

[3.3.1 Descriptif fonctionnel 23](#_v2f9k2o5tec6)

[3.3.2 Descriptif technique 24](#_kky93j72gkr)

[3.3.2.1 Extraction des données 24](#_9tyoz5wu09e)

[3.3.2.2 Insertion des données 25](#_f81o2fpkex35)

[3.3.2.3 Quelques chiffres 25](#_ne1a0s42m1fz)

[3.3.3 Répartition/Temps 26](#_mmhfjdn5g0n3)

[3.3.4 Justification 26](#_g6qda4wk4atv)

[3.4 Etat de la Base de Données 26](#_ei95mvf340qp)

[3.4.1 Descriptif fonctionnel 26](#_aer5ihyqikt0)

[3.4.2 Descriptif technique 27](#_sf5ulm9a9z3f)

[3.4.2.1 Player 27](#_yng1p0ia6xsx)

[3.4.2.2 Game 27](#_69b0grkosps9)

[3.4.2.3 Moves 28](#_aj1yj5ch6564)

[3.4.2.4 Tournament 28](#_ttwtm5w3eds8)

[3.4.2.5 Chess\_Opening 28](#_43dxi6ootycy)

[3.4.4 Répartition/Temps 28](#_l7ck6e5o7q0n)

[3.4.5 Justification 28](#_he2kzjqmoa0i)

[3.5 Utilisation de l’API de Lichess 28](#_6lbjf1ww6syu)

[3.5.1 Descriptif fonctionnel 28](#_99esik3wmjjh)

[3.5.2 Descriptif technique 29](#_7j5523qdnm5k)

[3.5.2.1 Appel de l’API 29](#_lx4ymmnbsebf)

[3.5.2.2 Extraction de données 29](#_2utjgfwwk4hn)

[3.5.2.3 Insertion de données 30](#_roartkyxsyry)

[3.5.3 Répartition/Temps 30](#_kphnchnlhgoh)

[3.5.4 Justification 30](#_esspwbl2vc8s)

[**4) Retour d’expérience sur l’environnement de développement et les choix techniques 30**](#_wks2dmhqmu6c)

[4.1 Git 30](#_7n7haffbpsly)

[4.2 Docker 31](#_fgzjlmpunge)

[4.3 Technologies Front 31](#_2sla2jexj4v5)

[4.3 Technologies ETL 32](#_qe609uuegqf)

[4.3 Technologies Back 32](#_lkw2veqe4l1b)

[**5) Retour d’expérience sur l’organisation du groupe 33**](#_xzp7d4a3s3rz)

[5.1 Planification et distribution des tâches 33](#_9hb41yfxa7b1)

[Ce qui a fonctionné 33](#_vpomkjkwzozp)

[Ce qui n’a pas fonctionné 33](#_mbeashhawpoa)

[Solutions à mettre en place pour améliorer la planification des tâches 33](#_9xcfvp18orn4)

[5.2 Communication 34](#_5ch3pqhebv2e)

[Ce qui a fonctionné 34](#_mri4a5fpsywo)

[Ce qui n’a pas fonctionné 34](#_skqrsp39ou6j)

[Axe d’amélioration 35](#_2j0006zelz1z)

[**6) Conclusion 35**](#_q2x18kyrldfl)

[**7) Sources 37**](#_hdgfulcvhh3z)

## 

## **Rappel du contexte du projet et des objectifs**

Le projet vise à développer une application autour des échecs, offrant une aide à la décision en analysant les parties en temps réel. En s'appuyant sur des données de parties issues de Lichess, l'application fournira des indications sur les coups joués et proposera des recommandations pour les coups futurs. Elle permettra également aux utilisateurs de disputer des parties contre des joueurs, assistés ou non par des moteurs comme Stockfish, et intégrera un système de détection de tricheurs basé sur l'analyse des coups.  
  
 Pour inclure les données de Lichess dans notre base de données, nous développerons un programme ETL. Ce programme filtrera les données entrantes pour conserver uniquement celles qui sont utiles à notre analyse, garantissant ainsi la pertinence et l'efficacité de notre système.

### **1.1 Rappel fonctionnalités & choix techniques associés**

#### **1.1.1 Front**

Cette application de jeu d'échecs vise à offrir une expérience fluide et interactive aux utilisateurs, leur permettant de jouer contre un bot ou un autre joueur, tout en visualisant des parties passées via l'importation de fichiers PGN. Elle utilise des technologies modernes telles que **React**, **TypeScript**, et **Vite** pour garantir une interface réactive, bien structurée et facile à maintenir. React gère l'interface utilisateur avec des composants réutilisables et des hooks pour l'état et les effets secondaires. **Chess.js** assure la logique du jeu (gestion des coups, validation des règles, détection de situations comme l'échec et mat), tandis que **React-Chessboard** fournit un affichage interactif de l’échiquier. L'IA **Stockfish** est intégrée via **Web Workers**, permettant d'exécuter les calculs sans perturber l'interface. L'authentification sécurisée s'appuie sur **OAuth 2.0 avec PKCE** pour l'intégration avec **Lichess**, et l'historique des coups peut être exporté en format PGN. Les tâches en cours incluent l'achèvement des routes backend et la récupération des statistiques des joueurs pour une meilleure analyse des parties.

#### **1.1.2 Back**

Le backend est conçu pour gérer et exploiter les données des parties d'échecs. Il joue un rôle central dans la logique métier, la manipulation des données, et les interactions avec le frontend ainsi que d'autres composants tels que l'ETL (Extract, Transform, Load) et la base de données. Parmi ses objectifs principaux figurent la fourniture de recommandations de coups, la génération de statistiques détaillées, et la détection d'éventuelles tricheries par des joueurs non humains.

**API REST :** Fournit des points d'accès RESTful permettant au frontend de communiquer efficacement avec le serveur.

**Traitement des données :** Gère l'intégration et le traitement des données provenant de l'ETL ou d'autres sources externes avant leur enregistrement dans la base de données.

**Connexion à la base de données :** Assure la persistance des données via une couche d'abstraction (JPA/Hibernate) pour simplifier leur manipulation.

**Intégration avec des systèmes externes :** Permet la communication avec des services ETL ou d'autres systèmes externes, notamment par des API ou des systèmes de messagerie.

Ce backend constitue un pilier fondamental pour l'application, garantissant une gestion fluide et efficace des données tout en facilitant leur utilisation par d'autres composants.

#### **1.1.3 ETL**

Notre programme ETL a pour rôle de migrer les données fournies par Lichess vers notre base de données. Ce dernier peut se décomposer en quatres modules symbolisant le traitement séquentiel des données.

Les données sont archivées au format **zst**; il est indiqué sur le [site](https://database.lichess.org/) de Lichess qu’après décompression, le volume de ces dernières est, en moyenne, multiplié par 7. Le temps nécessaire à la décompression des données ainsi que l’espace occupé par ces dernières est donc une problématique qui se doit d’être étudiée. A titre d’exemple un fichier de 4.37 Go demande 25 secondes de décompression et occupe 33 Go d’espace (serveur).

Les archives encapsulent des fichiers **.pgn** ([*Portable Game Notation*](https://fr.wikipedia.org/wiki/Portable_Game_Notation)) dont le contenu représente des parties d’échecs. Leur contenu respecte donc un standard, ce qui est idéal pour le parsing. Nous pouvons retrouver deux catégories d’informations dans ces fichiers. La première contient toutes les méta-données associées à la partie comme la date, le résultat, les joueurs, etc… . La seconde partie quant à elle décrit l’ensemble des coups joués lors de la partie. Ces derniers sont écrits selon la [*Standard Algebraic Notation*](https://fr.wikipedia.org/wiki/Portable_Game_Notation#SAN).

**1.1.4 API de LiChess**

L'API jouera un rôle essentiel dans notre système en permettant de récupérer de manière automatisée et structurée les informations pertinentes relatives aux joueurs, telles que leurs statistiques, leurs résultats, leurs parties passées, et bien d'autres données liées à leur activité sur la plateforme. Ces informations, provenant de LiChess, seront directement utilisées pour enrichir notre base de données, en fournissant des données actualisées et précises sur les joueurs.

**1.1.5 Base de Données**

La base de données jouera un rôle central en garantissant une gestion structurée et cohérente des données collectées depuis notre site web ainsi que des données issues de notre processus ETL (Extract, Transform, Load). Elle permettra non seulement de centraliser ces informations, mais aussi de faciliter leur traitement et leur analyse, tout en assurant leur intégrité et leur cohérence tout au long du cycle de vie des données.

Afin de garantir des performances optimales et une gestion efficace des données relationnelles, nous avons choisi *PostgreSQL* comme système de gestion de base de données (SGBD). Ce choix s’appuie sur plusieurs facteurs :

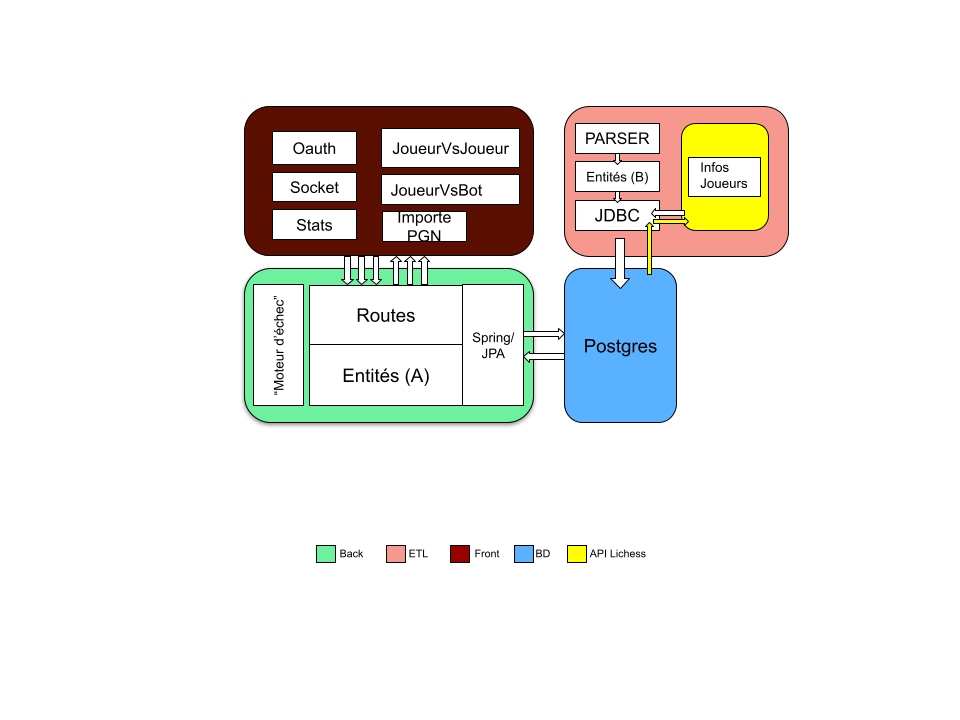
**Performance et Scalabilité** : PostgreSQL offre une excellente gestion des requêtes complexes et des transactions volumineuses, ce qui est essentiel pour des applications qui manipulent de grandes quantités de données issues d'un ETL.

**Support des données structurées et semi-structurées** : PostgreSQL permet de gérer efficacement tant les données structurées que semi-structurées, ce qui nous permet d’adapter la base aux évolutions futures de nos besoins.

**Fiabilité et sécurité** : Avec ses fonctionnalités avancées de gestion des accès, de sauvegarde et de reprise après sinistre, PostgreSQL assure une sécurité maximale des données stockées, un critère essentiel pour garantir leur disponibilité à long terme.

### **1.2 Schéma de dépendances global**

Voici le schéma de dépendances entre nos différentes parties permettant ainsi la liaison de celles-ci.



## **Documentation**

### **2.1 Plan de projet**

Le plan de projet a été minutieusement conçu et structuré à l’aide de l’outil OpenProject, qui facilite la gestion des tâches et le suivi des progrès. Il définit les objectifs à atteindre pour chaque étape du projet, en veillant à ce qu’ils soient clairs et mesurables. Les résultats attendus ont également été identifiés pour chaque phase, ce qui permet de vérifier l’avancement et d’ajuster les priorités en cas de besoin.

Chaque membre de l’équipe a un rôle bien défini, accompagné de responsabilités précises, afin de garantir une collaboration efficace et une répartition équitable des tâches. Des délais ont été établis pour chaque livrable afin de respecter les contraintes temporelles et d’assurer que le projet reste sur la bonne voie.

Un calendrier détaillé a été élaboré, incluant les grandes étapes, les jalons importants et les périodes dédiées à chaque activité. Enfin, toutes les tâches à exécuter ont été listées, priorisées et assignées de manière à optimiser leur réalisation et à assurer une progression fluide.

Ce plan de projet offre une vision claire et cohérente à l’ensemble de l’équipe, permettant un suivi précis et une coordination efficace pour atteindre les objectifs fixés.

### **2.2 README**

Il fournit les instructions nécessaires pour cloner, installer et lancer un projet ChessProject ETL/Back/Front qui utilise Docker pour exécuter un processus et interagir avec une base de données PostgreSQL. Il guide l'utilisateur à travers les étapes de clonage du projet, du lancement du projet avec Docker, ainsi que de l'accès à la base de données PostgreSQL à l'intérieur du conteneur. Des exemples de requêtes SQL sont également fournis pour interagir avec les tables principales du projet.Ce fichier est destiné à faciliter la mise en place et l'utilisation du projet pour les développeurs et les utilisateurs.

### **2.3 DOXYGEN**

Doxygen est un outil essentiel pour générer automatiquement de la documentation à partir de commentaires dans le code source, ce qui garantit une documentation à jour et cohérente avec l’implémentation du projet. Son utilité réside dans sa capacité à produire une documentation structurée, facilitant la compréhension du code et la navigation entre les différents modules. En encourageant les développeurs à ajouter des commentaires détaillés, Doxygen améliore la lisibilité du code, facilite la maintenance à long terme, et permet de partager facilement la documentation dans des formats variés, contribuant ainsi à des bonnes pratiques de développement et une meilleure collaboration au sein de l’équipe.

## **Fonctionnalités réalisées**

### **3.1 Front**

#### **3.1.1 Descriptif fonctionnel**

Le **front-end** de l'application a pour objectif principal d'offrir aux utilisateurs une expérience fluide et interactive pour jouer aux échecs. Il permet aux utilisateurs de se connecter, de jouer contre un autre joueur ou un bot, de visualiser des parties précédentes en important un fichier PGN le et de consulter des statistiques.

