PARIS PANTHÉON-ASSAS UNIVERSITÉ

Rapport de soutenance

JORDY - DYLAN - JOHAN - EMMMANUEL - JOHN
B2 DEV PARO1 NODEJS

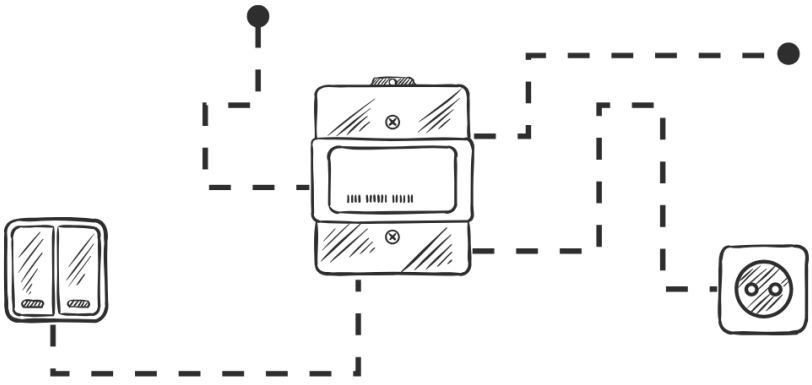


Table des matières

Introduction et Cahier des charges

Contexte du projet

Nous avons joué à League of Legends pendant toute notre enfance et nous avons un constat simple : il manque souvent un outil efficace permettant de suivre les performances des joueurs sur un jeu donné, que ce soit pour un usage personnel ou pour des compétitions entre amis.

Ce qui nous a motivé à mettre en place ce projet et aussi d'avoir une interface simple et ergonomique pour consulter rapidement notre statistique et celle des autres joueurs.

Ce jeu pour certains, était une source d'apprentissage de l'anglais et pour d'autres une grande découverte d'un bon milieu de jeu.

Dans l'ensemble nous sommes passionnés à s'entrainer en faisant du code raison pour laquelle on s'est lancé dans un tel projet afin de mettre aussi en avant notre passion.

En sommes, ce projet est aussi une opportunité d'appliquer des technologies modernes comme la gestion d'API et la mise en place d'une base de données performante

Objectif

LP Tracker est un site web conçu pour permettre aux joueurs de suivre leurs parties et celles des autres, d'enregistrer des statistiques et analyser leurs performances. Ce projet répond à un besoin spécifique des joueurs qui souhaitent garder un historique détaillé de leurs sessions de jeu ou, de pouvoir analyser leur partie de jeu.

Les fonctionnalités prévues sont :

1- **Recherche de joueur** : Permettre aux utilisateurs de rechercher un joueur via son pseudo et son serveur. Afficher les statistiques détaillées du joueur (rank, winrate, KDA, champions joués, etc.).
2- **Affichage des statistiques détaillées** : Récupération des données via notre API. Présentation des performances sur les dernières parties. Analyse des performances par champion joué.
3- **Comparaison de joueurs** : Fonctionnalité permettant de comparer les statistiques de plusieurs joueurs.

4- **Interface ergonomique et responsive :** Un design simple et efficace permettant une prise en main rapide.

Périmètres

Pour ce projet, nous avons défini un périmètre fonctionnel clair :

- Gestion des utilisateurs (inscription, connexion, profil)
- Création de parties et ajout de joueurs
- Enregistrement des scores
- Consultation de l'historique des parties
- Interface web ergonomique et adaptée aux différents supports

En termes de backend, nous avons mis en place une API exposant plusieurs endpoints, tels que :

- POST / auth/register : Inscription d'un utilisateur
- POST / auth/login : Connexion d'un utilisateur
- POST /games : Création d'une nouvelle partie
- GET /games : Récupération des parties existantes
- POST /scores : Ajout d'un score pour une partie
- GET /scores/ :gameId :Récupération des scores d'une partie

Conception et Architectures

Diagrammes de classe

Le diagramme de classe que nous avons conçu illustre la structure logique de notre application LPTracker. Il représente les entités principales, leurs attributs, les opérations associées, ainsi que les relations entre elles. Ce diagramme a été réalisé en UML pour faciliter la modélisation orientée objet et la correspondance avec la structure relationnelle de notre base de données PostgreSQL.

