PROJET TUTORÉ

**Rapport n°1bis**

**Définition des rôles, élaboration des spécifications techniques et fonctionnelles**

ligne horizontale

# 

ligne horizontale

Responsable : POUZIN Pierre-Emmanuel Groupe n°1

Table des Matières

[**1) Contexte du projet et de ses objectifs 4**](#_r7v84nl4swla)

[**2) Fonctionnalités 4**](#_s97yfbjc3ve6)

[2.1 - Front 4](#_3zlwjx8eb5f1)

[2.2.1 Description 4](#_bygirsf760vi)

[2.2.2 Technologies 6](#_thl1xijcs5pc)

[2.2.2.1 React 6](#_fdtdhtrbh83h)

[2.2.2.2 TypeScript 6](#_3ye0qetyn7pw)

[2.2.2.3 Vite 7](#_ungh0anqjvlw)

[2.2.3 Dépendances 8](#_to9tplje6t6x)

[2.2 - Back 8](#_61ss5mf6q1iy)

[2.2.1 Description 8](#_kz363mte7egy)

[2.2.2 Technologies 8](#_4xk9gcetfxdf)

[2.2.2.1 Java avec Spring Boot 8](#_ykqv1hjgf62b)

[2.2.2.2 JPA (Java Persistence API) 8](#_iu1uqjyiej3s)

[2.2.2.3 JDBC (Java Database Connectivity) 8](#_wtjox8axrzys)

[2.2.2.4 Maven 9](#_io7whbtg6zt4)

[2.2.3 Fonctionnalités principales du Backend 9](#_dinuv5xn6d3h)

[2.2.4 Dépendances 9](#_h93bhdjvlc5n)

[Dépendances avec le Frontend 9](#_wctt1s2pssb)

[Dépendances avec l'ETL 9](#_ga38v59hpwpz)

[2.3 - ETL 9](#_a8iql6zf6sf9)

[2.3.1 - Description 9](#_5ytvn5uddawp)

[2.3.2 - Structure 10](#_rs8baw5kxz05)

[2.3.2.1 - Téléchargement 10](#_llmvk8piv2p4)

[2.3.2.2 - Lecture 10](#_exznkpmcgcuv)

[2.3.2.3 - Formatage 11](#_z7fi0qtet3cr)

[2.3.2.4 - Insertion 11](#_ir7bqj9rosag)

[2.3.3 - Technologies 11](#_5ty44tvh1ef8)

[2.3.4 - Dépendances 12](#_x11011m1f1u3)

[2.4 - Schéma BD 12](#_ialxjckt9qjr)

[2.4.1 Analyse des données 12](#_qyi5ove6lyx3)

[2.4.2 Tables 16](#_xgonuanpgb7r)

[2.5 - API 17](#_jhp6nk3cuexs)

[2.6 - Contraintes liées au stockage et à l'API 19](#_apfjboas2ftf)

[2.7 - Oauth 20](#_tzzj4wizxora)

[**3) Environnement de développement 21**](#_o6prztkofy4d)

[3.1 - Git 21](#_lyxa0q75t6z4)

[3.1.1 - Structure des Repositories 21](#_lhpl8axr7e1x)

[3.1.2 Stratégie de branches 21](#_wqc8qy665ea2)

[3.1.3 Déploiement et CI/CD 22](#_w548yu20bm5l)

[3.1.4 Bonnes pratiques Git 22](#_7w8gzaxi15nc)

[3.2 - Docker 23](#_9uay3fkoibmk)

[3.2.1 Motivations du choix de Docker 23](#_h3vxdggzlc1t)

[3.2.3 Avantages de Docker pour les déploiements via GitLab Runner 24](#_974dypwngtsi)

[3.3 - Serveurs 24](#_p51khu4p1eiz)

[**4) Organisation de groupe 25**](#_i0f5ff2cwxla)

[4.1.1 Rôles 25](#_8pw0edhgd8ma)

[4.2.2 Répartitions des tâches 26](#_y6rzuuq4rsjy)

[Groupe BACK/FRONT : 26](#_45oz89fa8sm)

[Groupe ETL: 27](#_vivdmvw4scwp)

[1. Phase Extraction 27](#_xxz2mdgeq34y)

[2. Phase Transformation 27](#_3p837mdi7ba)

[3. Phase Chargement 28](#_212ajfmc0y8p)

[4. Supervision 28](#_tt9vsh81irh0)

[**5)SOURCE 29**](#_94b9zkma64tr)