Composants techniques clés :

**React :** Utilisation de hooks React pour gérer les états et les cycles de vie des composants, avec des éléments comme useState pour la gestion d’état et useEffect pour les effets secondaires.

**Chess.js :** Utilisé pour la logique du jeu d’échecs (gestion des coups, validation, état du jeu).

**React-Chessboard** **:** Une bibliothèque React permettant l’affichage interactif de l’échiquier.

**Web Workers :** Pour l’exécution de Stockfish sans perturber l’interface utilisateur.

Structure des pages

**Page de connexion** **:** Offre à l’utilisateur la possibilité de se connecter via un compte Lichess ou de jouer en tant qu’invité.

**Page de profil :** Présente les statistiques du joueur et l’historique des parties jouées.

**Page de jeu :** Affiche l’échiquier et les informations de la partie en cours (joueurs, chronomètres, historique des coups).

**Mode Joueur vs Bot (*Stockfish*) :** Permet de jouer contre l’IA avec différents niveaux de difficulté.

**Mode Joueur vs Joueur local via WebSocket :** Permet de jouer contre un autre joueur en local.

Authentification et sécurité

L'application utilise **OAuth 2.0 avec PKCE** pour garantir la sécurité lors de la connexion avec **Lichess**. Ce processus assure que la communication entre l’application et Lichess soit sécurisée, en validant l’échange d’informations de manière cryptographique. Le mécanisme comprend plusieurs étapes : génération d’un **Code Verifier**, redirection vers la page d’autorisation de Lichess, et échange sécurisé contre un **jeton d’accès** permettant d'interagir avec l'API Lichess.

Les déplacements sont validés et gérés grâce à **chess.js**, qui assure la mise à jour de l’état de la partie en temps réel. Lorsque c’est au tour du bot, l’IA analyse la position et génère un coup, qui est appliqué et ajouté à l’historique. Un bouton permet à la fin de chaque partie de télécharger l’historique des coups sous format **PGN**.

#### **3.1.2 Descriptif technique**

L’interface utilisateur comprend un échiquier interactif, rendu grâce à la bibliothèque react-chessboard, où le joueur peut déplacer ses pièces. Un chronomètre est intégré pour chaque joueur, indiquant le temps restant pour les Blancs et les Noirs. La bibliothèque qui gère les déplacements et les règles des coups d'échecs est **chess.js**. Elle fournit une logique complète pour valider et effectuer les mouvements, gérer l'état du jeu, détecter les situations comme l'échec et mat, l’égalité.

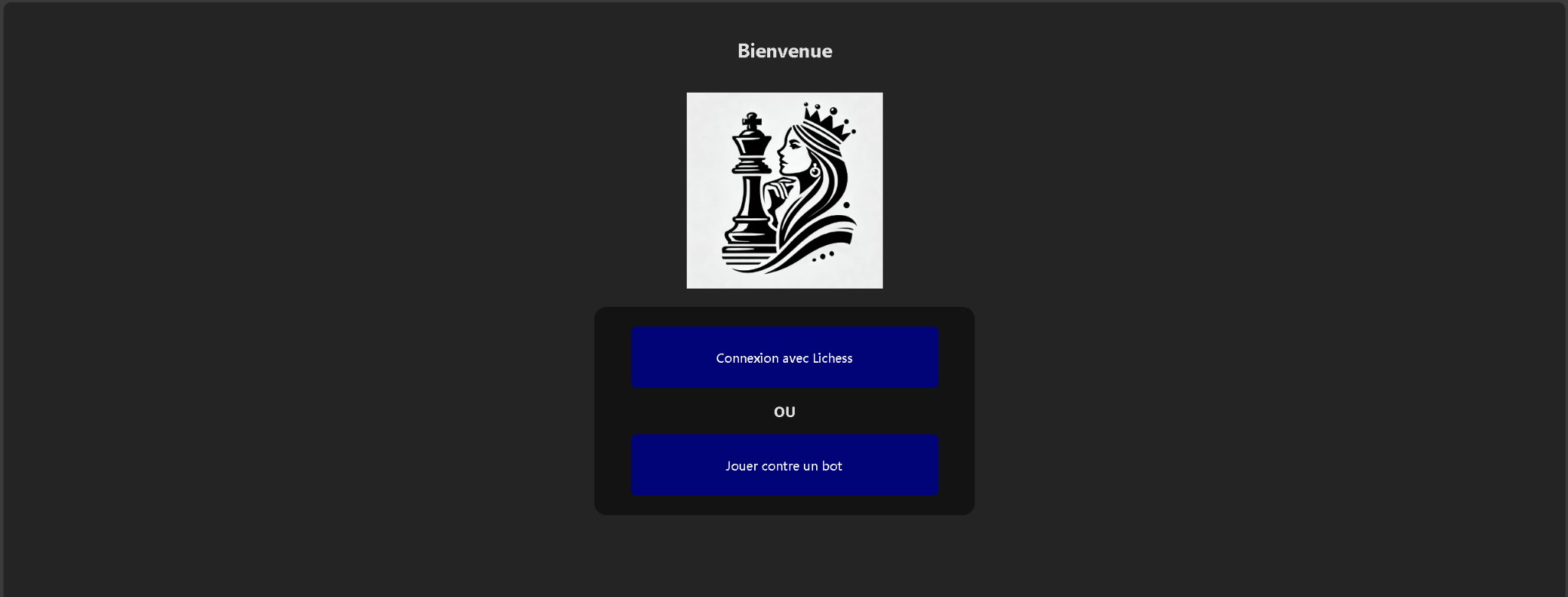
Pour rendre l’expérience plus immersive, l’intelligence artificielle *Stockfish* est exécutée dans un **web worker** afin d’éviter de bloquer l’interface utilisateur. Le niveau de jeu du bot est configurable via un paramètre prédéfini. Lorsque c’est au tour du bot, ce dernier analyse l’état de l’échiquier et génère automatiquement son meilleur coup, tout en ajoutant un délai aléatoire pour simuler un comportement réaliste.

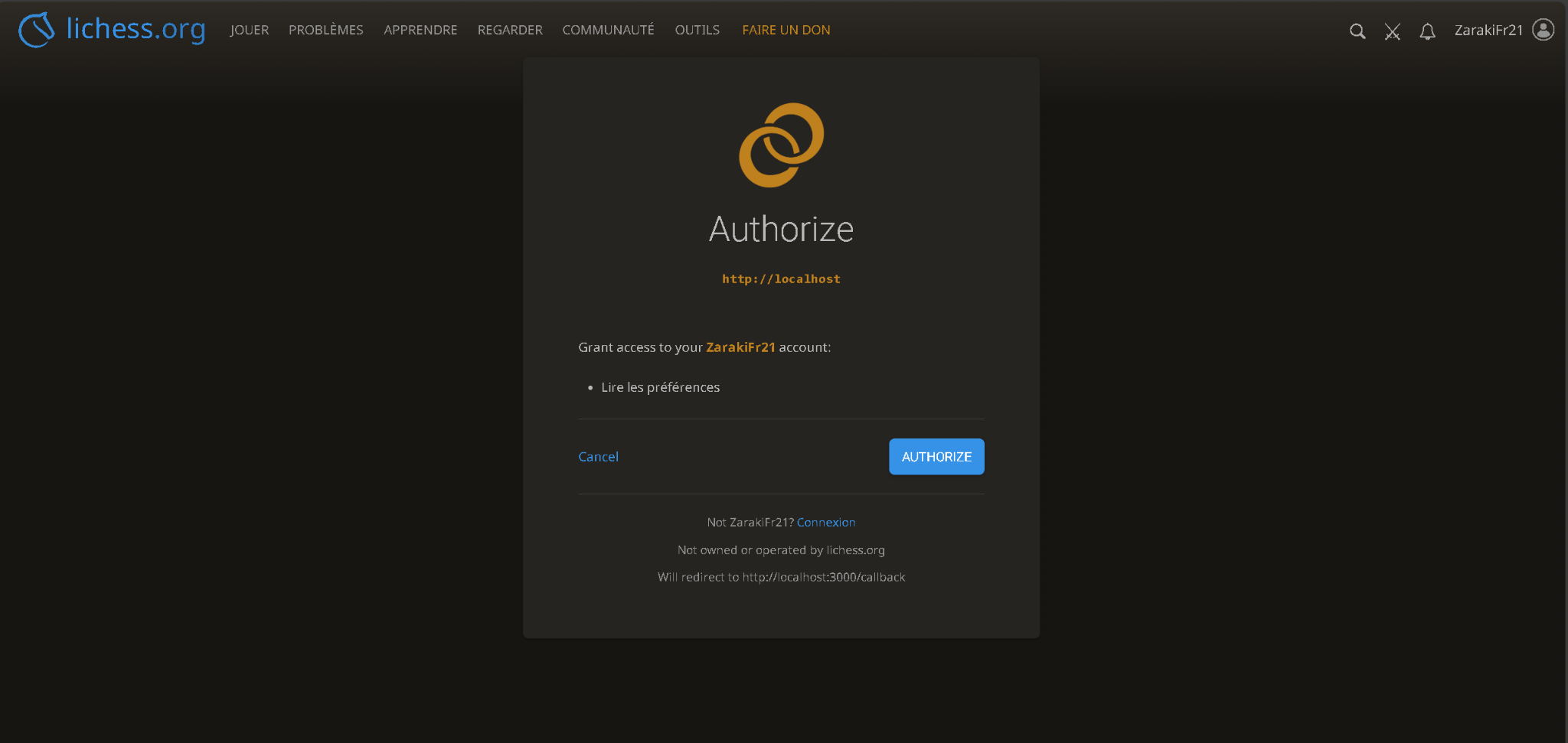
L’application gère également un historique des coups en notation algébrique, visible dans une section dédiée. À la fin de la partie, l’utilisateur peut télécharger un fichier au format **PGN** (Portable Game Notation) contenant les métadonnées et l’historique complet des mouvements. En termes techniques, l'application utilise des **hooks React**, comme useState pour la gestion des états et useEffect pour les cycles de vie. Les chronomètres des joueurs sont synchronisés avec le déroulement de la partie et basculent dynamiquement en fonction du joueur actif.

##### **3.1.2.1 Structure des pages**

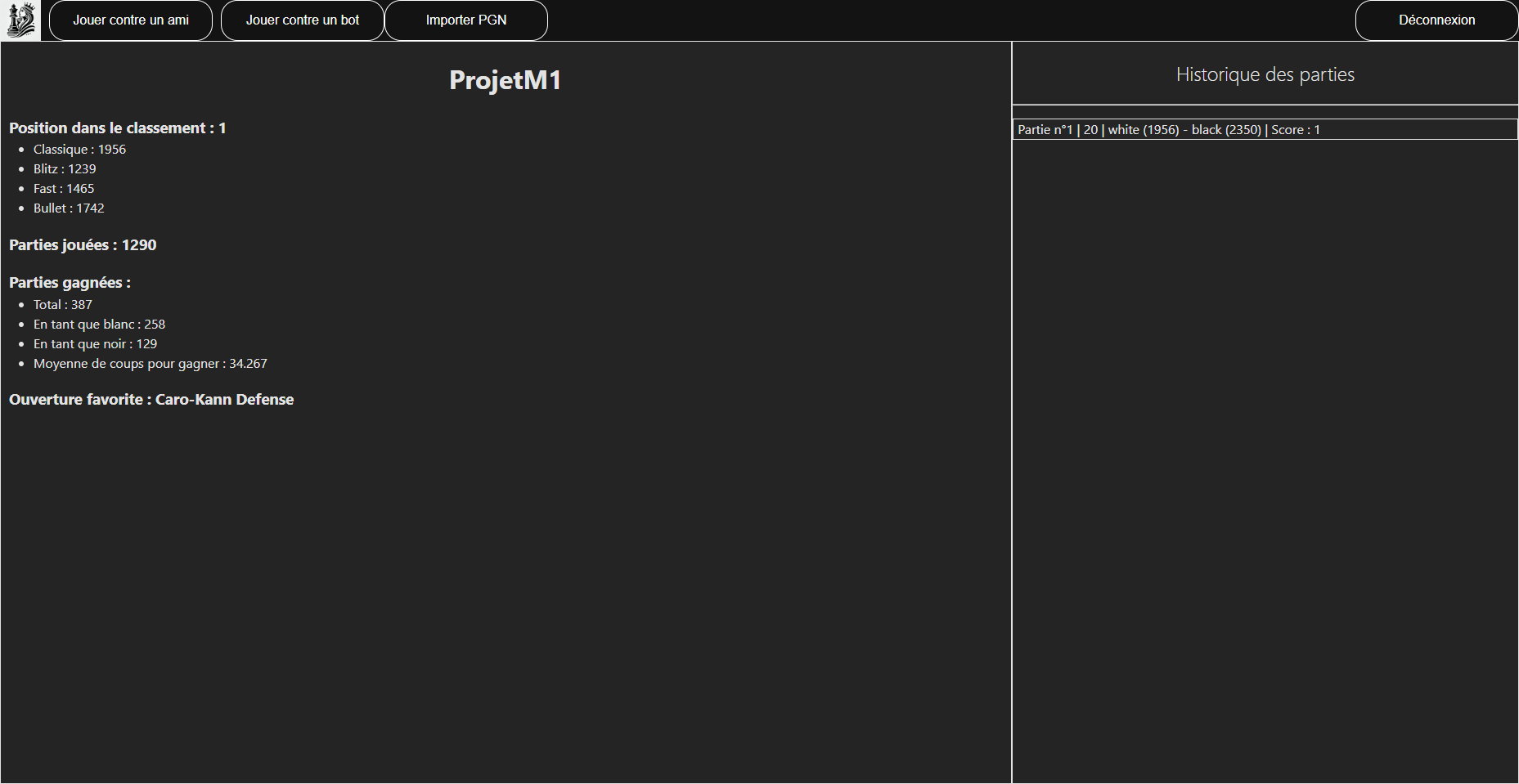
Notre application web est divisée en plusieurs pages, chacune ayant une fonction précise :

**Page de connexion(1) :** L’utilisateur à le choix soit de se connecter à notre application via un compte Lichess(2) soit de jouer une partie contre un bot en tant qu’invité. Si l’utilisateur décide de jouer en tant qu’invité, ses parties ne seront pas enregistrées.