Structure et entités

Summoner : Représente un joueur. Il contient les informations d'identification telles que le `puuid`, le nom d'invocateur, la région, le niveau, l'icône de profil, les rangs classés (solo et flex) et les points de ligue (LP).

Champion : Entité correspondant aux champions jouables dans le jeu. Chaque champion possède un nom, un rôle, une description (`lore`), une image et des métadonnées (dates de création et de mise à jour).

Match : Représente une partie de jeu. Elle regroupe plusieurs participants et contient des informations comme la durée, le mode de jeu et le résultat.

MatchParticipant : Entité qui fait le lien entre un match, un joueur et un champion. Elle stocke les performances individuelles du joueur durant la partie (kills, deaths, assists, or gagné, dégâts, objets utilisés, etc.).

Champion_Mastery : Associe un joueur à un champion, en indiquant son niveau de maîtrise avec ce dernier.

Stats : Stocke les performances globales d'un joueur : nombre de victoires et de défaites, statistiques cumulées (kills, deaths, assists), rangs et remakes.

Items : Représente les objets du jeu. Chaque item possède un nom, une description, un prix, ainsi que des informations de création et de mise à jour.

Runes et **Spells** : Entités correspondant aux runes et sorts d'invocateur utilisés pendant les parties, comprenant un nom, une description, une image, et un taux d'utilisation.

Relations principales

Un Summoner peut participer à plusieurs Matchs via l'entité MatchParticipant.

Un **Match** regroupe plusieurs MatchParticipants.

Chaque MatchParticipant est lié à un **Champion**.

Une entité **Stats** est associée de manière unique à un Summoner.

Un Summoner peut avoir plusieurs niveaux de Champion_Mastery**.**

Les **Items**, **Runes** et Spells sont reliés aux MatchParticipants pour représenter leur utilisation en jeu.

Le diagramme a été réalisé à l'aide d'un outil de modélisation UML. Il a servi de base à la création des tables dans PostgreSQL ainsi qu'à l'implémentation des modèles objets (JavaScript/Node.js) et des routes de l'API.

Le diagramme complet est disponible en annexe de ce document pour une visualisation détaillée de la structure du système. [Voir annexe .1]

Commenté [JV1]: mettre annexe

Endpoints / Méthode de l'API

L'API REST développée dans le cadre de ce projet permet de gérer les différentes entités du système, notamment les joueurs (Summoners), les champions, les objets, les statistiques et les parties jouées. Elle a été construite en respectant les bonnes pratiques REST, avec une séparation claire des ressources et l'utilisation appropriée des verbes HTTP.

Vous trouverez en annexe un récapitulatif des principaux endpoints, accompagnés des méthodes HTTP utilisées, des données attendues dans le corps de la requête (body), ainsi que des réponses retournées par l'API (response). [Voir annexe .2]

Commenté [JV2]: ajouter annexe

Toutes les réponses de l'API sont retournées au format JSON. En cas de succès, un objet ou une liste d'objets est retourné(e). En cas d'erreur (404, 500, etc.), un objet contenant une clé error est retourné, avec un message explicite.

Architecture logicielle

L'architecture logicielle de l'application LPTracker repose sur une séparation claire entre le frontend, le backend et la base de données. Cette organisation facilite la maintenance, les évolutions futures du projet ainsi que les tests.

Frontend – Interface utilisateur

Le frontend a été développé avec le framework Next.js, qui repose sur React. Ce choix permet de bénéficier à la fois du rendu côté serveur (SSR) et de la génération statique des pages, optimisant ainsi les performances et le référencement.

L'interface utilisateur est construite avec une approche modulaire et réactive, et le style est géré à l'aide de Tailwind CSS. Ce framework utilitaire permet de concevoir des interfaces propres, responsives et cohérentes tout en réduisant la quantité de CSS personnalisée à écrire.