## 

## 1) Contexte du projet et de ses 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.

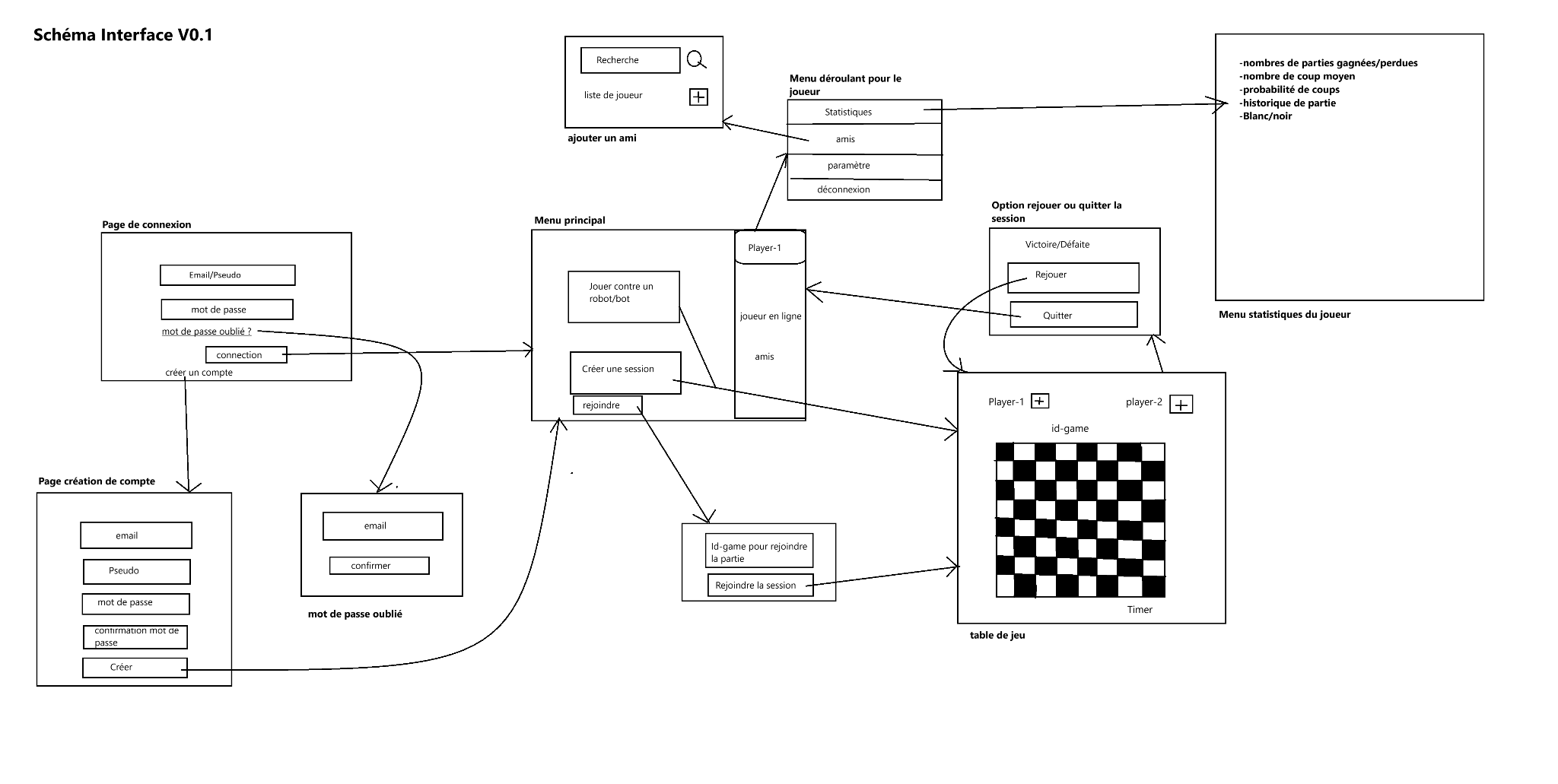
## 2) Fonctionnalités

### 2.1 - Front

#### 2.2.1 Description

Le front-end est la partie visible et interactive de l’application que les utilisateurs vont manipuler. C’est un aspect crucial pour l’expérience utilisateur, car il impacte l’apparence de l’interface, mais aussi la fluidité et la réactivité des intéractions.

Schéma d’interface v0.1:



#### 2.2.2 Technologies

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.

##### 2.2.2.1 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.

##### 2.2.2.2 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.

##### 2.2.2.3 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.

#### 2.2.3 Dépendances

Le front-end et le back-end seront connectés de manière à ce que le back-end récupère les données, les traite, puis les envoie au front-end pour affichage. Le front-end enverra également au back-end les données des parties jouées , gagnées et perdues par l'utilisateur. Elles seront également traitées puis affichées dans le front et stockées dans la base de données.

### 2.2 - Back

#### 2.2.1 Description

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. 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.

#### 2.2.2 Technologies

##### 2.2.2.1 **Java** avec **Spring Boot**

***Spring Boot*** est une solution souvent utilisée pour le développement d'applications web en Java grâce à son framework léger et sa gestion efficace des dépendances. Il permet de gérer le routing des requêtes entre le frontend et la base de données, ainsi que l'accès aux différents services internes.

***Spring Boot*** est utilisé pour créer l'API RESTful qui permet au frontend de communiquer avec le serveur. Il gère les requêtes HTTP, les réponses, l'authentification des utilisateurs et l'accès aux fonctionnalités comme la consultation des statistiques et des recommandations de coups.

##### 2.2.2.2 **JPA** (Java Persistence API)

**JPA** est une spécification standard pour la gestion des données relationnelles en Java. Il permet de mapper les objets Java aux tables de la base de données via une abstraction du modèle relationnel. Cela offre plus de flexibilité et de contrôle sur les opérations de persistance, en permettant aux développeurs de définir eux-mêmes leurs requêtes et transactions via **EntityManager.**

##### 2.2.2.3 **JDBC** (Java Database Connectivity)

Permet une connexion fluide et une gestion robuste des requêtes à la base de données relationnelle, essentielle pour la récupération et l'insertion rapide des statistiques et des recommandations de coups en fonction des parties.

La communication avec la base de données relationnelle sera assurée via **JDBC**. Ce choix permet une interaction directe avec la base de données, garantissant un contrôle fin des transactions et des opérations CRUD (Create, Read, Update, Delete). JDBC est parfaitement intégré à Spring Boot, facilitant ainsi la gestion des connexions à la base de données et des requêtes SQL.

##### 2.2.2.4 **Maven**

Outils utilisés pour la gestion des dépendances et l’automatisation du build du projet. ***Maven*** simplifient le processus d’intégration des librairies externes et d’outils tout en facilitant la gestion des versions de celles-ci.

#### 2.2.3 Fonctionnalités principales du Backend

1. **API REST** : Fournit des points d’accès RESTful pour que le front-end puisse communiquer avec le serveur.
2. **Traitement des données** : Gère l’intégration des données venant de l’ETL ou d'autres sources externes, et l'enregistrement dans la base de données.
3. **Connexion avec la base de données** : Assure la persistance des données à travers une couche d'abstraction (JPA/Hibernate).
4. **Intégration avec des systèmes externes** : Communication avec des services ETL ou autres systèmes externes via des API ou des systèmes de messagerie.

#### 2.2.4 Dépendances

##### Dépendances avec le Frontend

**API REST** : Le frontend interagit avec le backend via des endpoints **REST** exposés par Spring Boot. Ces endpoints permettent de récupérer les statistiques des joueurs et des parties, ainsi que les recommandations de coups basées sur les calculs effectués par le backend.

##### Dépendances avec l'ETL

**ETL (Extract, Transform, Load)** : Le programme ETL est chargé d'extraire les données brutes des parties jouées sur LiChess, de les transformer en un format exploitable et de les charger dans la base de données. Le backend consomme ces données pour fournir des statistiques aux utilisateurs.

### 2.3 - ETL

#### 2.3.1 - Description

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).

##### 2.3.2 - Structure

La structure de notre ETL suivra le processus suivant :

**Étape 1 :** Les données sont téléchargées depuis LiChess.

**Étape 2 :** Les données sont lues afin d'en retranscrire les informations.

**Étape 3 :** Les données lues sont traitées et formatées.

**Étape 4 :** Les données traitées sont insérées dans notre base de données.

###### 2.3.2.1 - Téléchargement

Bien qu’un module de téléchargement des fichiers ne soit, pour le moment, pas envisagé, ce dernier pourrait s’avérer pertinent. En effet, il permettrait de répondre aux problématiques de temps et de stockage citées précédemment. L’automatisation de cette tâche utiliserait l’espace de stockage comme un tampon dans lequel les fichiers se succéderont en maximisant l’utilisation de l’espace mis à notre disposition. Une première gestion “primitive” des doublons pourrait s’effectuer à son niveau.

De plus, ce module enrichirait notre jeu de données au maximum lorsque notre application aura atteint un stade de maturité suffisant. L’intérêt majeur de ce module est d’éviter de longues et fastidieuses opérations manuelles (temps de téléchargement, de décompression, gestion des fichiers/de l’espace). Une [implémentation](https://github.com/jcw024/lichess_database_ETL/) d’un tel module a été effectuée en Python par Joe Wong. Bien qu’il fasse partie d’un programme ETL pour les données Lichess, il est envisageable de le réutiliser à nos fins.

###### 2.3.2.2 - Lecture

Le module de lecture des données aura pour tâche de lire les fichiers .pgn afin de retranscrire l’ensemble des informations dans des objets java. Ce dernier doit implémenter le standard PGN pour assurer la cohérence et l’intégrité des données manipulées. Il doit également être résistant aux erreurs telles que des données erronées ou incorrectement formatées. Les fichiers pourront être parsés avec les outils fournis par java (Reader, Token, Regex…).