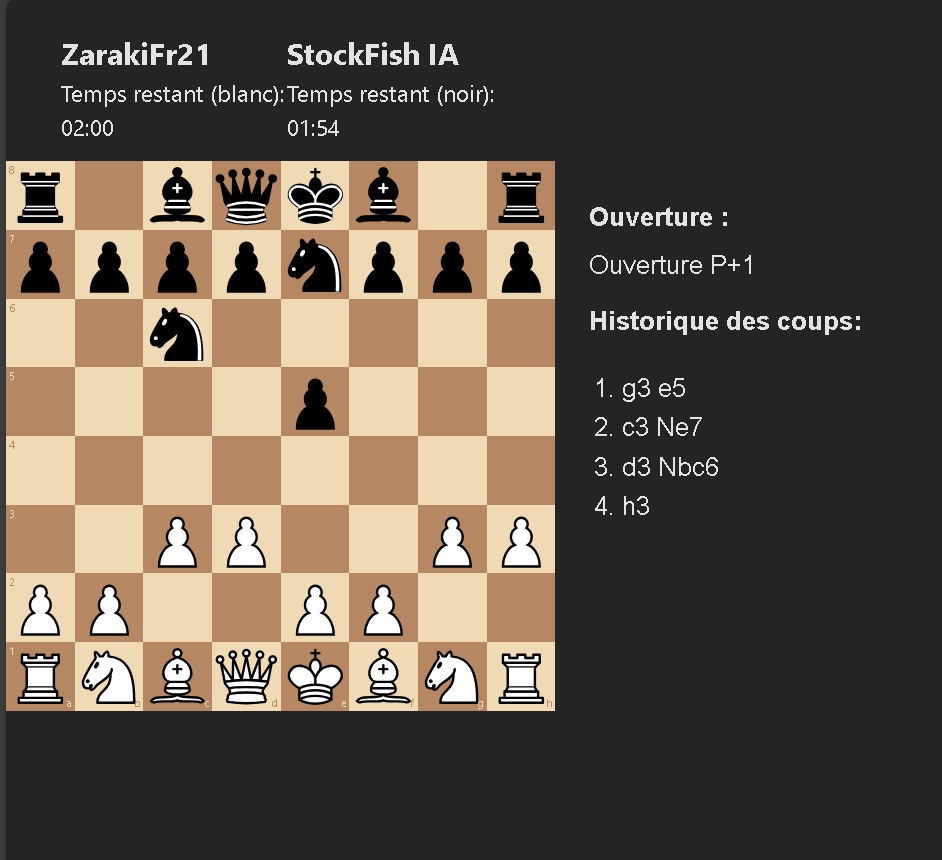
*(1)*

*(2)*

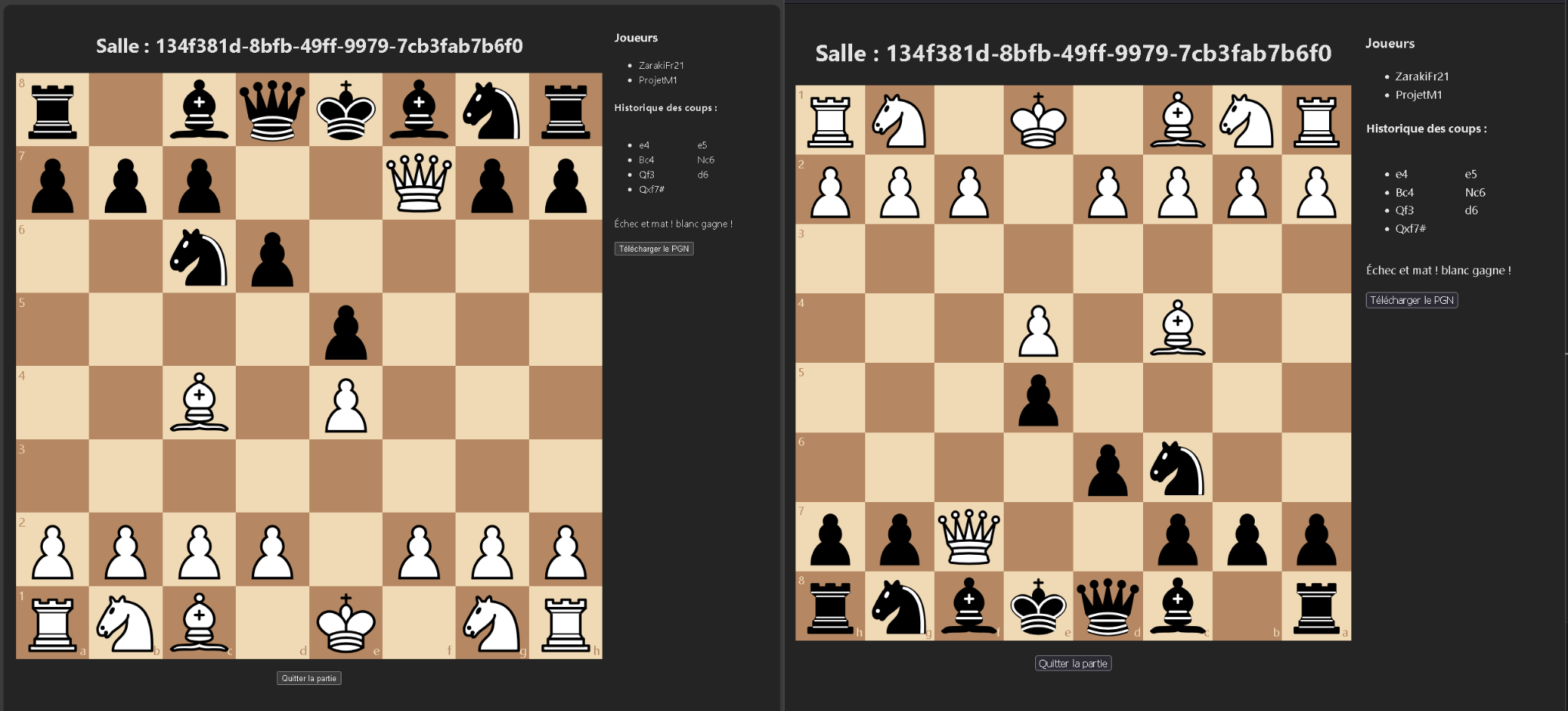
**Page de profil(3) :** Une fois connecté, l’utilisateur est redirigé sur la page de son profil. Cette page est découpée en 3 parties principales : le header, les statistiques et l’historique de parties. Dans le header, l’utilisateur peut trouver un bouton pour jouer contre un joueur et un bouton pour jouer contre un bot, qui à l’appui, font apparaître un pop-up permettant à l’utilisateur de spécifier les règles de la partie, un bouton pour lire les fichiers PGN, ainsi qu’un bouton pour afficher les statistiques générales des joueurs de notre base de donnée, enfin un bouton de déconnexion. Dans les statistiques qui sont les statistiques propre à ce joueur, situées sur la moitié gauche de la page, l’utilisateur peut retrouver son pseudo et des statistiques chiffrées dans chaque mode de jeu (Classique, Blitz et Bullet) tels que son elo actuel, son meilleur elo ou encore son nombre de parties jouées et bien d’autres. L’historique de parties, situé sur la moitié droite de la page, liste les dernières parties jouées par l’utilisateur en précisant le mode de jeu, les pseudos de l’utilisateur et de son adversaire, leur elo et le vainqueur.

*(3)*

**Page de jeu(4) :** Lors d’une partie, l’utilisateur peut voir son pseudo et le pseudo de la personne en face de lui ainsi que ceux de son adversaire et les clocks au-dessus de l’échiquier. Le nom de l’ouverture jouée et l’historique de coups sont visibles sur la droite de l’échiquier. À la fin de la partie, un message apparaît avec le nom du gagnant et la manière dont la partie s’est terminée.



*(4) Joueur vs Bot (IA stockfish)*



*(4bis) Joueur vs joueur en local via websocket*

##### **3.1.2.2 Authentification**

L'authentification est basé sur **OAuth 2.0 avec PKCE (Proof Key for Code Exchange)**, qui garantit une sécurité en utilisant la cryptographie pour valider l'échange d'informations entre le client et Lichess.

Le processus commence lorsque l'utilisateur accède à l'application, où une page d'accueil simple propose un lien de connexion. En cliquant sur ce lien de connexion à Lichess, l'utilisateur est redirigé vers un endpoint interne (/login) où une chaîne aléatoire appelée **Code Verifier** est générée côté serveur. Ce **Code Verifier** est ensuite transformé en un **Code Challenge** via une fonction de hachage SHA-256 et encodé en Base64. Ces valeurs sont utilisées pour sécuriser la communication entre le client et Lichess.

Le serveur conserve le **Code Verifier** dans la session utilisateur, tandis que le **Code Challenge**, accompagné d'autres informations nécessaires comme l'identifiant client (client\_id), l'URL de redirection (redirect\_uri) et la portée d'accès (scope), est transmis à Lichess dans une requête de redirection. Cette requête redirige l'utilisateur vers la page d'autorisation de Lichess, où il peut approuver ou refuser l'accès demandé par l'application.

Une fois l'autorisation accordée, Lichess redirige l'utilisateur vers l'URL spécifiée (/callback) avec un **code d'autorisation** inclus dans les paramètres de la requête. À ce stade, l'application effectue un échange du code d'autorisation contre un **jeton d'accès** en envoyant une requête POST sécurisée au serveur Lichess, en incluant le **Code Verifier** précédemment généré. Ce mécanisme garantit que seul le client qui a initié la requête d'autorisation peut finaliser l'échange, empêchant ainsi des attaques de type interception.

Avec le jeton d'accès obtenu, l'application peut interagir avec l'API Lichess pour récupérer les informations de l'utilisateur authentifié, comme son nom d'utilisateur. L'utilisateur est alors redirigé vers une interface front (/profil) avec son jeton et son nom d'utilisateur inclus dans les paramètres de l'URL, permettant une personnalisation immédiate de l'expérience utilisateur.

##### **3.1.2.3 Parties**

Les coups d'échecs sont récupérés et gérés grâce à la bibliothèque chess.js. Lorsque le joueur effectue un déplacement, la méthode game.move vérifie si le coup est valide, puis met à jour l'état du jeu (position en FEN et historique des coups). Les coups sont ensuite enregistrés dans un tableau appelé movesHistory, où chaque paire de coups (Blancs et Noirs) est regroupée pour un affichage clair. Si c'est au tour du bot, l'IA Stockfish calcule son meilleur coup en fonction de la position actuelle en FEN. Ce coup est ensuite appliqué au jeu et ajouté à l'historique.

Lorsque la partie est terminée, un bouton permet de télécharger l'historique complet des coups au format PGN. Ce fichier inclut non seulement la liste des coups mais également des métadonnées comme le nom des joueurs, la date, le résultat, et l'ouverture utilisée. Le téléchargement se fait en générant un fichier texte dynamique que l'utilisateur peut enregistrer localement. Enfin, pendant toute la partie, l'historique des coups est affiché sous forme de liste pour un suivi en temps réel.

#### **3.1.3 Tâche/Répartition**

Les estimations sont données en jours (1 jour = 3-4 heures de travail effectif) :

Dépendances et liens fonctionnels

1. **Affichage du plateau de jeu d’échecs avec React-Chessboard (**Temps estimé : 1 jour): Implémentation du plateau de jeu interactif via react-chessboard, avec synchronisation de l'état interne à l'aide de chess.js.
2. **Gestion des déplacements des pièces (**Temps estimé : 7 jours) : Validation des coups à l'aide de chess.js, mise à jour de la position du plateau via notation FEN, et gestion des déplacements avec un serveur (WebSocket) pour le joueur contre joueur.
3. **Gestion des interactions avec le serveur (Socket.IO) (**Temps estimé : 3 jours) : Mise en place de la communication en temps réel pour gérer les déplacements.
4. **Gestion des états du jeu (React) (**Temps estimé : 3 jours) :Centralisation de l'état du jeu, incluant les informations sur la partie, l'historique des coups, et la gestion des déconnexions.
5. **Mise en place des fonctionnalités de nettoyage et de redirection (navigation) (**Temps estimé : 1 jour) : Implémentation de la gestion de la déconnexion du joueur et la navigation vers la page précédente.
6. **Téléchargement de l’historique des coups sous forme de PGN (**Temps estimé : 1 jour) : Génération d'un fichier PGN du jeu, incluant les informations nécessaires (comme l'événement, les joueurs, etc.) et permettant le téléchargement via un bouton.
7. **Gestion de l'interface de lecture des PGN (**Temps estimé : 2 jours) :Implémentation d'un lecteur de parties PGN permettant de charger un fichier PGN, visualiser l'historique des coups et naviguer entre les coups dans une interface interactive.

À savoir que les tâches numérotées ont été réalisées dans l'ordre des tâches 1 à 7, car elles dépendent toutes de la précédente.

**Estimation du temps global:** Total global : ~18 jours

#### **3.1.4 Technologie**

Pour le développement du front-end de notre projet, nous avons opté pour les technologies **React**, **TypeScript**, et **Vite**. Ces choix sont motivés par plusieurs arguments techniques et pragmatiques.

##### **3.1.4.1** **React**

React est une bibliothèque JavaScript largement adoptée pour le développement d'interfaces utilisateur. Elle présente plusieurs avantages :

**Composants réutilisables** : React permet de créer des composants modulaires et réutilisables, facilitant ainsi le développement. Cela contribue à un code plus propre et plus facile à gérer, c’est un argument de taille dans une application développée à 20 mains.

**Virtual DOM** : React utilise un **DOM virtuel** qui améliore les performances en mettant à jour uniquement les éléments de l'interface qui ont changé. Cela optimise le rendu. Même si notre application front n’est pas d’un degré de complexité excessive et ne comporte pas de nombreuses interactions, nous pouvons définir que si notre front venait à accueillir plus de fonctionnalités, le Virtual DOM de React présenterait un atout majeur.

**Écosystème mature** : L'écosystème de React est vaste, offrant une multitude de bibliothèques et d'outils qui facilitent l'ajout de fonctionnalités, comme le routage (React Router), la gestion d'état (Redux ou React Context), ou encore l'intégration avec des API. La qualité de ces outils facilite grandement la phase de développement.