Le frontend interagit avec l'API via des requêtes HTTP asynchrones (fetch ou axios) pour afficher dynamiquement les données relatives aux joueurs, aux champions et aux statistiques.

Backend – API REST

Le backend repose sur Node.js avec le framework Express, utilisé pour construire une API RESTful modulaire, performante et maintenable. Chaque ressource (champions, joueurs, statistiques, objets…) est associée à un ensemble de routes organisées par fonctionnalité.

Pour le développement et les tests de l'API, nous utilisons Postman, un outil permettant de tester les différentes méthodes HTTP, de simuler des appels avec ou sans paramètres, et de valider la cohérence des réponses. Cela permet un développement itératif efficace et un débogage précis.

La logique métier est séparé en modèles, contrôleurs et routes.

Base de données – PostgreSQL

Toutes les données sont stockées dans une base de données PostgreSQL. On y retrouve les tables pour les joueurs (summoners), les champions, les objets, les matchs, les statistiques, etc. Chaque table est bien reliée aux autres grâce à des clés primaires et étrangères, ce qui permet de garder une structure claire et cohérente.
On utilise le module pg pour envoyer des requêtes SQL depuis le backend vers la base de données.

Intégration externe

Pour l'instant, l'application fonctionne uniquement avec les données stockées en base. Mais l'architecture est prête à évoluer. Par exemple, il serait possible d'ajouter une connexion avec l'API officielle de Riot Games pour aller chercher directement des données réelles (profils, performances, historique de matchs, etc.).

Étapes de développement

Organisation de l'équipe

Chaque membre a travaillé en parallèle sur les différentes couches de l'application : le frontend, le backend et la base de données.

Voici la répartition des tâches :

- Jordy s'est occupé du frontend et a également géré l'intégration de l'API dans l'interface. Il a mis en place les appels aux routes backend et l'affichage dynamique des données récupérées.

- John a travaillé avec Jordy sur la partie frontend, en participant à la construction des pages, à l'interface utilisateur et à la mise en forme avec Tailwind CSS.

- Johan et Dylan ont pris en charge une grande partie du backend, notamment la création des routes de l'API, la gestion des modèles, les contrôleurs et la mise en place des différentes fonctionnalités REST.

- Emmanuel s'est chargé de la base de données. Il a conçu le schéma relationnel, créé les tables, les relations et écrit les requêtes SQL nécessaires au bon fonctionnement de l'application.

Gestion du temps et des ressources

Le projet a été découpé en plusieurs étapes :

- Phase de conception : définition des besoins, modélisation de la base de données et création du diagramme de classe.

- Mise en place de l'architecture : séparation frontend/backend, initialisation des fichiers, préparation du serveur Express et de Next.js

- Développement parallèle :

  - Backend : création des routes, des modèles, des contrôleurs.

  - Frontend : intégration des pages, design avec Tailwind, appels API.

  - Base de données : création des tables, insertion de données, tests des requêtes.

- Tests et ajustements : utilisation de Postman pour tester l'API, vérification des connexions entre les différentes couches.

- Finalisation et nettoyage du code.

L'équipe a travaillé de façon collaborative tout au long du projet, en se répartissant les tâches selon les compétences de chacun et en se réunissant régulièrement pour faire le point et avancer ensemble.

Gestion de version (GIT)

Pour organiser notre code et collaborer efficacement, nous avons utilisé Git avec un hébergement sur GitHub. Chaque membre de l'équipe travaillait sur une branche dédiée à sa fonctionnalité.

Voici notre workflow :

La branche main contient la version stable et fonctionnelle du projet.

Chaque fonctionnalité (ex : affichage des champions, routes backend, configuration BDD) a été développée dans une branche spécifique.

Une fois le développement terminé, les membres de l'équipe créaient une pull request pour demander la fusion de leur branche avec main.

Les autres membres pouvaient alors relire le code, faire des commentaires ou approuver la modification.