Une implémentation de ce module manuelle est envisageable. Néanmoins, de nombreux projets contiennent une implémentation d’un tel module. La plus intéressante semble être celle de Bernhard Seybold dans [Chesspresso](https://github.com/BernhardSeybold/Chesspresso/tree/master) comportant de quoi lire et écrire des parties d’échecs au format PGN. Il existe également celles de Frederic Cuillandre ([pgn-parser)](https://github.com/supareno/pgn-parser) et de Joe Wong ([lichess\_database\_ETL](https://github.com/jcw024/lichess_database_ETL/tree/main/)). Les deux premières sont écrites en java mais sont relativement anciennes. Quant à l’implémentation de Joe Wong, elle est bien plus récente mais écrite en python. Dans le cas où les implémentations existantes ne s’avèrent pas satisfaisantes, elles pourront nous servir d’exemple afin de faciliter et d'accélérer le temps de développement.

Améliorations possibles, nous pouvons diviser nos fichiers d'entrée, pour pouvoir effectuer notre lecture en multithreading.

Nous avons pris en note la possibilité par la suite d’utiliser nos notion en systèmes distribués pour améliorer la rapidité de lecture des données cela semble assez compliqué à mettre en place.

###### 2.3.2.3 - Formatage

Le module de formatage se chargera de traiter les données lues et chargées au préalable. Ce dernier assurera que les données lues et chargées dans nos objets java sont conformes à notre schéma de données. Le schéma de notre base de données intègre tous les champs présents dans le format PGN. De ce fait, cette étape du traitement ne sera qu’une sécurité contrôlant par exemple des valeurs trop longues ou contenant des caractères spéciaux.

###### 2.3.2.4 - Insertion

La dernière étape dans le traitement des données de notre ETL sera leur insertion dans notre base de données. À ce stade, les données auront été formatées pour correspondre à notre schéma et leur contenu aura été validé. Une problématique pouvant être traitée à cette étape est la duplication des données. La présence de deux parties identiques dans notre base est un gâchis de ressources. Or, la détection d’un tel doublon nécessite un parcours séquentiel de nos parties associé à une comparaison de leurs contenus. Quand bien même l’échantillon de parties serait diminué avec une sélection sur critères (date, joueurs, …), cette solution ne semble pas idéale.

#### 2.3.3 - Technologies

De prime abord, nous avons discuté de la possibilité d’utiliser **Sparks** comme outil afin de faire le traitement des données à notre place. Mais après en avoir discuté, cela semble bien trop compliqué à mettre en place avec le peu de temps que nous avons à notre disposition, en plus des difficultés de cet outil.

De ce fait, nous utiliserons soit JDBC soit JPA. JDBC est une API “bas niveau” où la communication avec la base de données se fait par des requêtes SQL rédigées directement dans le code. Bien que ce type d’API soit connu de la majorité des membres du groupe, ce dernier est très lourd au niveau de la syntaxe et peut s’avérer contraignant à maintenir. JPA est une surcouche à JDBC faisant abstraction de nombreux concepts. Dans cette dernière, il est possible d’effectuer une liaison entre le code java et la base de données via des POJO (Plain Old Java Object). Par le biais d’annotations dans le code java et/ou des fichiers xml, JPA effectue la correspondance entre les POJO et les structures de la BD. Grâce à cette dernière il est possible, par exemple, d’effectuer toutes les opérations CRUD sur nos données par le biais d’appels à des fonctions fournies par une implémentation de JPA (Hibernate, Eclipselink\_JPA, ...). L’utilisation de requêtes SQL rédigées dans des chaînes de caractères est toujours possible avec JPQL (langage de query implémenté dans JPA). JPA permet notamment de définir des requêtes comme des constantes identifiées par des alias.

#### 2.3.4 - Dépendances

Le programme ETL ne comporte pas réellement de dépendances avec d’autres composants de notre application. Il est évident que notre schéma de données doit être capable d’accueillir les données traitées par ce dernier. Les dépendances du programme ETL sont internes à ce dernier. En effet, les données traversent ce dernier de manière séquentielle, module par module. De ce fait, il peut être fastidieux de paralléliser le développement de ce celui-ci. Néanmoins, il est possible d’anticiper le produit de chaque module pour esquisser le fonctionnement général du programme.

[Discussion sur parallélisation ETL -> Hadoop, Akka, MPI avec Corba, multithreading java]