**Grande communauté et support** : Étant l'une des bibliothèques les plus populaires, React bénéficie d'une grande communauté, d'une excellente documentation, et de nombreux tutoriels. Cela facilite l'apprentissage et la résolution des problèmes. Étant donné que nous avons une expérience limitée dans le développement avec React, cet accès à des ressources de qualité est un point rassurant quant à la bonne conduite des développements front de notre projet.

##### **3.1.4.2** **TypeScript**

TypeScript est un sur-ensemble de JavaScript qui introduit des types statiques. Notre choix de TypeScript repose sur plusieurs raisons :

**Typage statique** : TypeScript permet de détecter les erreurs lors de la phase de compilation, avant même que le code ne soit exécuté. Cela réduit considérablement les bugs liés aux types et améliore la fiabilité du code. Ainsi du temps est gagné sur la phase de débogage.

**Meilleure lisibilité et maintenabilité** : Le typage explicite des variables et des fonctions rend le code plus lisible, surtout sur des projets collaboratifs. Il facilite aussi la navigation dans le code, en particulier lorsqu'on utilise des éditeurs comme VSCode ou Jetbrains qui intègrent des fonctionnalités d'autocomplétion.

**Interopérabilité avec JavaScript** : TypeScript est totalement interopérable avec JavaScript. Ainsi, si nous avions à utiliser des exemples de code issus d’internet, nous n’aurions pas à les adapter à notre choix de langage. De plus, il compile en JavaScript, garantissant ainsi une compatibilité totale avec les navigateurs.

**Écosystème en croissance** : La plupart des bibliothèques modernes, dont React, offrent un support natif pour TypeScript. Cela permet de bénéficier des avantages de TypeScript tout en utilisant un écosystème mature, fiable, aux performances assurées.

##### **3.1.4.3** **Vite**

Pour la gestion de notre environnement de développement et de build, nous avons choisi **Vite**, un bundle moderne et performant. Les avantages de Vite sont nombreux :

**Démarrage ultra-rapide** : Vite utilise une approche qui sépare le build de développement et de production. En développement, il tire parti de l'**ESM (ECMAScript Module) natif** des navigateurs, ce qui permet un démarrage instantané, même pour des projets plus lourds que pourrait être le nôtre. Lors du développement, il n'y a pas de processus de bundling initial lourd comme dans Webpack. Chaque fichier JavaScript est chargé individuellement sous forme de module dans le navigateur, ce qui accélère le démarrage de l'application.

**Rechargement à chaud performant** : Contrairement aux anciens outils comme Webpack, Vite met à jour uniquement les modules nécessaires lors des modifications du code, offrant ainsi une expérience de développement fluide.

**Build optimisé pour la production** : Pour la production, Vite utilise **Rollup** sous le capot, ce qui génère des bundles optimisés et minimise la taille du code final. Cela garantit des performances optimales pour les utilisateurs finaux.

**Support natif pour TypeScript** : Vite supporte nativement TypeScript, ce qui s'intègre parfaitement avec notre choix de cette technologie. De plus, sa configuration simplifiée permet de se concentrer sur le code plutôt que sur la gestion des outils de build.

**Facilité d'installation et de configuration** : Ce qui rend Vite particulièrement attractif pour les développeurs, et qui a été le principal argument motivant ce choix, c'est la simplicité de mise en place de l'environnement de développement. Il suffit de quelques commandes pour avoir un projet fonctionnel, et ce, **en moins d'une minute**. Cela accélère considérablement les premiers pas dans un projet, et nous permet de nous concentrer directement sur le code plutôt que sur la configuration complexe.

#### **3.1.5 Justification**

Le développement des fonctionnalités prévues progresse, mais plusieurs tâches importantes restent à réaliser avant leur finalisation.

Tout d'abord, l'envoi de l'historique des parties en format JSON est en attente de l'achèvement des routes backend nécessaires. Une fois ces routes mises en place, il sera possible de transmettre les données au frontend de manière efficace.

Ensuite, la récupération des informations sur les parties jouées et les statistiques associées depuis le backend est une priorité. Ces données devront être affichées sur l'interface utilisateur une fois que les routes correspondantes auront été développées.

Le développement de la fonctionnalité "Joueur contre joueur" est toujours en cours. Ce mode permettra d'ajouter une dimension compétitive entre deux utilisateurs, et son achèvement reste une tâche essentielle pour le projet.

Par ailleurs, l'identification et l'affichage du nom de l’ouverture jouée au cours de chaque partie doivent encore être implémentés. Cette fonctionnalité apportera une valeur ajoutée en permettant aux utilisateurs de mieux analyser leurs parties également à récupérer depuis notre BD, chessOpening.

Enfin, la métrique des coups, qui permet une analyse détaillée de chaque partie, reste à mettre en place. Elle contribuera à enrichir les statistiques et les outils d’évaluation des performances.

Ces étapes représentent des priorités importantes pour atteindre les objectifs fixés et offrir une expérience utilisateur complète et performante.  
  
Pour le moment et la fin du semestre, la priorité est d'assurer l'importation des données JSON depuis le backend vers le frontend. Ces données, comprenant les statistiques et les historiques des joueurs provenant de notre base de données, qui devront être affichées de manière claire et visuelle, notamment sous forme graphique.

### **3.2 Back**

#### **3.2.1 Introduction**

Le **backend** est conçu pour ***gérer l'accès aux données*** des parties d'échecs, effectuer les traitements nécessaires pour ***fournir des recommandations*** de coups, des ***statistiques***, et ***détecter d'éventuelles tricheries*** par des joueurs non humains et également ingérer les données générées par les utilisateurs. Il joue un rôle central dans la gestion de la logique métier, la manipulation des données, ainsi que l'interaction avec le frontend et d'autres composants tels que l'ETL et la base de données.

#### **3.2.2 Descriptif fonctionnel**

Dans un premier temps, la partie back de l'application a pour objectif de gérer l'accès aux données. Les utilisateurs y accèdent depuis la partie front par l'intermédiaire de routes qui permettent de récupérer certaines informations. À terme, un système d'authentification permettra de filtrer les utilisateurs afin d'implémenter des autorisations — par exemple, un utilisateur n'aura pas le droit d'utiliser la route qui mène au compte d'un autre utilisateur.

La fonctionnalité sur la détection de la triche durant les parties n'a pas encore été implémentée, mais elle dépend totalement du moteur de jeu qui réalise les recommandations de coups. Le moteur détectera la triche par exemple en comparant l'ELO du joueur avec les coups joués : si les coups joués se rapprochent plus d'un joueur d'ELO 2500 alors qu'il en a 1700, le moteur le détectera.

La consultation des statistiques des utilisateurs dépend fondamentalement de l'accès aux données, car les informations sont transmises par l'intermédiaire des routes entre la partie back et front. La fonctionnalité dépend aussi de la capacité de l'application à réutiliser les données générées par les joueurs durant les parties.

L'application doit également ingérer et réutiliser les données générées par les joueurs. Cette fonctionnalité dépend de la bonne exécution des parties sur le front de l'application, et de sa capacité à effectuer les insertions de manière efficace.

#### **3.2.3 Descriptif technique**

##### **3.2.3.1 Création de la base de données**

Nous avons d'abord conçu un script SQL regroupant la création de toutes les tables nécessaires à notre base de données. Ces tables, ainsi que leurs attributs, ont été définies en fonction des informations pouvant être extraites des fichiers PGN, de l’API Lichess, ainsi que des statistiques que nous souhaitons collecter et analyser. Les tables sont identiques à celles utilisées par l’ETL et sont gérées par Flyway. Flyway est un outil de gestion des migrations de base de données qui permet de versionner, de suivre et d'appliquer des modifications au schéma de la base de manière cohérente et automatisée. Il assure que chaque modification est enregistrée sous forme de scripts SQL numérotés, garantissant ainsi la synchronisation entre les environnements de développement, de test et de production.

Bien que l’ETL et le backend disposent de leurs propres bases de données, elles partagent le même volume grâce à Docker, ce qui garantit que leur contenu reste identique.

##### **3.2.3.2 Mapping des objets java avec Spring data JPA et Springboot**

Pour réaliser les différentes opérations sur la base de données, nous avons décidé d'utiliser les fonctionnalités avantageuses que proposent Spring Data JPA et Hibernate, qui permettent non seulement de faciliter les échanges avec la base de données en fournissant une surcouche SQL qui formate les requêtes, mais aussi de mapper nos objets java (nos entités) sur les tuples présents dans la base, et ce, de manière automatique.

**Dépendances**

Le projet général étant généré avec **Springboot** et compilé avec **Maven**, il nous suffisait d'ajouter dans le fichier **pom.xml** les dépendances suivantes pour commencer le mappage :

**Spring Web :** Spring Web permet de démarrer un serveur web intégré, sur lequel sera exposée notre application. Dans notre cas, **Tomcat** est utilisé. Il permet aussi de créer les API REST, nécessaires à l'élaboration des routes entre la partie « front » et « back » de l'application.

**Spring Data JPA :** Spring Data JPA, qui intègre le module Hibernate, permet de faciliter les opérations et d'annoter les classes d'objets pour les mapper aux tables de la base de données.

**Lombok :** Lombok est une dépendance d'utilitaires, facultatif, qui ajoute des fonctionnalités qui facilitent la vie du développeur. Parmi elles, la possibilité d'annoter les classes avec « @AllArgsConstructor », « @Builder » ou encore « @Getter » pour respectivement générer automatiquement un constructeur standard, créer une instance d'une classe avec des paramètres personnalisés, ou générer tous les getters de la classe.

À celles-ci s'ajoutent les dépendances qui étaient déjà nécessaires pour faire fonctionner le projet, à savoir **JDBC API** qui permet de se connecter à la base de données avec java, et **PostgreSQL Driver**, nécessaire également.

Le frontend communique avec le backend via des endpoints REST exposés par Spring Boot. Ces endpoints permettent d’accéder aux statistiques des joueurs et des parties, ainsi qu’aux recommandations de coups basées sur les calculs effectués par le backend.

Le programme ETL (Extract, Transform, Load) joue un rôle crucial en extrayant les données brutes des parties jouées sur LiChess, en les transformant dans un format exploitable, et en les chargeant dans la base de données. Ces données sont ensuite consommées par le backend pour fournir des statistiques précises et utiles aux utilisateurs.

**Architecture en couche (Controller, Entity, Repository)**

Dans l'objectif de structurer notre application, notre choix (motivé par les ressources de formation disponibles sur internet, comme les tutoriels et la documentation de Springboot) s'est porté sur une architecture en quatre couches distinctes.

La couche **Controller** est celle qui s'occupe de l'interaction entre la partie « front » et « back » de l'application. Spring Web fournit un panel d'annotations qui permettent d'associer des méthodes à des urls (@RequestMapping("/chemin"), @Getmapping, @Postmapping…) pour construire les routes de l'application, de répondre aux différentes requêtes HTTP du client en provenance du front (GET, POST…), et aussi d'exposer des « endpoints » (des points d'accès) qui permettront aux clients de récupérer efficacement des fichiers CSV et XML à l'aide des différentes API RESTful.

La couche **Entity** est celle liée aux tuples de la base de données. Une classe « Entity » correspond à une table de la base de données, et chacun de ses attributs correspond à une colonne de la table. Des annotations sont utilisées pour spécifier quel attribut est la clé primaire, ou si la valeur de la clé primaire doit être générée automatiquement, par exemple. C'est également ici que l'on spécifie le mappage avec « @Table(name = nom\_de\_la\_table) ».

Enfin, l’interface **Repository** est une interface. Des interfaces utiles mises à disposition par Spring Data JPA dépendent de celle-ci, comme CrudRepository (dont l'acronyme signifie « Create Read Update Delete »), JpaRepository, ou encore *ListCrudRepository*. Ces interfaces se distinguent par la variété de méthodes qu'elles proposent et qui peuvent se prêter à différentes situations, afin de faciliter les opérations sur la base de données.

###### **3.2.3.3 Routes**

L’établissement des routes a été réalisé en considérant deux points. Le premier est de fournir un accès bi-directionnel à l’ensemble de nos données. Le second est de proposer un niveau de granularité variable et adapté aux besoins du front pour limiter ses traitements sur les données reçues.

L’implémentation des routes s’est appuyé sur la [documentation](https://spring.io/guides/tutorials/rest) de spring. Cette dernière détaille la mise en place des routes mais également les fonctionnalités supplémentaires qui font de l’API une API REST. Trois packages distincts des entités sont utilisés.

Le premier d’entre eux nommé *assembler* permet d’encapsuler les entités (Game, Player,...) dans un type global: *EntityModel*. Ce type provient d’un module de spring nommé **Hateoas**. Ce dernier permet d’ajouter des routes relatives à la ressource demandée; fournissant des informations supplémentaires sur les capacités de l’API développée dans notre backend. A titre d’exemple, une requête GET sur un partie par id (une unique partie retournée) contiendra également un champ référençant la route donnant accès à l’ensemble des parties. De ce fait, pour chaque entité présente dans notre backend, il existe une classe associée du package *assembler* contenant une unique méthode **toModel(instance)** transformant l’entité originale en une instance d’*EntityModel* comportant ces routes additionnelles. Ce sont les instances d’*EntityModel* que le frontend recevra.

Le second package *repository*, utilisé de manière générale pour manipuler nos entités (liées aux tuples de la BD -> JPA…) est aussi utilisé pour nos routes. Tout comme pour le package *assembler*, le package *repository* contient une interface correspondant à chaque entité. Dans ces dernières, il nous est possible de décrire des méthodes permettant d’effectuer des requêtes sur notre base de données et d’initialiser les entités correspondantes (ou sauvegarder). Il faut noter que ces interfaces héritent d’interfaces pré-définies par Spring et comportent donc implicitement l’ensemble des méthodes CRUD. Ces méthodes ne comportent aucun corps. Spring déduit la requête que doit effectuer la méthode à partir de son nom ([mots-clés et exemples](https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html)) et des ses éventuels paramètres. Cependant, il est possible que ce dernier n’y parvienne pas. Dans ce cas, il est possible d’écrire la requête en SQL ou en JPQL au sein des annotations respectives **@NativeQuery** et **@Query**. Dans notre cas, seule la syntaxe JPQL est utilisée quand nécessaire, cette dernière étant plus simple de compréhension et ne limitant pas nos possibilités pour le moment.

La mise en commun des ressources décrites précédemment s’effectue au sein du package *controller*, plus précisément dans le corps de méthodes de routes GET (mise en forme des données envoyées uniquement). De la même façon que pour les deux packages précédents, le package *controller* comporte une classe par entité. Chacune de ces classes comprend deux attributs étants l’*assembler* et le *repository* correspondants. Lors du questionnement d’une route, le repository initialise une instance de l’entité correspondante puis cette dernière est transformer une une instance d’*EntityModel* à l’aide de la méthode **toModel(instance)**. Cette dernière est retournée par la fonction puis Spring se charge de la transmission. Pour se faire, Spring se référence à des mappings associant les méthodes à des actions HTTP. Ainsi, l’ensemble des méthodes au sein des classes du package *controller* sont préfixées par des annotations du type **@GetMapping**, **@PutMapping**, **@PostMapping** ou encore **@DeleteMapping**. La transmission de plusieurs entités à la fois est similaire. L’unique différence est l’emploi d’un nouveau “type” fournit pas **Hateoas** nommé *CollectionModel*. Brièvement, cette classe est une liste de *EntityModel*.

Les routes implémentées ont exclusivement été testées manuellement sur des faibles volumes de données. Spring propose l’automatisation de cette tâche avec JUnit. L’implémentation de ce framework devra être mise en place dans le futur (apprentissage et documentation préalable). L’intégration dans Spring de ce dernier devrait néanmoins faciliter cette tâche. Une évolution potentielle, notamment sur les classes de *repository*, concerne la façon de gérer la quantité de données retournées. Pour le moment, la quantité des données retournées par les méthodes de nos repository n’est pas prise en compte. Il pourrait s’avérer judicieux d’effectuer des tests et d’implémenter, dans le cas défavorable, les méthodes dites de “scrolling” permettant de recevoir des volumes de données importants par batch. Ces dernières sont détaillées dans la [documentation(sections contenant “scrolling”)](https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html) de Spring.

###### **3.2.3.4 Algorithme Minimax**

Pour donner des recommandations de coups à notre utilisateur, nous avons décidé de développer l’algorithme *Minimax*.

Pour ce faire, nous avons développé notre propre moteur d'échecs, *Board.java*, qui permet ainsi à notre algorithme de fonctionner indépendamment du projet.  
Ensuite, *Minimax* à l’aide du moteur d’échecs qui connaît tous les coups légaux et possibles (*roques, en passant,...*) va nous ressortir le coup ayant une bonne évaluation regardant en avance de 4 coups.

Il est nécessaire d’avoir la logique de *Minimax* afin de comprendre comment cela retourne le meilleur coup.

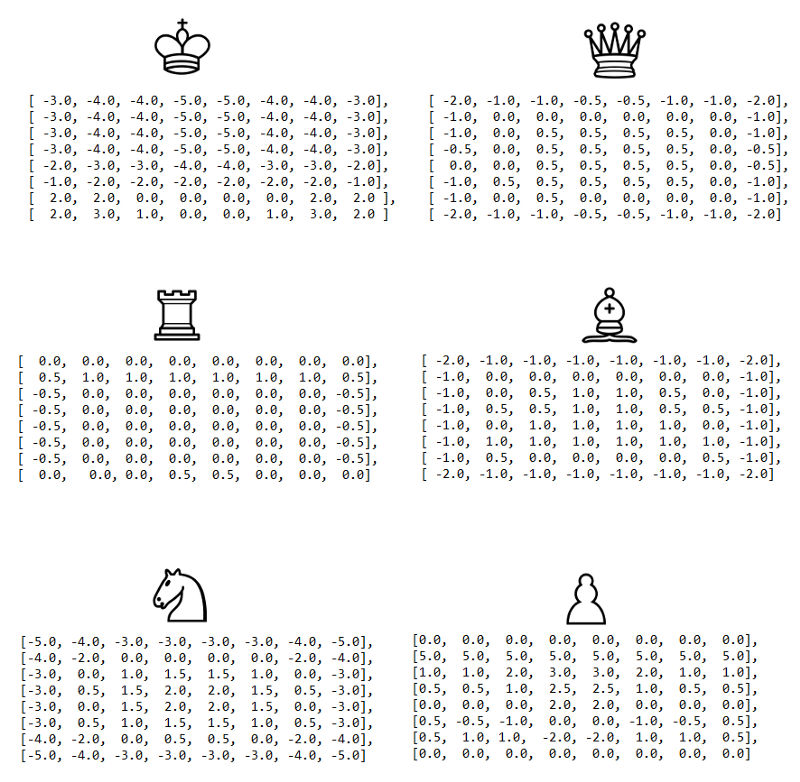
Pour cela *Minimax* va évaluer chaque coup selon une méthode définie. La méthode que nous utilisons pour évaluer l’échiquier est la suivante:

Nous calculons les pions noirs et blancs selon un score défini (roi blanc: 20000 points, dame: 900,.. et l’opposé pour les noirs) et nous effectuons la formule suivante:

***eval\_blanc = eval\_pion\_blanc - eval\_pion\_noir***

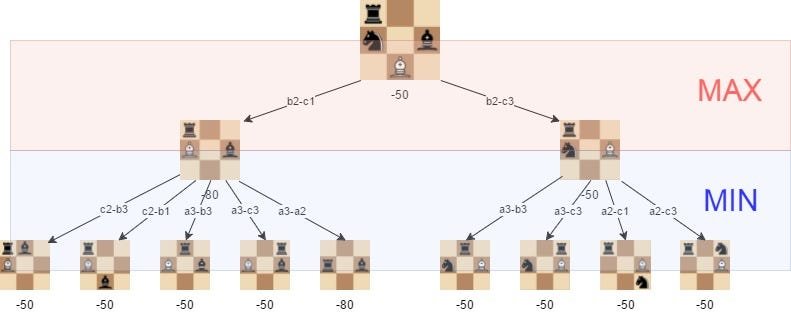
***eval\_noir = eval\_pion\_noir - eval\_pion\_blanc***

Ensuite, nous évaluons la position sur l’échiquier à l’aide de matrice de positionnement.

Généralement, nous valorisons le Roi qui est protégé par ses pions en début de partie, les pions qui essayent de se promouvoirs, les cavaliers qui ne sont pas sur les bords,...

Nous valorisons les pions qui sont au centre du plateau.  
  
 Enfin, nous regardons si la pièce, dans sa position, est en danger ou non.

Cela va permettre de donner une évaluation aux coups que l’on prédit afin de définir le meilleur.  
  
Afin de trier toutes les branches possibles, l’algorithme va chercher à maximiser pour le joueurs qui joue, ce que l’ennemi aura minimisé dans ce jeu à somme nulle.  
En d’autres termes, l’algorithme retourne le coup qui nous fera perdre le moins de points en termes d’évaluation.



Le problème, est le fait que certaines branches sont inutiles à calculer si nous trouvons déjà une sous-branche qui ne peut pas influencer le résultat final.

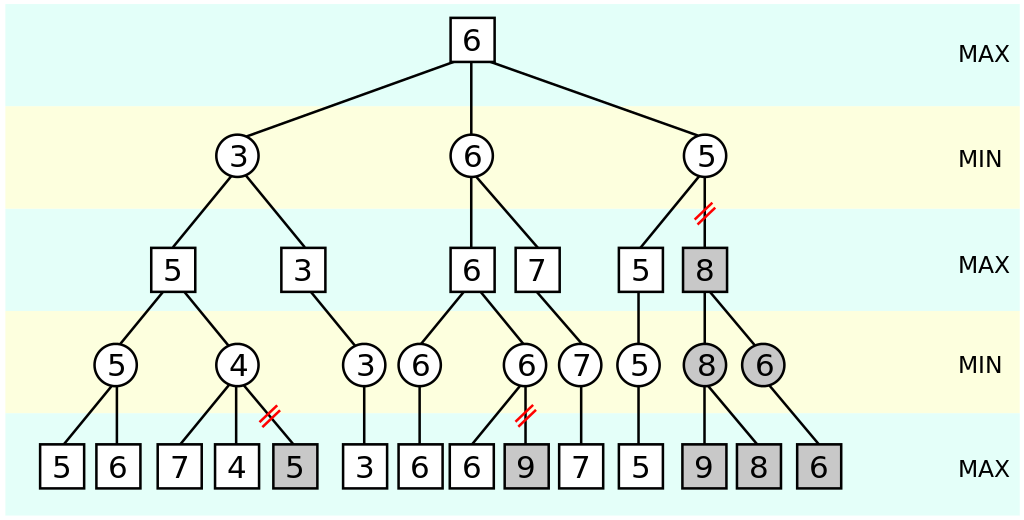
Nous allons donc ajouter *l’élagage Alpha-Bêta* à notre algorithme.

Cela est possible grâce à deux valeurs : **alpha** et **bêta**.

**Alpha** : La meilleure valeur trouvée jusqu'à présent pour le joueur à Maximiser.

**Bêta** : La meilleure valeur trouvée jusqu'à présent pour le joueur à Minimiser.

L'idée est que si à un moment donné une branche de l'arbre est déjà **pire** que ce qui a été trouvé dans une autre branche, il n'est pas nécessaire de l'explorer davantage. On peut "couper" cette branche (d'où le terme "élagage").



Cela va permettre de drastiquement réduire le nombre d’évaluations de coups.

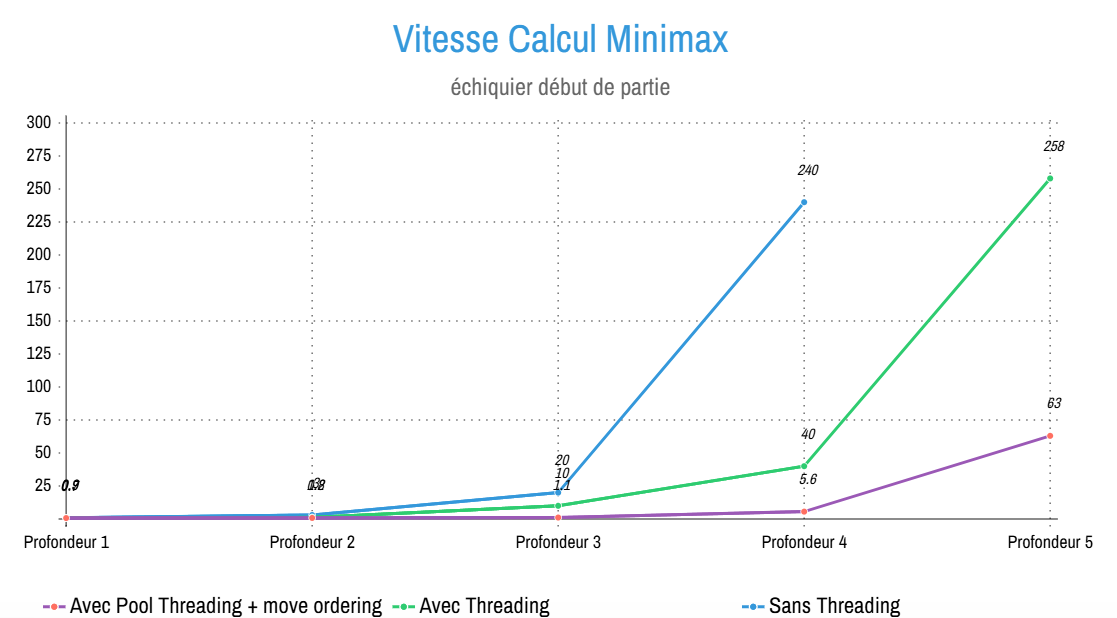
Cela reste tout de même trop long pour seulement 3 coups d’avance, à ce moment-là.  
Nous avons donc décidé d’utiliser du *Thread Pooling* qui augmente la vitesse de l’algorithme.

En effet, avant nous utilisions des *Threads* tout simple mais étant donné le nombre de coups à calculer, le fait de créer et de détruire les Threads n’était pas optimale, de ce fait, le Thread Pooling était une bonne solution qui permet la réutilisation des Threads.  
  
Ensuite, pour optimiser l’algorithme, nous avons mis en place un *Move Ordering*. C’est très simple à comprendre. Nous effectuons la recherche des coups dans un ordre précis. Actuellement, notre code regarde dans cette ordre:

***échec > prise > roque > en passant > autres coups***

En effet, si le roi est directement en échec, alors rien ne sert de calculer les autres branches, et ainsi de suite.  
  
Cela a permis de grandement réduire la complexité de cet algorithme afin de pouvoir voir jusqu’à 4 demi-coups en avance tout en ayant un temps raisonnable.

Ci-dessous, les tests effectués sur nos différentes versions de l'algorithme.



Temps de recherche Minimax en secondes/ Profondeur

On remarque que quand nous augmentons la profondeur, le temps en secondes explose. En effet, avec les *Pool Threading + move ordering* et une profondeur de 6, nous avons eu notre résultat au bout de 37 min et 10 secondes!

Il reste nonobstant, différentes manières d’optimiser cet algorithme, tel que de la mémoïsation de coups ou bien empêcher l’utilisation excessive de *new Board* à chaque coups, qui réduit la vitesse de calcul de notre programme.

Sans compter notre classe *Board* qui est sans doute mal optimisée car elle utilise un tableau 2D au lieu de [bitsboard](https://www.chessprogramming.org/Bitboards) ou d’un tableau 1D.

En outre, notre prédiction de coups, nous permet actuellement de battre des *Bots Chess.com*  de 1500 elo avec une précision entre 70-80%, ce qui signifie que nos coups sont précis à x% de ce que *StockFish* recommanderait.

###### **3.2.3.5 Statistiques**

L'implémentation des statistiques est toujours en cours de réalisation. L'objectif est que les joueurs puissent consulter différentes informations comme leur taux de victoire et de défaite, leur classement ELO actuel et maximum dans chacun des modes de jeu.

Dans une conception telle que l'usage du CPU doit être la plus faible possible pour limiter les temps de calcul, il sera nécessaire d'ajouter un certain nombre d'attributs aux tables existantes pour stocker les informations liées aux statistiques sans avoir besoin de les calculer. Les modifications sur la base de données seront réalisées avec de nouveaux scripts de migration avec flyway.

Puis, une fois que les tables seront modifiées en accord avec les statistiques que l'on souhaite fournir à l'utilisateur, il ne restera plus qu'à créer les routes correspondantes pour livrer les informations à la partie front de l'application.