### 

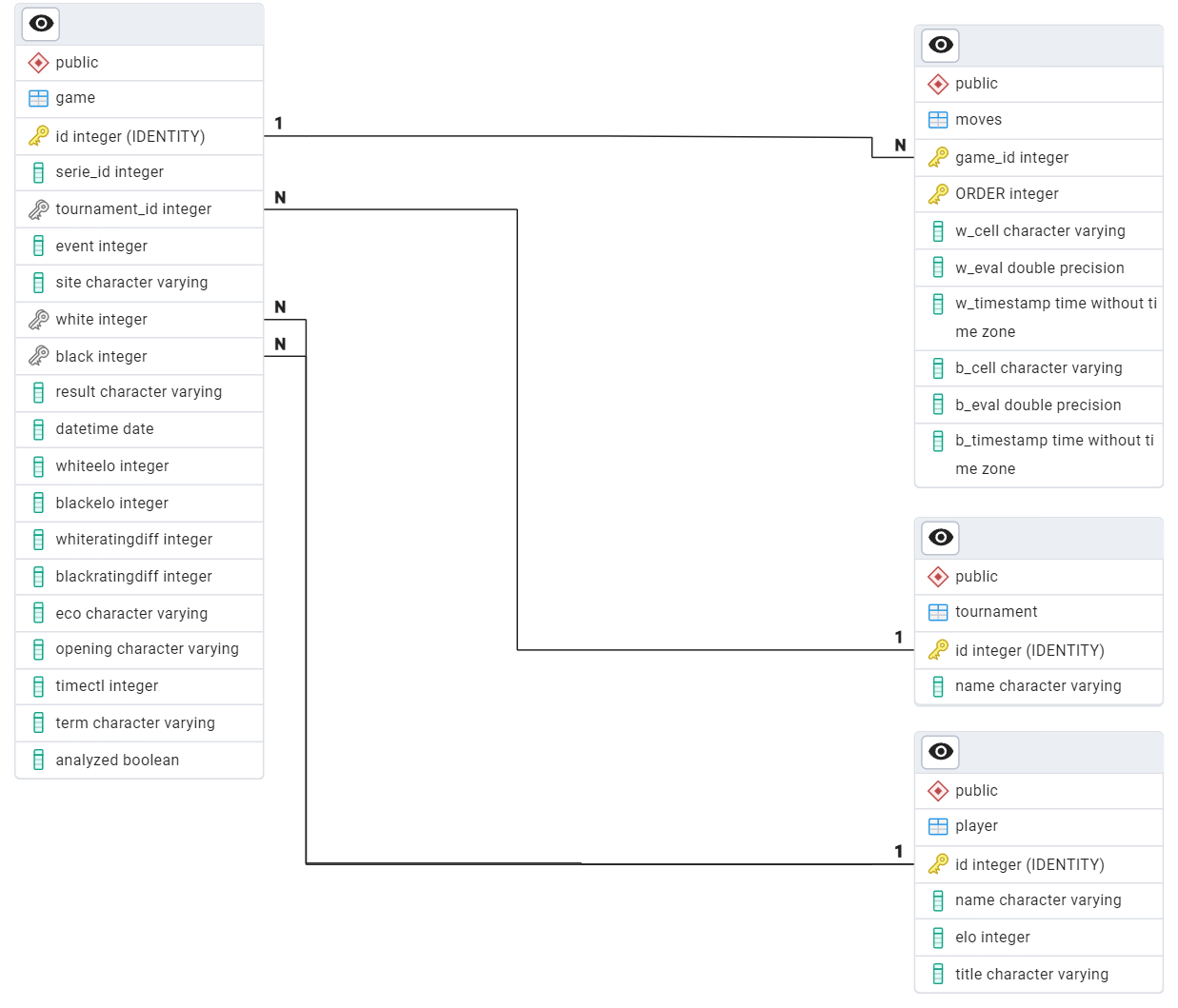
### 2.4 - Schéma BD

#### 2.4.1 Analyse des données

Notre premier schéma de données, créé à partir du fichier PGN que nous récupérons de LiChess, sera conservé comme base qui sera par la suite peaufinée en analysant les données que l’on va pouvoir obtenir.

| [Event "Rated Bullet tournament https://lichess.org/tournament/yc1WW2Ox"] [Site "https://lichess.org/PpwPOZMq"] [Date "2017.04.01"] [Round "-"] [White "Abbot"] [Black "Costello"] [Result "0-1"] [UTCDate "2017.04.01"] [UTCTime "11:32:01"] [WhiteElo "2100"] [BlackElo "2000"] [WhiteRatingDiff "-4"] [BlackRatingDiff "+1"] [WhiteTitle "FM"] [ECO "B30"] [Opening "Sicilian Defense: Old Sicilian"] [TimeControl "300+0"] [Termination "Time forfeit"]  1. e4 { [%eval 0.17] [%clk 0:00:30] } 1... c5 { [%eval 0.19] [%clk 0:00:30] } 2. Nf3 { [%eval 0.25] [%clk 0:00:29] } 2... Nc6 { [%eval 0.33] [%clk 0:00:30] } 3. Bc4 { [%eval -0.13] [%clk 0:00:28] } 3... e6 { [%eval -0.04] [%clk 0:00:30] } 4. c3 { [%eval -0.4] [%clk 0:00:27] } 4... b5? { [%eval 1.18] [%clk 0:00:30] } 5. Bb3?! { [%eval 0.21] [%clk 0:00:26] } 5... c4 { [%eval 0.32] [%clk 0:00:29] } 6. Bc2 { [%eval 0.2] [%clk 0:00:25] } 6... a5 { [%eval 0.6] [%clk 0:00:29] } 7. d4 { [%eval 0.29] [%clk 0:00:23] } 7... cxd3 { [%eval 0.6] [%clk 0:00:27] } 8. Qxd3 { [%eval 0.12] [%clk 0:00:22] } 8... Nf6 { [%eval 0.52] [%clk 0:00:26] } 9. e5 { [%eval 0.39] [%clk 0:00:21] } 9... Nd5 { [%eval 0.45] [%clk 0:00:25] } 10. Bg5?! { [%eval -0.44] [%clk 0:00:18] } 10... Qc7 { [%eval -0.12] [%clk 0:00:23] } 11. Nbd2?? { [%eval -3.15] [%clk 0:00:14] } 11... h6 { [%eval -2.99] [%clk 0:00:23] } 12. Bh4 { [%eval -3.0] [%clk 0:00:11] } 12... Ba6? { [%eval -0.12] [%clk 0:00:23] } 13. b3?? { [%eval -4.14] [%clk 0:00:02] } 13... Nf4? { [%eval -2.73] [%clk 0:00:21] } 0-1 |
| --- |

On peut donc remettre notre premier schéma pour comprendre ce que l’on va faire évoluer.



Pour analyser des données, il nous faut savoir ce que l’on peut récupérer depuis les fichiers mis à disposition par LiChess, et comprendre comment les valeurs y sont stockées. Grâce aux requêtes API, nous pouvons récupérer les informations liées aux joueurs (détaillées ci-dessous) et les fichiers PGN nous permettent de récupérer le déroulement d’une partie.

1) Joueur

Il faut d’abord pouvoir stocker les joueurs et leur données donc cette table contient les informations sur les joueurs : leur nom, leur classement Elo et leur titre.

Grâce aux informations collectées nous allons pouvoir traiter ces données pour en sortir des statistiques ce qui va faire évoluer notre modèle que nous avions conçu à la base.

A partir de la colonne ELO on va pouvoir créer un classement au fil des parties selon leur niveau.

On peut aussi analyser les performances dans les joueurs selon le titre qu’il possède(GM,NM). Cela nous amène à nous poser la question suivante: est-ce qu'un joueur titré est plus performant qu’un joueur non titré?

On pourrait aussi penser à avoir un historique de confrontations, savoir le nombre de parties jouées contre ce joueur et le pourcentage de victoire/défaite.

2) Game

Il faut maintenant comprendre les données dont nous avons besoin pour la table **"Game"**. Elle va contenir les informations essentielles de chaque partie : les joueurs qui s'affrontent, le résultat, le lieu où la partie s'est déroulée (Lichess, un autre site ?), la date, les classements Elo des joueurs, leur différence de classement, l'ouverture de la partie, et la manière dont la partie s'est terminée.

À partir des données stockées, nous pourrons calculer la performance moyenne des joueurs en fonction de leur Elo. Par exemple, nous pourrons déterminer si un joueur avec un Elo inférieur bat souvent des adversaires mieux classés. De plus, nous pourrons étudier l'impact des ouvertures sur les résultats des parties. Il sera possible de connaître, en moyenne, le nombre de coups nécessaires pour gagner ou perdre une partie.

Nous allons également analyser quelles ouvertures sont les plus efficaces pour chaque joueur, en tenant compte de leur Elo, et identifier les ouvertures qui conduisent à de meilleurs résultats. Cette analyse peut révéler des tendances intéressantes, comme la corrélation entre certains types d'ouvertures et le succès des joueurs.

En outre, nous pourrons examiner les classements Elo des joueurs en fonction de leurs performances, et ainsi visualiser leurs évolutions à l'aide de graphiques et de courbes. Cela nous permettra de suivre la progression des joueurs dans le temps et d'identifier des périodes de forte performance ou de déclin.

Enfin, nous pourrions aussi explorer des statistiques avancées, telles que le taux de succès par ouverture et le nombre moyen de coups par victoire, pour fournir une vue plus complète des performances des joueurs et des dynamiques de jeu.

On pourrait aussi considérer l’impact du temps sur une partie pour aller plus loin.

La table Game contiendra aussi un attribut hash\_code permettant de fournir un hash unique en prenant en compte les détails de la partie jouée. Cet attribut permettra de pouvoir insérer des parties tout en regardant si elles existent déjà dans la table.  
En effet, nous aurons juste à regarder le hashcode des parties stockées pour voir si cette partie existe déjà dans la table.