##### **3.2.4 Tâche/Répartition**

Lors des séances de projet tutoré, nous avons procédé à un découpage des tâches pour organiser le travail efficacement. Voici les étapes principales :

1. **Jeu de test** : élaboration d’un jeu de données d’insertion pour valider la structure de la base de données.
2. **Mapping des objets Java** : mise en place des entités Java correspondant aux tables de la base de données.
3. **Routage entre le front et le back** : développement des routes nécessaires pour établir la communication entre l’interface utilisateur (front) et l’API (back). Cette tâche a nécessité environ 15 heures de travail, la première moitié de ce temps a été utilisée afin de se familiariser avec les différentes documentations relatives à RESTful sur Spring. La seconde a consisté en l’implémentation de ces dernières. De nombreuses routes additionnelles pourraient être définies à l’avenir.
4. **Algorithmes de statistiques** : écriture des algorithmes nécessaires pour calculer les statistiques et développement fonctions associées pour transmettre ces données au front. Cette tâche, encore en cours, durant l'écriture de ce rapport, nécessite une dizaine de jours.
5. **Migration Flyway** : remplacement du fichier SQL initial par Flyway, un outil de migration de base de données permettant de gérer les modifications du schéma de manière structurée et répétitive.
6. **Doxygen** : Permet une documentation technique détaillée

Le travail sur le backend a pris du retard en raison de plusieurs facteurs. Tout d'abord, nous avons sous-estimé la complexité de l'apprentissage de Spring Boot, en particulier pour le mapping des objets, qui s'est avéré plus difficile que prévu afin d'obtenir une structure correcte. Ensuite, la mise en place de l'environnement Docker, combinée aux phases de configuration et de débogage, a également consommé une part importante de notre temps. De plus, la progression des tâches a été ralentie par le fait que la moitié des membres affectés au backend étaient en alternance, ce qui a limité leur disponibilité.

Malgré ces obstacles, nous avons tout de même pu accomplir certaines étapes clés. La création de la base de données a été finalisée, accompagnée de l'élaboration d'un jeu de test d’insertion. Le travail sur le mapping des objets Java a été finalisé, posant ainsi les bases pour la suite du développement.Une majorité de route a été aussi compléter.Flyway et doxygen on eux aussi été mis en place. Ces réalisations nous permettent d'avancer malgré les défis rencontrés.

##### **3.2.5 Justification**

Le développement du backend est encore en cours et progresse mais des tâches importantes restent à planifier et à réaliser. Bien que le mapping des objets Java soit achevé, nous nous sommes rendu compte de certaines erreurs qu’il faut absolument corriger pour la suite, il est primordial de finaliser cette étape avant de passer à d'autres phases critiques. La mise en place des routes restant à développer est essentielle pour permettre une interaction fluide entre le frontend et le backend. Cela comprend également la gestion des données JSON, indispensables pour l’affichage des informations et la communication avec l’API Lichess pour la récupération des données des joueurs.Il reste aussi a finir de documenter notre code pour doxygen.

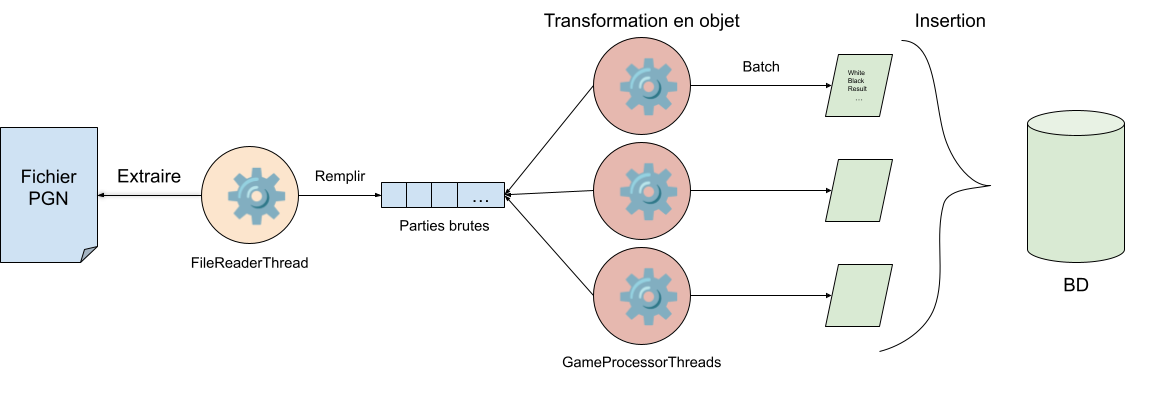
Afin d’améliorer encore la sécurité et la hiérarchie des routes, nous prévoyons de développer notre propre système d’authentification. Ce système personnalisé permettra de mieux contrôler l’accès aux différentes parties de l’application, en mettant en place des niveaux d’autorisations adaptés aux besoins, tout en renforçant la sécurité globale.

Nous chercherons à améliorer et optimiser notre algorithme Minimax afin de pouvoir essayer d’avoir une profondeur de 5 à 6 en quelques secondes seulement. Cela demandera une restructuration de la classe *Board* et de notre algorithme en prenant en compte les différents aspects expliqués dans la partie *Minimax*.

Enfin, une réévaluation du temps et des ressources alloués à chaque tâche est indispensable. Le planning initial n’étant plus aligné avec la réalité actuelle, il est nécessaire de réviser les priorités et d’établir un plan de travail plus réaliste. À cette fin, une redistribution des membres de l’équipe pourrait être envisagée, notamment maintenant que la partie ETL est pratiquement terminée.

### **3.3 ETL**

#### **3.3.1 Descriptif fonctionnel**



Notre programme ETL a pour rôle de peupler notre base de données à partir des fichiers de données disponibles sur Lichess. Ce dernier traite les parties une à une selon la procédure suivante. Premièrement, les données brutes sont extraites et chargées dans notre programme. Les informations contenues dans ces données sont ensuite lues et stockées dans des structures de données appropriées. Enfin, les données de notre programme sont insérées dans notre base de données. Ce processus se répète pour chaque partie contenue dans le fichier source.

#### **3.3.2 Descriptif technique**

##### **3.3.2.1 Extraction des données**

L’extraction de données brutes s'effectue à l’aide d'expressions régulières. D’abord, le fichier source est lu ligne par ligne à l’aide d’un BufferedReader. Chaque ligne est stockée temporairement dans une chaîne de caractères jusqu’à ce l’expression régulière représentant une partie complète soit trouvée. Dès lors, les champs tels que les coups et les différentes métadonnées de la partie sont extraits de la chaîne à l’aide d’expressions régulières spécifiques à chacun d’entre eux. Deux modules d’extraction distincts ont été implémentés, un pour les métadonnées et un pour les coups.

Les métadonnées sont strictement rédigées sous la forme suivante: [<nom\_du\_champ> “<valeur>”]. Certains champs ne sont pas toujours présents en fonction de la révision du format PGN. Une classe nommée **PGNLichess** référence les différents champs disponibles, les différents caractères de délimitation ou encore les valeurs récurrentes. Cette dernière décrit également l’ensemble des expressions régulières utilisées dans l’extraction des métadonnées. L’utilisation des expressions régulières s’effectue avec la combinaison des classes **Pattern** et **Matcher**. La première permet de transformer les chaînes de la classe **PGNLichess** en expressions régulières tandis que la seconde trouve les données s’y conformant. La classe **Matcher** comporte une méthode *group(int nbParenthèses)* permettant d’extraire une sous-chaîne dont l’expression régulière correspondante est comprise entre le nombre spécifié de parenthèses.

De la même manière que pour les métadonnées, le module d’extraction des coups implémente une classe (**SAN**) référençant les valeurs et symboles contenus dans ces derniers. La classe **SANParser** transforme les coups (contenus sur une ligne) en une liste (ArrayList) d’objets **Moves**. Les fichiers comportent deux types de coups distincts. Le premier type comporte trois informations: l’ordre, le coup du joueur blanc et le coup du joueur noir. Le second type se base sur le précédent à l’exception qu’il ajoute des informations supplémentaires associées à chaque coup. Ces informations aussi appelées commentaires fournissent des données supplémentaires telles que le temps auquel a été joué le coup (partie chronométrée) ou encore une évaluation sur la qualité du coup.

L’ensemble des données extraites de la chaîne de caractères brutes sont stockées dans une instance de la classe **ChessGame** dont tous les attributs reflètent les données lues. Cette classe contient également deux instances de la classe **Player** et une liste stockant les objets de type **Move**.L’extraction des données est effectuée au sein de la classe **PGNParser**. Cette dernière comporte deux **Queue** permettant de stocker les parties brutes (chaîne de caractères) et les parties traitées (instances de **ChessGame**). Ces structures ne sont pour le moment pas réellement exploitées puisque les parties sont traitées une à une. Elles permettront à l’avenir de paralléliser l’ingestion des données brutes et leur traitement. Le nombre de parties que le programme peut contenir est également limité. En effet, 100 000 parties nécessitent environ 1 Go de RAM d’où la nécessité d’effectuer un traitement sous forme de pipeline.

C’est pourquoi nous avons revu le modèle de notre parser pour l’optimiser. Nous avions pour idée de faire un cluster de machines et d’utiliser Java MPI pour traiter les parties, cependant les contraintes de temps nous ont ravisé et nous sommes partis vers une optimisation plus simple mais satisfiable : le multi-threading. En effet, la classe **PGNParser** a été modifiée pour utiliser des threads. En particulier, 1 thread pour la lecture du fichier PGN et le remplissage de la queue de parties brutes, et *n* autres threads pour s’occuper de transformer ces parties et les insérer dans la base de données (dans notre cas nous utilisons 5 threads dû aux contraintes de l’infrastructure comportant 6 coeurs).

##### **3.3.2.2 Insertion des données**

L’insertion des données était à la fois le **transtypage**, la **vérification** puis **l’insertion** dans notre base de données. En effet, nous nous étions mis d’accord préalablement pour que chaque méthode qui insère dans la base de données ne prenait que des chaînes de caractères en entrée afin d’avoir un code cohérent peu importe l’insertion. On ne cherche pas à savoir si on doit récupérer un int, une date,... Seulement des chaînes de caractères.  
 Ensuite, dans nos méthodes d'insertion, on va transtyper toutes nos chaînes de caractères en types voulus correspondant au schéma de la BD.

Durant cette phase, on va de même chercher à minimiser les erreurs possibles comme par exemple des “?” à la place d’entier pour l’elo de la partie.

Pour se faire, on va utiliser les opérateurs ternaires afin de rendre le code le plus compact possible. On regardait si la chaîne de caractères que l’on récupérait était vide, null, ou ne correspondait pas à ce que l’on cherchait, et si c’était le cas, on insère soit un 0 quand c’est un int, soit null, sinon on transtype la chaînes de caractères.

Une fois la vérification et le transtypage fait, on va alors pouvoir insérer dans la base de données tous nos attributs fraîchement correspondants.

Mais lors de l’insertion des données pendant les tests, on remarque plusieurs problèmes que nous avons omis par manque de rigueur. Plusieurs même joueurs peuvent être insérés, et de même pour les parties. Cela nous a poussé à faire de la vérification supplémentaire avant insertion mais aussi à créer un attribut pour les parties, **hash\_code**.   
A la différence des méthodes “bienveillantes” et “traditionnelles” pour créer un hashcode, nous avons tout simplement concaténé différents attributs qui, une fois ensembles, se rendaient unique.  
Nous avons donc concaténé *joueur\_blanc+joueur\_noir+date* (Y-M-D hh:mm:ss).

Cela ne pose pas de problème en termes de sécurité du fait que si un joueur ou une partie existe, alors il ne sera pas ajouté à notre base de données.

Cela à bien fonctionné pour les fichiers de petite taille mais lorsque nous avons commencé à traiter des millions de parties, certaines étaient corrompues (erreur dans les métadonnées, ligne manquante, etc.), nous avons alors décidé de ne pas les insérer du tout dans la base de données puisqu’elles sont négligeables environ 50 parties corrompues / millions de parties).  
Une fois tout cela mis en place, les tests fonctionnent parfaitement et la base de données se remplit entièrement.

Néanmoins, notre classe d’insertion sépare les requêtes par type d'insertion (joueurs, parties, moves, etc.), ce qui demande beaucoup d’accès à la base de données et ralentit tous les threads. Pour optimiser les requêtes nous les avons regroupées dans des batches afin de toutes les lancer en une seule connexion à la base de données, cela limite les accès à cette dernière et augmente l’efficacité des insertions.

##### **3.3.2.3 Quelques chiffres**

Plusieurs versions de l’ETL se sont succédées avec des hauts et des bas. En premier lieu, notre parser originel mono-thread qui était robuste mais très peu efficace. Puis nous avons ajouté des threads pour augmenter sa vitesse. Cependant, malgré cela, la vitesse d’insertion dans la base de données était bien trop lente. Il était alors nécessaire de regrouper ces requêtes sous forme de batches. Après ces différentes modifications nous avons obtenu de biens meilleurs résultats :

| Parties | ETL d’origine | ETL threads | ETL threads + batches |
| --- | --- | --- | --- |
| ~120 000 | En dizaines d’heures | ~3 heures | ~8 min |
| ~8 000 000 | n/a | n/a | ~8 heures |

Quelques chiffres supplémentaires :

* Il y a 6,202,058,053 parties sur la base de données Lichess actuellement (décembre 2024), il faudrait alors environ 258 jours pour notre ETL pour les insérer dans la BD.
* 90,847,982 de parties ont été jouées en novembre 2024, donc il faudrait un peu moins de 4 jours pour les traiter.
* En prenant en compte les limites matérielles, notre ETL pourrait en 1 mois traiter jusqu’à 720 millions de parties, au-delà de ce nombre, il y aurait trop de parties jouées par mois (en prenant en compte qu’il n’y a aucune panne).

#### **3.3.3 Répartition/Temps**

Nous avons séparé les tâches en 2 grands groupes, le parser avec l'insertion dans les classes puis l'insertion dans la BD en effectuant des tests au préalable.

Nous avons bien discuté entre nous afin de pouvoir mettre en commun nos codes le plus facilement possible.

Des deux côtés, les groupes avançaient bien malgré quelques problèmes qui nous ralentissaient (problème de parties parsées, de null dans les méthodes d'insertions,...). L’implémentation du parser de fichiers pgn a nécessité environ 30 heures (tests inclus). La globalité de ce module a été implémentée au cours des 20 premières heures. Le reste du temps a principalement permit l’ajustement de certaines expressions régulières et des tests associés. Le processus de test s’est avéré relativement lent de part le fait que certaines erreurs se produisaient après des heures de parsing (fichiers volumineux).

Une fois que les deux codes fonctionnaient séparément, il ne restait plus qu'à joindre les deux afin de terminer l'ETL. Du fait de nos discussions préalables, cela n'a posé aucun problème.

Cela s’est déroulé au long du début du semestre jusqu’à mi-Novembre.

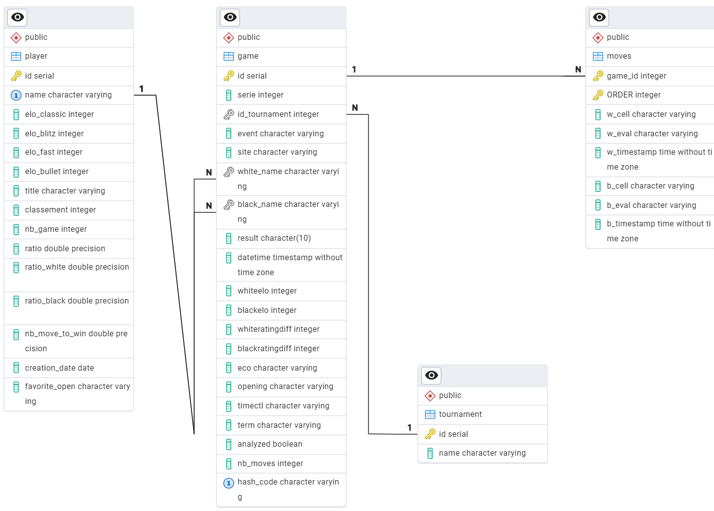
#### **3.3.4 Justification**

Notre programme ETL effectue sa fonction primaire, à savoir peupler notre base de données avec les données Lichess. Cependant ce dernier peut encore être amélioré notamment sur la parallélisation du processus. L’approche envisagée est de garder un programme quasiment similaire mais de diviser le travail initial en divisant le fichier source. Dès lors, le temps requis au traitement des données serait nettement diminué. Néanmoins, ces améliorations ne sont pas la priorité de notre groupe de part son utilité et sa fréquence d’utilisation.

### **3.4 Etat de la Base de Données**

#### **3.4.1 Descriptif fonctionnel**

La base de données joue un rôle central dans l’application, en assurant le stockage de toutes les données essentielles liées aux utilisateurs, aux parties et aux tournois. Elle conserve les informations clés des joueurs, telles que leurs statistiques personnelles, leur elo dans différents modes de jeu, ainsi que leur ratio de victoires en tant que joueur blanc ou noir. Chaque partie enregistrée est accompagnée des détails nécessaires pour sa reconstitution, incluant les coups joués et leurs évaluations. L’objectif est de permettre à l’application de fournir des analyses précises et de proposer des recommandations stratégiques aux joueurs. Par ailleurs, la base de données intègre également les tournois et les ouvertures d'échecs.



*Schéma de la Base de Données de notre Projet*

#### **3.4.2 Descriptif technique**

##### **3.4.2.1 Player**

La table **Player** est dédiée à la gestion des informations personnelles des joueurs, avec une attention portée sur leurs performances dans différents contextes. Chaque joueur est identifié de manière unique par un nom, garantissant l’absence de doublons lors des insertions. Cette table stocke l’elo des joueurs dans plusieurs modes de jeu, ce qui permet de suivre leur progression et d’offrir des statistiques détaillées, comme le ratio de victoires en tant que joueur blanc et noir. Ces données permettront à l’application de générer des tableaux de classement et de fournir un suivi personnalisé des performances.

##### **3.4.2.2 Game**

La table **Game** contient les détails spécifiques de chaque partie jouée, y compris les noms des joueurs, l’elo en début de partie, l’issue de la partie (victoire, défaite, ou match nul) et l'identité du gagnant. Cette structure facilite l’analyse postérieure des parties pour en extraire des données utiles, telles que les coups décisifs. Elle est également indispensable pour peupler l’application avec des exemples concrets de parties jouées, permettant ainsi d’illustrer des stratégies ou de recommander les meilleurs coups aux utilisateurs.

##### **3.4.2.3 Moves**

La table **Moves** contient la liste des coups d’une partie associée par son id.

Elle possède tous les coups associés à son évaluation et son temps dans la partie.

Cela permettra de déceler les meilleurs coups dans plusieurs situations ressemblantes.

##### **3.4.2.4 Tournament**

Pour le moment, dans les fichiers *.pgn* de LiChess que nous récupérons, il n’y a pas de *Tournoi*, mais cela n’empêche pas de notre côté de rajouter cette fonctionnalité.

Cela permettra de pouvoir avoir plusieurs parties groupées par un **id** afin de pouvoir fonctionner.

##### **3.4.2.5 Chess\_Opening**

Cette table contient environ 2200 ouvertures permettant ainsi de pouvoir rechercher les ouvertures correspondantes aux coups joués d'une partie.  
Elle permettra plus tard de l’utiliser pour les recommandations de coup et le reconnaissances des ouvertures en cours de partie.

#### **3.4.4 Répartition/Temps**

La mise en place de ces tables a nécessité un travail de recherche approfondi et de nombreuses heures d’optimisation. Lors de la création des méthodes d’insertion et des requêtes SQL, une attention particulière a été accordée à l’efficacité des jointures et à la réduction des temps d’exécution. Comme vu précédemment dans l’ETL, l’insertion des coups semble être le plus gros problèmes actuellement du fait que nous insérons les coups 1 par 1. Cela ralentit drastiquement la vitesse d’insertion ce qui fait qu’un fichier *pgn* de 100 *go* met 8 heures environ. Cette optimisation est essentielle pour garantir que l’application fonctionne de manière fluide, même avec un volume important de données. Le travail effectué sur la structure des tables a permis de minimiser les temps de réponse et de maximiser la performance des requêtes complexes. Cela s’est déroulé sur tout le semestre, il est donc difficile de donner un temps précis.

#### **3.4.5 Justification**

La base de données est un élément vital pour le projet afin de nécessiter des informations des utilisateurs et des parties en général.

Celle-ci évoluera en fonction de nos besoins surtout pour les statistiques des joueurs.

Elle évoluera encore sûrement si nous voulons intégrer d'autres statistiques.

### **3.5 Utilisation de l’API de Lichess**

#### **3.5.1 Descriptif fonctionnel**

Le rôle de l’API est de remplir la table *Player* de notre base de données à l’aide d’un appel de l’API de LiChess. En effet, cet appel permet de récupérer des informations sur les joueurs de notre base de données, tels que le nombre de parties par mode de jeu, l’elo de chacun d’entre eux,...  
Chacunes de ces informations sont récupérées dans un *json* par lequel nous devons traiter la réponse reçue afin de récupérer les informations utiles pour nous.

Ultérieurement, nous chercherons à automatiser cet appel à l’aide de la bibliothèque java *ScheduledExecutorService* que nous appellerons tous les mois.

#### **3.5.2 Descriptif technique**

##### **3.5.2.1 Appel de l’API**

Nous devons comprendre quelles sont les limites de l’API avant de pouvoir récupérer toutes les données de chaque joueur. En effet, l’API semble plutôt stricte car elle ne permet que de faire des appels par groupe de 300 au maximum et sans pouvoir l’appeler toutes les secondes.

Pour se faire, afin de récupérer les statistiques des joueurs, nous récupérons à l’aide d’une requête SQL, tous nos joueurs présents sur notre BD ainsi que leurs nombres.

Cela va nous permettre de faire des appels successifs de l’API par paquet de 300 joueurs à chaque fois. Afin de ne pas dépasser la limite LiChess, on peut mettre un “sleep” afin que le prochain paquet de 300 joueurs ne se fasse pas bloquer par LiChess.

##### **3.5.2.2 Extraction de données**

Une fois la réponse de l'API obtenue, il fallait pouvoir parser la réponse en json (le tout concaténé en un bloc).

Pour cela, nous avons fait un *regex*, bien que des librairies existent, nous aimons les *regex*. Mais cela ne va récupérer que pour 1 joueur. Alors, nous avons créé une *Map<String, Map<String, String>>* qui contient le nom du joueur avec les statistiques (elo classique, elo rapide,...) puis la statistique associée. Une fois fait, nous pouvons parcourir notre json et récupérer chacune des statistiques et l'associer à chaque joueur.

Cela peut sembler archaïques mais nous avons vraiment le contrôle sur ce que l’on récupère et traite, en plus de nous apprendre comment le faire nous-même sans dépendre de librairies.

Malgré cela il n'y a qu'un seul problème que le regex a du mal à parser. En effet, il n'est pas impossible que certains joueurs désactivent leurs comptes et que la réponse de l'API dans le JSON est juste : {“id”:id\_joueur,”username”: nom\_joueur, “disabled”:true}

Notre regex le détecte mais continue de lire et parse jusqu'au prochain joueur.

Ce qui fait, que dans notre terminal, le joueur qui a désactivé son compte aura les statistiques du prochain joueur. Et ce joueur-là aura son compte sans statistiques.

La seule solution, hormis de rechercher un nouveau regex, et de supprimer dans notre fichier de partie, toutes les parties où le joueur qui n'a plus de compte apparaît.

Cela permet de gagner du temps “en trichant” mais aussi de ne pas falsifier les données qui seront insérées plus tard dans notre base de données.

Mis à part ce problème, il y a des statistiques qui sont impossibles à récupérer par groupe de joueurs. En effet, il n'est pas possible de récupérer le nombre de parties gagnées en tant que joueur blanc ou noir lors de l’appel de l’API pour plusieurs joueurs en même temps. Pour cela, nous avons créé une méthode qui récupère toutes les parties joués par un utilisateur et nous avons regardé s'il gagne en tant que joueur blanc ou noir. Une fois le parcours des parties, nous avons stockés dans un tableau d'entier de taille 6 les données suivantes: 0: Nombre de partie; 1: Nombre de parties gagnées en tant que blanc; 2: Nombre de parties gagnées en tant que noir; 3: Ratio Victoire; 4: Ratio Victoire Blanc; 5: Ratio Victoire Noir.

Cela permet de ne pas passer seulement par l’API mais aussi d’exploiter notre base de données afin de ressortir des statistiques différentes de l’API.

##### **3.5.2.3 Insertion de données**

Afin d'insérer les données, il ne suffit plus que de lire la Map et d'appeler la méthode de la classe Joueur permettant d'insérer les statistiques de la partie.

On vérifie bien entendu que chacune des chaînes de caractères que l'on récupère est cohérente en effectuant des vérifications avant l'insertion.

Cela permet de garnir notre table Joueur de différentes statistiques permettant plus tard de faire des graphes sur celles-ci.

#### **3.5.3 Répartition/Temps**

La partie concernant l’API s’est déroulée une fois l’ETL fini. C'est-à-dire qu’on a décidé de faire les scripts nécessaires qui récupèrent les données de chaque utilisateur.  
Au final, comme ce qui était prévu, cette partie était plutôt rapide malgré les quelques problèmes vus précédemment.

Une seule personne était affectée à la tâche pour une durée d’environ 5 jours afin de peaufiner le *regex* et mettre en place les méthodes d’insertions*.*

#### **3.5.4 Justification**

L'API nous permet de récupérer toutes les statistiques des joueurs présents sur notre base de données même si toutes ses parties ne sont pas enregistrées sur la nôtre.

Cependant l’API a des limites qui ne nous permettent pas de pouvoir faire tout le travail.

C'est pour cela que nous utilisons les résultats de l'API afin de fournir d'autres statistiques possibles.

Pour le moment cela reste rapide pour 300 joueurs mais quid de plusieurs milliers de joueurs. Il sera sans doute nécessaire de paralléliser cela, en lançant un thread pour chaque groupe de 300 joueurs afin de maximiser la vitesse d'exécution.

Le problème étant que l'API ne supporte que 8000 joueurs toutes les 10 minutes et 120 000 joueurs par jour, il faudrait limiter de notre côté le nombre de requêtes que l’on envoie.

## **Retour d’expérience sur l’environnement de développement et les choix techniques**

### **4.1 Git**

L'implémentation de Git au sein du groupe s'est faite de manière assez naturelle. Un avantageux transfert de connaissance a pu se réaliser, où ceux qui maîtrisaient l'outil ont initié ceux qui ne le maîtrisaient pas aux bonnes pratiques et manières de faire, afin que chacun puisse mettre en commun ses travaux efficacement. De plus, certaines branches ont déjà dû subir un « merge », et peu, voire aucun conflit majeur n'a eu lieu lors des fusions.

Cependant, certains aspects peuvent encore être améliorés, comme une utilisation plus poussée des fonctionnalités proposées par Git. Le système d'*issue*, par exemple, permet d'obtenir un suivi de l'avancement d'une tâche en particulier en y associant une branche spécifique avec la possibilité d'écrire des notes d'activité, des commentaires ou de téléverser des fichiers, qui seront vus par les développeurs affectés à l'*issue* en question. Il en va de même pour le système des *Merge Request*, qui permet non seulement d'affecter des « rewiewer » qui auront pour tâche de valider le code du développeur, mais qui en plus permet de suivre en direct l'évolution (les commits) de la branche de feature par rapport à la branche originelle en affichant les conflits potentiels qui pourraient survenir en cas de fusion. Bien que cela semble un peu accessoire, ce genre de fonctionnalités pourraient, une fois prises en main, être une réelle valeur ajoutée.

### 

### **4.2 Docker**

L’utilisation de Docker s’est avérée judicieuse pour de nombreuses raisons. Cette technologie nous a permis de distribuer des environnements d’exécution aux membres de l’équipe sans que ces derniers n’aient besoin d’installer d’outils supplémentaires sur leurs machines. De plus, ces environnements sont utilisables à l’aide de deux commandes: *docker compose up* et *docker compose down* rendant leur utilisation très simple. L’isolation des différents services dans des containers à éliminer les éventuels conflits provoqués par une implémentation sur des machines dont les environnements diffèrent.

La mise en place fut également simple. Docker est une technologie facile d’accès et documentée. Cet outil nous offre une modularité sur les différents services composant notre application. De ce fait, il nous sera aisé de regrouper ces derniers (répartis sur ETL, back, front) pour composer notre application complète. La spécification des versions de chaque service nous permet de contrôler ces dernières limitant alors les problèmes de mises à jour indésirées.