3) Moves

On aura aussi besoin d’une table “Moves” on aura besoin de stocker le numéro d'ordre du mouvement dans la partie.On aura aussi besoin de la position du coup effectué par les blancs en notation algébrique(ex e4,Nf3), de même avec les noirs.On aura aussi le temps écoulé depuis le début de la partie pour chaque couleur et l’évaluation de la position.

On peut imaginer un graphique montrant l’évolution de l’avantage pour chaque joueur durant la partie.On peut aussi mesurer la qualité d’un coup( ex : bon si le coup n’a pas fait chuté évaluation).On peut étudier si le joueur a utilisé un tactique en regardant si l'évaluation change brusquement.On peut aussi analysé le temps utilisé suivant la position.On peut aussi détecter l’ouverture joueur par les joueurs et savoir si elle est théorique.

4) Tournament

A partir du fichier PNG on peut savoir si le joueur est en tournois. Il faudra donc stocker le nom du tournoi et peut être le type. On pourra donc faire quelque statistique du type si le joueur est plus performant en tournois qu’en partie normals.Cette table va surement évoluer.

#### 

#### 2.4.2 Tables

Voici l'évolution de nos tables en tenant compte des données à stocker. Ce schéma de base de données est encore en cours de développement : certaines statistiques seront générées de manière dynamique à l'aide de requêtes, tandis que d'autres seront directement stockées.Pour avoir plus de détails sur les tables un wiki a été créé pour ça.



Afin d’obtenir la suite de coups correspondant à l’ouverture de la partie, nous prévoyons de créer une table “opening” qui contiendra toutes les ouvertures possibles avec la suite de coup qui y correspond. Cette table sera liée à notre table game via la clé “eco”, qui est le code associé aux ouverture.  
Nous avons extrait les données depuis une ressource pgn[[1]](#footnote-0) disponible sur internet et prévoyons d’insérer la données dans notre base via l’exécution d’un script, probablement écrit en python.   
La liste des ouvertures est fixe, aussi cette partie du projet n’as pas vocation à être exécutée plus d’une fois.

### 2.5 - API

Nous allons utiliser l’API de LiChess afin de récupérer les différentes données des utilisateurs.

Pour se faire, l’utilisateur devra lier son compte directement depuis notre application à LiChess avec un bouton comme “Se Connecter à LiChess”.  
De ce fait, nous n’avons pas à avoir les informations personnelles des utilisateurs LiChess mais nous pouvons tout de même récupérer toutes les statistiques de cette personne.  
  
Voici quelques exemples de ce que nous pouvons récupérer grâce à cette API:

1. Informations sur les utilisateurs

**Profil du joueur** : Chaque joueur dispose d’un profil comprenant son nom d’utilisateur, un éventuel titre FIDE (Maître International, Grand Maître, etc.), ainsi qu'une courte biographie.

**Statistiques du joueur** : Ce segment inclut le nombre de parties jouées, les parties gagnées, et le classement actuel du joueur dans différents formats de jeu (blitz, rapide, bullet, échecs classiques, etc.).

**Historique des parties** : Liste des parties jouées, accompagnée des résultats, des noms des adversaires, et des classements Elo des deux joueurs au moment de chaque partie.

**Données des parties en cours** : Informations sur les parties que le joueur est en train de disputer.

**Performances spécifiques** : Données détaillées sur les performances du joueur dans divers formats comme le blitz, le bullet, le rapide, les échecs classiques, ou des variantes comme les échecs 960.

2. Parties d'échecs

**Téléchargement des parties** : Possibilité de télécharger les parties jouées au format PGN (Portable Game Notation) pour les analyser ou les sauvegarder.

**Détails sur une partie spécifique** : Chaque partie peut être examinée en détail, avec l’historique des coups, la gestion de l’horloge, et le nom du vainqueur.

**Parties en direct ou terminées** : Visualisation des parties actuelles ou passées d’un joueur particulier.

**Parties filtrées** : Recherche de parties selon des critères spécifiques comme la période, le type de jeu, la variante jouée, ou la couleur jouée par l’utilisateur.

3. Événements en direct et tournois

**Tournois en cours** : Accès aux informations sur les tournois en cours, y compris le nom du tournoi, la liste des joueurs inscrits, et d'autres détails.

**Tournois passés** : Détails sur les tournois passés et leurs résultats.

**Suivi des événements en direct** : Flux d'événements pour suivre en temps réel les coups joués et les positions dans les parties ou les tournois en direct.

4. Challenges (Défis)

**Créer des défis** : Fonctionnalité permettant de créer des défis pour jouer contre d'autres joueurs en ligne.

**Accepter/refuser des défis** : Option pour accepter ou refuser les défis envoyés par d'autres joueurs.

5. Stream et diffusion

**Détails sur les streams** : Informations sur les streams en direct concernant les échecs, qu’il s’agisse de parties, de commentaires ou d’analyses.

**Liste des parties diffusées** : Visualisation des parties actuellement diffusées ou en cours de commentaire.

6. Messages et notifications

**Notifications** : Alerte pour informer l'utilisateur des nouveaux messages ou des événements en lien avec son profil, comme une invitation à un tournoi.

**Messagerie** : Option pour envoyer et recevoir des messages privés d'autres joueurs.

7. Statistiques et analyses

**Analyse des parties** : Analyse détaillée des parties jouées avec l'évaluation des coups par un moteur d’échecs, identifiant les meilleurs coups, les erreurs, et offrant une évaluation globale de la partie.

**Statistiques de performance** : Informations détaillées sur les mouvements effectués pendant une partie, les erreurs commises, et les évaluations des positions.

8. Compétitions

**Informations sur les compétitions d’équipes** : Liste des équipes inscrites dans des compétitions, avec leurs résultats.

**Classements des équipes** : Classements et performances des équipes dans des tournois ou compétitions spécifiques.

9. Événements de puzzle

**Accès aux puzzles** : Disponibilité de puzzles d’échecs, avec des statistiques sur les réussites, le niveau de difficulté, et d'autres données liées à ces exercices.

**Historique des puzzles** : Liste des puzzles résolus par un utilisateur, accompagnée de statistiques de performance.

10. Statistiques sur les ouvertures

**Performances des ouvertures** : Analyse des ouvertures jouées par un utilisateur, avec des informations sur leur fréquence et leur efficacité.

11. Flux de données en temps réel (WebSockets)

**Suivi en direct** : Suivi des parties en temps réel avec notifications instantanées lors de chaque coup joué ou changement de position.

**Notifications en direct** : Alerte en temps réel pour chaque nouveau jeu ou événement de la partie.

12. Communauté et interactions

**Clubs et équipes** : Informations sur les clubs d'échecs et les équipes auxquelles un utilisateur peut appartenir, ainsi que les classements des membres.

**Classements des équipes** : Détails sur les classements d’équipes et les membres.

13. Données sur les variantes

**Statistiques des variantes** : Performances et statistiques sur les variantes d'échecs comme les échecs 960, la horde, ou l’antichess.

Nous allons nous concentrer principalement sur la collecte et l'analyse des statistiques des joueurs ainsi que des parties jouées via l'API de Lichess. Cependant, nous devons prendre en compte plusieurs contraintes liées au stockage et aux limites d'utilisation de l'API.