### 

### **4.3 Technologies Front**

Ce qui a fonctionné :

Le composant React ChessGameWithBotApp permet de jouer contre le bot Stockfish. L'initialisation de Stockfish, la mise à jour de l'échiquier après chaque coup, la gestion du chronomètre pour les deux joueurs, ainsi que l'affichage des historiques de coups et de l'ouverture, fonctionnent globalement bien.

Ce qui n’a pas fonctionné :

La mise à jour de l’ouverture reste incomplète et ne reflète pas l'ouverture jouée par les blancs et les noirs de manière précise.

Quelques fonctionnalités (comme l'ajout dynamique de l'ECO dans le fichier PGN) ne sont pas encore implémentées.

Solutions à mettre en place pour améliorer :

Gestion des ouvertures : Intégrer une bibliothèque ou un fichier de données permettant de détecter et d'afficher les ouvertures en fonction des coups joués (par exemple, une base de données PGN pour correspondances), soit notre BD ChessOpening.

Ajout des métadonnées ECO : Automatiser l'identification du code ECO et son intégration dans les métadonnées PGN.

### **4.3 Technologies ETL**

Ce qui à fonctionné :

Nous avons opté pour des technologies que nous connaissons bien ce qui a rendu le développement de l’ETL très compréhensible et intuitif. Après plusieurs mises à jour, nous avons réussi à obtenir un ETL plutôt performant.

Ce qui n’as pas fonctionné :

Cependant, les expressions régulières sont une potentielle source d’erreur car les fichiers de la base de données Lichess sont étalés sur 10 ans, ce qui rend leur contenu différent d’une année à l’autre. Et d’un autre côté, le temps d’exécution est assez long, ce qui peut être pénible pour certains tests.

Solution à mettre en place pour améliorer :

Pour résoudre le problème du temps d’exécution serait d’optimiser le développement de la pipeline séquentielle puis de la “clusterisée” afin de répartir la charge de travail (que ce soit pour la lecture ou la transformation ou l’insertion).

### **4.3 Technologies Back**

L’utilisation de Spring Boot s’est avérée une excellente idée car il simplifie considérablement le développement d’applications en offrant des outils intégrés pour configurer rapidement un serveur web, gérer les dépendances avec Maven, et établir des APIs REST. Sa compatibilité avec Spring Data JPA et Hibernate a facilité le mapping des entités Java avec les tables de la base de données, tout en automatisant les opérations courantes. De plus, Spring Boot suit une architecture structurée (Controller, Service, Repository, Entity), ce qui rend le code clair, maintenable et extensible. Enfin, sa vaste documentation et communauté ont été d'une grande aide.

Mis à part la gestion du temps, notamment sous-estimé pour l’apprentissage des technologies comme Spring Boot et l’ API REST, tout s’est déroulé comme prévu.

Pour améliorer la gestion du projet, il serait important de réaliser une planification plus réaliste en tenant compte des temps nécessaires pour l’apprentissage des technologies, comme Spring Boot et les API REST. Il serait également utile de renforcer la formation initiale sur ces outils afin d’en accélérer la prise en main. Une meilleure répartition des tâches et une communication efficace entre les membres de l’équipe pourraient éviter les blocages. Enfin, utiliser l'outil de gestion de projet Openproject de façon plus efficace permettrait de suivre l’avancement en temps réel et d’ajuster les priorités si nécessaire.

## **Retour d’expérience sur l’organisation du groupe**

### **5.1 Planification et distribution des tâches**

#### **Ce qui a fonctionné**

**Établissement d’un planning global** : Nous avons défini dès le départ un planning global des grands jalons du projet, ce qui a permis d’avoir une vision d’ensemble des étapes principales à réaliser. Bien que cette planification n’ait pas été détaillée, elle a offert une structure minimale pour guider les efforts collectifs.

**Avancement global malgré une planification sommaire** : Les tâches globales identifiées ont été prises en charge par les membres du groupe en fonction de ce qui avait été défini dans notre premier rapport. Cette approche, bien que non optimale, a permis d’avancer sur le projet.

**Utilisation d’OpenProject** : Nous avons utilisé OpenProject, un outil complet pour la planification et le suivi des tâches. Cependant, notre utilisation a été limitée, et nous n’avons pas exploité toutes ses fonctionnalités.

**Priorisation des tâches critiques** : Nous avons su identifier et prioriser les tâches dont d’autres dépendent, notamment la conception et le développement de l’ETL. Cela a permis de structurer les efforts autour des livrables les plus importants.

**Adaptabilité** : Malgré le manque de structuration, nous avons su nous adapter et orienter les développements lorsque cela était nécessaire, montrant une capacité de réaction efficace face aux imprévus.

#### **Ce qui n’a pas fonctionné**

**Absence de planification détaillée** : Si un planning global a été établi, aucun détail précis n’a été ajouté concernant les tâches individuelles, leurs délais ou les ressources nécessaires. Cela a conduit à une répartition improvisée des tâches, avec des membres prenant des responsabilités "à la volée" selon leurs disponibilités.

**Manque de visibilité globale** : L’absence de suivi régulier et structuré a rendu difficile la compréhension de l’état d’avancement global du projet. Les membres avaient parfois une perception partielle ou incomplète de la progression collective.

**Utilisation limitée et inefficace d’OpenProject** : Bien que cet outil ait été choisi pour la gestion des tâches, il s’est révélé lent et quelque peu complexe à utiliser. En conséquence, il a été sous-exploité, ce qui a réduit son utilité pour le suivi et la coordination.

**Manque de synchronisation des efforts** : Bien que nous ayons avancé collectivement, l'absence de structure claire a occasionné une progression moins fluide, avec des moments où certains membres étaient en attente d'autres.

#### **Solutions à mettre en place pour améliorer la planification des tâches**

1. **Adopter une méthode simplifiée et adaptée**

Utiliser le **tableau Kanban** seul d’OpenProject, plus intuitif; pour suivre les tâches à faire, en cours, et terminées.

Décomposer les grandes tâches en sous-tâches détaillées, avec des descriptions claires et un suivi précis.

1. **Introduire des délais réalistes et des responsabilités claires**

Associer chaque tâche à une **deadline réaliste** et désigner un **responsable identifié**, afin d’améliorer la gestion des priorités et d’assurer un meilleur suivi.

1. **Favoriser une meilleure visibilité**

Mettre en place un tableau de bord collaboratif qui montre l’état global d’avancement du projet et les tâches en cours, afin que chaque membre puisse facilement suivre la progression collective.

En appliquant ces solutions, la planification des tâches pourra devenir plus claire et efficace, permettant une gestion plus fluide et mieux coordonnée pour les projets à venir.

### **5.2 Communication**

#### **Ce qui a fonctionné**

La communication au sein du groupe s’est appuyée principalement sur deux canaux : **Discord** pour les échanges en ligne et les **réunions physiques** pour des interactions directes. Malgré une utilisation irrégulière, Discord s’est révélé plus accessible et flexible que Mattermost, car il ne nécessite pas de connexion VPN, ce qui a simplifié et fluidifié la tenue des échanges.

Les réunions physiques ont constitué le socle de notre communication, permettant à chaque membre de participer activement. Ces moments d’échanges directs ont permis une coordination efficace et une implication collective.   
Par ailleurs, l’utilisation de Discord pour la communication en temps réel a aidé à résoudre certains problèmes concrets liés au développement.

#### **Ce qui n’a pas fonctionné**

Cependant, plusieurs limitations ont impacté la fluidité de nos échanges :

**Manque de communication hors réunions physiques** : Durant les périodes de vacances ou lorsque nous travaillions à distance, les interactions étaient limitées. Cela a entraîné des mises à jour sporadiques sur l’avancement du projet, rendant difficile un suivi continu.

**Organisation sur Discord** : Bien que pratique, l’utilisation d’un seul canal pour tous les échanges a rapidement conduit à une surcharge d’informations, rendant les discussions moins claires.

**Absence de règles claires de communication** : Le groupe n’a pas établi de directives spécifiques, comme la fréquence des mises à jour ou les délais de réponse attendus, ce qui a parfois nui à l’efficacité globale du projet.

#### **Axe d’amélioration**

**Structurer les canaux de communication sur Discord**

Créer des **canaux thématiques** pour organiser les échanges selon les différents aspects du projet, tels que :

Développement (questions techniques, bugs, solutions proposées)

Organisation (planning, échéances)

Documentation (partage de fichiers et notes)

Discussions générales.  
Cela permettra d'éviter les discussions brouillonnes sur un seul canal et de retrouver plus facilement des informations spécifiques.

**Encourager la communication asynchrone**

Introduire des pratiques favorisant les échanges en différé, comme l’utilisation de courriels ou de messages résumant les avancées et décisions importantes.

Prévoir un moment fixe chaque semaine pour que chaque membre partage une **mise à jour individuelle** sur l’état d’avancement de ses tâches (via Discord ou un autre outil collaboratif). Cela garantira une visibilité continue, même lorsque les membres travaillent à distance ou pendant les périodes moins actives.

**Instaurer des règles de communication claires**

Préciser les outils et situations pour les interactions (par exemple, Discord pour les discussions courantes, réunions physiques pour les prises de décision importantes).

Encourager une culture de **signalement** lorsqu’un membre rencontre une difficulté ou un blocage, afin que le groupe puisse intervenir rapidement.

## **Conclusion**

L'implémentation du projet a permis de tirer de nombreuses leçons, tant sur le plan technique que sur le plan organisationnel. Si certaines technologies, telles que Git, Docker, Spring Boot, et React, ont bien fonctionné et ont permis d'atteindre les objectifs principaux, plusieurs axes d'amélioration ont été identifiés, notamment en termes de gestion de projet, de communication au sein de l'équipe et d'optimisation des processus techniques.

**Bilan technique**

Concernant les technologies, bien que les outils comme Git aient été utilisés de manière satisfaisante dans une première approche, une meilleure exploitation de ses fonctionnalités aurait permis de rendre le suivi de l'avancement plus structuré et précis. Par exemple, l'utilisation plus poussée des systèmes d'issues et de Merge Requests aurait facilité la gestion des tâches et amélioré la visibilité du projet au quotidien. La gestion des dépendances et la mise en place de processus de validation du code auraient aussi renforcé la qualité du travail en équipe.

Dans le domaine de Docker, le projet a fonctionné, mais des améliorations sont nécessaires pour garantir une meilleure expérience utilisateur, notamment en ce qui concerne l’interface front-end. Quant aux technologies front-end, bien que le composant React *ChessGameWithBotApp* ait permis de répondre aux objectifs de base, des améliorations sont nécessaires pour gérer les ouvertures et l'intégration des métadonnées ECO dans les fichiers PGN. Il reste encore des tâches à accomplir pour automatiser et rendre cette partie plus fluide.

Les technologies de traitement des données ETL ont, pour leur part, permis un développement intuitif, mais le temps d'exécution reste un problème qui nécessite une optimisation, notamment par l'usage de solutions multi-threading. Quant au back-end, Spring Boot a été un choix judicieux, permettant un développement rapide grâce à son architecture claire et à ses outils bien intégrés. Cependant, une meilleure planification pour l’apprentissage des technologies, notamment Spring Boot et les APIs REST, aurait permis de gagner en efficacité et en temps.

**Bilan organisationnel**

Sur le plan organisationnel, une planification plus détaillée et réaliste aurait permis une gestion du projet plus fluide. Le fait de ne pas avoir décomposé les tâches de manière suffisamment précise a conduit à une certaine confusion et à une répartition imprévisible des responsabilités. Il est essentiel à l’avenir de mettre en place une gestion de projet plus rigoureuse, en définissant clairement les sous-tâches et en attribuant des délais réalistes. Utiliser pleinement les fonctionnalités de l'outil OpenProject aurait également permis une meilleure coordination et un suivi plus précis de l’avancement du projet.

En termes de communication, bien que Discord et les réunions physiques aient facilité les échanges, l'absence de règles claires sur la fréquence et la nature des communications a parfois freiné l'efficacité du groupe. Il aurait été bénéfique de structurer davantage les canaux de communication sur Discord en créant des espaces thématiques pour chaque aspect du projet. De plus, une meilleure organisation des échanges asynchrones et l'instauration de moments réguliers de mise à jour sur l’état d’avancement auraient permis de maintenir une visibilité continue sur les tâches, même pendant les périodes où les membres travaillaient à distance.

**Recommandations pour l'avenir**

Pour les projets futurs, il est essentiel d’adopter une approche plus structurée, tant au niveau de la gestion des tâches que de la communication. L’utilisation optimale d’outils comme Git, OpenProject et Discord nécessite de clarifier leur usage et de former les membres de l’équipe à leurs fonctionnalités avancées. En parallèle, une gestion plus rigoureuse du temps, avec des délais réalistes et un suivi régulier de l’avancement, permettrait de mieux gérer les imprévus et d'éviter les blocages.

Le travail technique doit également inclure une attention particulière aux optimisations nécessaires, notamment pour les aspects liés aux performances du système et à la gestion des données. L’automatisation de certaines tâches, comme l’intégration des métadonnées dans les fichiers PGN, ainsi que la gestion des ouvertures dans les jeux d'échecs, est essentielle pour améliorer la robustesse et l’efficacité de l'application.

En somme, ce projet a offert une expérience riche en apprentissages. En tirant parti des leçons extraites, notamment sur la gestion de projet et la communication, il sera possible d'améliorer considérablement la mise en œuvre de futurs projets collaboratifs et d'assurer une meilleure efficacité collective.

## **Sources**

-Données de LiChess : <https://database.lichess.org/>

-Exemples de projets et documentation :-https://lichess.org/source -https://github.com/jcw024/lichess database ETL -<https://blog.scottlogic.com/2017/09/01/apache-spark-meets-chess.html>

-ECO : <https://www3.diism.unisi.it/~addabbo/ECO_aperture_scacchi.html>

-<https://docs.spring.io/spring-boot/index.html>

-<https://legacy.reactjs.org/docs/getting-started.html>

-<https://docs.docker.com/>

-<https://docs.oracle.com/en/java/>

-<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_expressions>

-<https://fr.wikipedia.org/wiki/%C3%89lagage_alpha-b%C3%AAta#:~:text=L'algorithme%20minimax%20effectue%20une,exploration%20partielle%20de%20l'arbre>.

Code du projet:

-back : <https://gitlabvigan.iem/groupe1/ChessProject_BACK.git>

-etl <https://gitlabvigan.iem/groupe1/ChessProject_ETL.git>

-front <https://gitlabvigan.iem/groupe1/ChessProject_FRONT.git>

Le code se trouve sur la branche main.