### 2.6 - Contraintes liées au stockage et à l'API

Contraintes de stockage :

Capacité totale du serveur : 100 Go.

Répartition de l'espace :

80 Go seront dédiés aux données des parties jouées.

Environ 10 Go seront réservés aux informations relatives aux joueurs et à leurs statistiques.  
  
Le reste sera pour les mises à jour de la BD.

Contraintes liées à l'API Lichess :

*Nombre d'appels API :* Nous sommes limités à un appel API à la fois, ce qui signifie qu'il n'est pas possible de faire plusieurs requêtes simultanément.

*Limitation en volume par appel* ***:*** Lors de chaque appel API, nous pouvons récupérer un nombre limité de données. Par exemple, nous ne pouvons obtenir les informations que pour un maximum de 200 parties ou joueurs par appel.

*Mise à jour des données :* Nous pensons ajouter les utilisateurs qui se connectent depuis notre application (via LiChess) directement dans la base de données afin de mettre à jour les statistiques de tous “nos” joueurs, tous les mois par exemple.

Nous pensons donc faire des appels tous les X temps à l'API pour ne pas avoir de problèmes.

Nous étions partis sur un schéma de bases de données où les statistiques n’étaient pas incluses dans celui-ci et que l’on appellerait l’API à chaque fois.  
Mais au vu des limitations de l’API, l’intégration des statistiques doit se faire directement dans notre BD comme vu dans la partie 2.4.

### 2.7 - Oauth

OAuth (Open Authorization) est un protocole qui permet à une application d'accéder à des ressources protégées d'un utilisateur sur un autre service sans avoir à stocker ou manipuler directement les identifiants de l'utilisateur (comme un nom d'utilisateur et un mot de passe).

Dans notre cas, OAuth permet à notre application de rediriger l'utilisateur vers Lichess, pour qu'il s'authentifie et autorise notre application à accéder à certaines de ses données.

L'objectif de cette intégration est de permettre à un utilisateur de :

1. Se connecter à notre application en utilisant son compte Lichess.
2. Autoriser notre application à accéder à certaines informations de son compte (comme son adresse e-mail ou son profil etc).
3. Rédiger l'utilisateur vers notre application une fois connecté.

Cela nous permet également de réduire la charge de travail liée à la gestion de la connexion des utilisateurs. Notre idée initiale était que les utilisateurs sans compte Lichess créeraient un compte sur notre application, tandis que ceux qui possèdent déjà un compte Lichess pourraient simplement s'y connecter via leur compte existant.

## 3) Environnement de développement

### 3.1 - Git

Dans le cadre de notre projet, il nous a été imposé GitLab comme plateforme de gestion du contrôle de version et de développement collaboratif. Git nous permet de collaborer de manière efficace et structurée, tout en assurant une bonne gestion des différentes étapes de notre projet. Voici un aperçu de notre stratégie d'utilisation de Git pour chaque partie du projet.

#### 3.1.1 - Structure des Repositories

Nous avons opté pour la mise en place de **trois repositories distincts** :

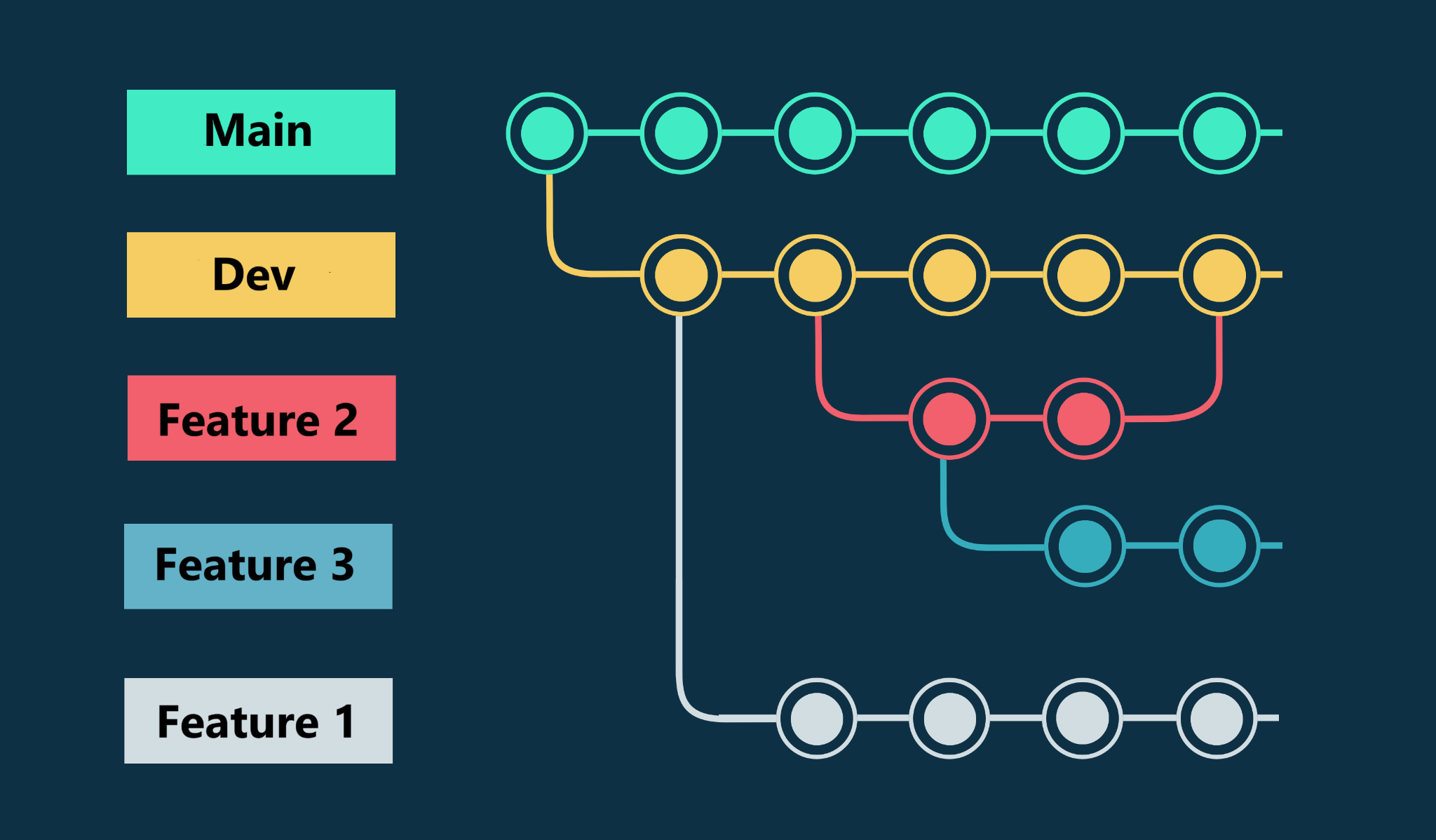
* **ChessProject\_FRONT** : pour le développement de la partie Frontend en react.
* **ChessProject\_ETL** : pour la gestion des tâches d’extraction, de transformation et de chargement des données issus de Lichess.
* **ChessProject\_BACK** : pour la partie Backend avec Springboot.

La raison pour laquelle nous avons choisi de ne pas utiliser un mono-repo (c’est-à-dire un seul repository pour tout le projet) est que chaque partie du projet a des besoins technologiques et des cycles de développement différents.   
Séparer les repositories nous permet d'éviter des conflits entre les dépendances des différentes sous-parties du projet, de faciliter les tests individuels, et d'assurer un déploiement plus modulaire et indépendant.

#### 3.1.2 Stratégie de branches

Notre stratégie de branche repose sur un modèle inspiré de **GitFlow[[2]](#footnote-1)** :

* **Main** : Les branches principales des repositories sont protégées. Elles représentent la version stable et déployée en production de chaque partie du projet. Aucun développement direct n'y est permis, et les seules “merges” possibles proviennent des branches **Dev** lors des déploiements finaux.
* **Dev** : Chaque repository possède une branche **Dev** qui est le point central de nos développements. Toutes les nouvelles fonctionnalités ou corrections de bugs partent de cette branche sous forme de sous-branches dédiées aux différentes fonctionnalités.
* **Feature** : Les branches **Feature** permettent de développer une fonctionnalité spécifique sans perturber le reste du projet. Une fois la feature implémentée, elle est fusionnée dans **Dev** après une revue de code.



Nous adoptons un format de commit standardisé afin de garder un historique clair des différents développements qui seront réalisés :

| git commit -m "[Feature]-Détails du commit" |
| --- |

#### 3.1.3 Déploiement et CI/CD

Nous mettrons en place des pipelines de **CI/CD** (Continuous Integration/Continuous Deployment) via **GitLab Runner**, ce qui nous permettra de tester et de déployer automatiquement nos modifications. Les branches **Main** déclencheront un pipeline pour le déploiement final en production, tandis que les branches **Dev** permettront un déploiement en environnement de test.  
Ce système nous permettra également d’assurer la qualité de nos développements, avec l'exécution automatique des éventuels tests unitaires que nous pourrions réaliser. Ces tests nous permettraient d’interrompre le déploiement s’ils venaient à ne pas s’exécuter convenablement.

#### 3.1.4 Bonnes pratiques Git

Pour garantir une collaboration fluide entre tous les acteurs du projet et éviter les conflits lors de nos fusions de branches, nous suivons les bonnes pratiques suivantes :

1. **Commits clairs et fréquents** : Chaque modification est accompagnée d'un commit descriptif en suivant le format défini. Cela permet de comprendre rapidement ce que chaque modification implique.
2. **Branches isolées pour chaque fonctionnalité** : Nous travaillons sur des branches spécifiques pour chaque nouvelle fonctionnalité ou correction de bug. Cela minimise les risques de conflits lors des fusions et assure une meilleure traçabilité.
3. **Revue de code** : Avant toute fusion dans les branches **Dev**, nous effectuons, lorsque le temps le permet, des revues de code. Cela permet d’améliorer la qualité du code et de partager les connaissances entre les membres de l’équipe.
4. **Tests automatisés** : Grâce aux pipelines CI/CD, des tests automatisés peuvent être exécutés à chaque push, garantissant que le code reste fonctionnel à chaque étape du développement. Ainsi, les tests font partie intégrante du processus de développement.

Cela conclut notre stratégie, assurant à la fois une bonne organisation du travail collaboratif et une gestion efficace des différentes phases de développement.

### 3.2 - Docker

Nous avons choisi d'utiliser **Docker** pour gérer nos environnements de développement et de déploiement. Chaque repository du projet (Frontend, Backend et ETL) dispose de son propre **Dockerfile** et de son **docker-compose.yaml**.  
L’objectif principal est d’éviter les conflits de version. 10 développeurs travaillant sur le même projet se doivent de travailler dans des environnements strictement similaires.

#### 3.2.1 Motivations du choix de Docker

1. **Portabilité** :
   * **Docker** permet de s'assurer que nos applications tournent de manière **identique** sur différentes machines. En encapsulant les dépendances et la configuration dans des **containers**, nous évitons les problèmes de compatibilité entre environnements. Qu'il s'agisse de nos postes de développement ou des serveurs de production, l'application fonctionnera toujours de la même manière.
2. **Isolation des services** :
   * Grâce à Docker, chaque service (Frontend, Backend, ETL) peut être isolé dans son propre container. Cela signifie que nous pouvons tester ou déployer une partie du projet sans affecter les autres. Cela réduit également les conflits entre bibliothèques ou configurations, qui pourraient survenir dans un environnement partagé.
3. **Déploiements simplifiés** :
   * Docker facilite les déploiements en **paquetant** notre application et toutes ses dépendances en un seul ensemble : une image Docker. Cela nous permet de livrer des containers prêts à être déployés sur les serveurs, sans avoir à se soucier de la configuration des serveurs en eux-même.
4. **Isolation des dépendances** :
   * Chaque repository utilise des dépendances spécifiques. Docker nous permet de les isoler dans chaque container, sans risquer des conflits entre les versions de ces dépendances, comme cela pourrait arriver sur une machine partagée.
5. **Scalabilité et gestion des versions** :
   * Avec Docker, il est facile de **scaler** nos services en fonction de la charge. Si une partie du projet (par exemple le Backend) nécessite plus de ressources à un moment donné, nous pouvons déployer plusieurs instances du container sans difficulté. Ceci étant, dans le cadre de ce projet, nous n’avons accès qu'à une seule machine de production, et déployer plusieurs instances d’un container peut apporter d’autres problèmes, notamment en termes de communication entre lesdits containers.

#### 3.2.3 Avantages de Docker pour les déploiements via GitLab Runner

Nous intégrerons Docker avec **GitLab Runner** pour faciliter l'intégration continue (CI) et le déploiement continu (CD). Voici les principaux bénéfices :

1. **Automatisation complète du pipeline** :
   * Avec Docker, nous pouvons facilement automatiser nos pipelines CI/CD grâce à **GitLab Runner**. À chaque push sur nos branches de développement ou de production, GitLab Runner exécute nos **Dockerfiles** (via nos différents docker-compose.yaml) pour construire les images, lancer les tests, puis déployer les containers sur nos serveurs de développement ou de production.
2. **Environnements cohérents** :
   * En utilisant Docker dans nos pipelines, nous nous assurons que les environnements de développement, de test et de production sont strictement identiques. Cela permet de minimiser les erreurs qui pourraient survenir à cause de différences entre les machines de développement des membres de l’équipe et les serveurs de production.
3. **Facilité de rollback** :
   * Grâce aux images Docker versionnées, si un déploiement échoue ou qu'un bug critique est détecté, nous pouvons facilement revenir à une version précédente de l’application en déployant l’image Docker d'une version antérieure.
4. **Scalabilité des pipelines CI/CD** :
   * Avec Docker, il est possible de paralléliser les tests ou les déploiements de différentes parties du projet (Frontend, Backend, ETL) en lançant plusieurs containers en parallèle via GitLab Runner. Cela accélère le processus de CI/CD.
5. **Gestion simplifiée des dépendances dans les pipelines** :
   * Comme chaque repository contient un **Dockerfile**, toutes les dépendances et configurations spécifiques aux différentes parties du projet sont encapsulées dans les containers. Cela signifie que GitLab Runner n’a pas besoin de connaître les détails des dépendances du projet ; il se contente de lancer l’image Docker spécifiée.

Cela conclut notre stratégie d'utilisation de Docker, avec un intérêt pour l'automatisation des différents processus de déploiement et la gestion simplifiée des environnements, notamment sur les machines des développeurs du projet.

### 3.3 - Serveurs

Dans le cadre de notre projet, deux vm nous sont fournis par l’UB via **Proxmox** : un serveur de développement et un serveur de production.

Le **serveur de développement** sera utilisé pour déployer nos versions intermédiaires et tester nos nouvelles fonctionnalités en conditions réelles, mais sans impacter la version stable du projet. Il servira de bac à sable pour expérimenter et corriger les bugs identifiés lors des revues de code et des tests CI/CD. Ici, la stabilité n’est pas une priorité et le service peut être défaillant.

Le **serveur de production**, quant à lui, sera réservé aux versions finales et stables de notre projet. Chaque déploiement sur ce serveur sera issu d'une fusion validée des branches **Dev** dans **Main**, garantissant que seules les fonctionnalités abouties et testées soient accessibles aux utilisateurs finaux.

Grâce à l'intégration de **GitLab Runner** et de **Docker**, nous assurons une continuité entre ces deux environnements, facilitant ainsi le processus de déploiement automatisé et garantissant la stabilité des versions en production.

## 4) Organisation de groupe

### 4.1.1 Rôles

Le **référent développeur** assure la cohérence des développements pour chaque fonctionnalité, en s’assurant que tous les développeurs respectent les mêmes standards.

Il est essentiel pour que le code du back\_end front\_end et ETL soit uniforme, bien structuré et fonctionne correctement ensemble.Cela permet d’éviter les incohérences entre les différentes parties du projets.

Le **référent données** assure que les données sont cohérentes à toutes les étapes de l’application, de la collecte à l'utilisation.Pour l’ETL et le back , ce rôle est primordial pour garantir que les données sont correctement extraites,transformées et chargées.

Le **référent qualité** garantit que les tests sont en accord avec les fonctionnalités et que les développements respectent les exigences de qualité.Il est indispensable pour s’assurer que le code du back,front et ETL est bien tester et fonctionnent sans erreur, cela permet de réduire les bugs et d’assurer un bonne expérience utilisateur.

### 

### 4.2.2 Répartitions des tâches

Les premières tâches ont déjà été planifiées pour préparer l'environnement de travail, le traitement des données et la conception de l'application.

Nous avons décidé de séparer le travail entre 2 groupes: Le Back/Front et l’ETL afin de ne pas travailler tous sur la même chose afin d’avancer plus vite tout en ayant une vraie structuration d’équipe.

#### **Groupe BACK/FRONT :**

Ce groupe sera chargé de la mise en œuvre de la partie graphique et fonctionnelle de notre application.  
Elle devra en outre créer l’application visuelle et fonctionnelle d’un jeu d’échec

Nous allons principalement nous concentrer sur la partie Back, ou nous essayerons de mettre en place un jeu de données test.

Ce groupe sera composé des personnes suivantes: GIE Jimmy, POUZIN Pierre-Emmanuel, POULET Alexandre, VOUETTE Maxence,

Les tâches suivantes seront réparties entre les membres du groupe.

**Tâche 1 :** Installer et configurer **SpringBoot** et l'**API RESTful** pour gérer la connexion front-back pour créer les routes entre le front et le back de l'application.

**Tâche 2 : Mettre en place la base de données (BDD)**

Créer les schémas de données nécessaires.

Définir les relations entre les tables.

**Tâche 3 : Développer un script de migration**

Écrire un script pour importer un premier jeu de données dans la BDD.

Tester et valider le script de migration.

**Tâche 4 : Effectuer des jeux de tests**

Rédiger et exécuter des tests unitaires pour chaque fonctionnalité.

Mettre en place des tests d'intégration pour vérifier l'ensemble du système**.**

**Tâche 5 : Configurer JPA**

Configurer Java Persistence API (JPA) pour gérer la persistance des données.

Mapper les tables de la base de données en objets Java.

**Tâche 6 : Configurer JDBC**

Mettre en place JDBC pour permettre la connexion et l'interaction avec la base de données.

**Tâche 7 : Intégrer l'API Lichess**

Implémenter le système d'authentification avec OAuth.

Faire les appels API nécessaires pour récupérer les données des parties et des joueurs.

#### 

#### **Groupe ETL:**

##### 1. Phase Extraction

**Effectif** : 2 personne

**Tâche principale** : Lecture des fichiers PGN et récupération des données.

Lecture des fichiers PGN (Portable Game Notation).

Interaction avec l’API LiChess si nécessaire pour récupérer des fichiers ou des données supplémentaires.

Gestion des erreurs liées à la lecture des fichiers (formats incorrects, fichiers corrompus, etc.).

**Livrables** : Des objets Java initialisés avec les données des fichiers.

##### 2. Phase Transformation

**Effectif** : 2 personnes

**Tâche 1 : Vérification du contenu des objets**

Identifier et supprimer les doublons ou les données inutiles (par exemple, certains types de parties non pertinentes).

Filtrer les parties en fonction des critères définis (type de jeu, durée, etc.).

**Tâche 2 : Remplissage des objets Java**

Utiliser un parseur PGN pour générer automatiquement des objets Java à partir des données extraites.

Assurer une gestion correcte des formats de données, notamment pour les dates.

**Livrables** : Des objets java dont les données ont été contrôlées.

##### 3. Phase Chargement

**Effectif** : 1 personne

**Tâche principale** : Insertion des données dans la base de données.

Vérifier l'existence des joueurs dans la base avant d'insérer une partie.

Ajouter de nouveaux joueurs si nécessaire et récupérer les clés étrangères pour les joueurs existants.

Insertion des parties dans la base tout en respectant l’intégrité des relations entre les tables.

**Attention** : Ne pas mettre à jour les statistiques des joueurs récupérées via l’API de LiChess pour éviter les incohérences.

**Livrables** : Un ensemble de données insérées avec succès dans la base sans doublons ou erreurs.

##### 4. Supervision

**Supervision**  : 1 personne

**Tâche principale** : S'assurer que les différentes phases de l'ETL se déroulent correctement et que la transformation ne commence qu’après l’extraction réussie.

**Coordination** : Synchronisation entre les équipes d'extraction, de transformation, et de chargement. Il surveille tout le monde, prend notes des problèmes et adapte le processus si nécessaire.

**Livrables** : Un processus ETL bien coordonné et sans erreurs majeures.

Ce groupe sera composé des personnes suivantes: STIZ Romain, GUIBARD Théo, GOURMAND Nicolas, ROUSSEAU Emile, REMBERT Lucas

## 5)SOURCE

-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>

1. https://www3.diism.unisi.it/~addabbo/ECO\_aperture\_scacchi.html [↑](#footnote-ref-0)
2. https://www.atlassian.com/fr/git/tutorials/comparing-workflows/gitflow-workflow [↑](#footnote-ref-1)