Une fois validée, la pull request était merge dans main.

Ce fonctionnement a permis de travailler en parallèle, d'éviter les conflits, et de garder une base de code propre et bien organisée.

Choix techniques

Les technologies utilisées pour ce projet ont été choisies en partie sur les conseils de notre enseignant.

- Express.js (backend) : Recommandé pour sa légèreté et sa simplicité, Express permet de créer une API REST de manière rapide et claire. C'est un choix courant dans le développement web et il est bien adapté à une première approche d'une architecture serveur.

- Next.js (frontend) : Ce framework basé sur React permet de développer des interfaces modernes avec un bon rendu côté serveur (SSR) et une gestion efficace des routes. Il a été conseillé car il facilite la création de pages dynamiques tout en restant accessible.
- Tailwind CSS : Ce framework CSS utilitaire a été utilisé pour styliser rapidement l'interface sans avoir à écrire du CSS personnalisé. Il offre une bonne lisibilité du code HTML et permet de concevoir des interfaces responsives facilement.

- PostgreSQL (base de données) : Ce système de gestion de base de données relationnelle a été retenu pour sa fiabilité, sa structure rigoureuse et sa bonne compatibilité avec les requêtes complexes.

Design de la solution

Pour la conception de l'interface, nous avons choisi de nous inspirer du site u.gg, un site bien connu des joueurs de League of Legends. L'idée était de créer une interface intuitive, claire et agréable à utiliser.

Le site u.gg est souvent utilisé comme référence dans la communauté pour consulter des statistiques, des builds ou encore les performances des joueurs. Sa navigation est simple, ses informations sont bien organisées, et les pages sont rapides à comprendre. Nous avons donc repris cette logique pour organiser notre propre affichage : sections bien séparées, cartes claires pour les champions ou les joueurs, et une navigation fluide entre les pages.

Intégration / Déploiement

Nous n'avons pas utilisé Docker pour ce projet. Le lancement de l'application se fait localement :

- Le frontend (Next.js) est lancé avec la commande :

```
npm run dev
```

- Le backend (Express) est lancé via :

```
node index.js
```

Ou avec nodemon :

```
npx nodemon index.js
```

La base de données PostgreSQL est gérée via PGAdmin.

Mise en œuvre et démonstration

Exemples de requêtes / réponses

Pendant le développement du backend, nous avons utilisé Postman pour tester nos différentes routes. Cela nous a permis de vérifier que les données étaient bien envoyées, reçues et traitées.

Voici quelques exemples de requêtes et de réponses JSON :

Exemple 1 :

```
// Requête envoyée
{
  "itemName": "Potion de soin",
  "description": "Restaure 50 points de vie",
  "price": 150
}
```

```
// Réponse attendue
{
  "item_id": 1,
  "item_name": "Potion de soin",
  "description": "Restaure 50 points de vie",
  "price": 150,
  "created_at": "2024-03-25T12:34:56.000Z",
  "updated_at": "2024-03-25T12:34:56.000Z"
}
```

Exemple 2 :

```
[
  {
    "champion_id": 1,
    "champion_name": "Ahri",
    "role": "Mage",
    "lore": "Une renarde mystique...",
    "champion_image": "ahri.png",
    "created_at": "...",
    "updated_at": "..."
  },
  ...
]
```


Ces tests nous ont permis de valider la bonne communication entre le frontend et le backend.

Base de données (PGAdmin)

Pour la gestion de notre base de données PostgreSQL, nous avons utilisé l'outil PGAdmin. Il s'agit d'une interface graphique officielle fournie avec PostgreSQL, qui permet d'administrer la base facilement sans passer uniquement par le terminal.

- Grâce à PGAdmin, nous avons pu :
- Créer les tables de manière structurée (summoners, champions, items, stats, matchs, etc.) ;
- Définir les relations entre les entités à l'aide de clés étrangères ;
- Ajouter, modifier ou supprimer des données pour faire des tests manuels ;
- Exécuter des requêtes SQL directement depuis l'éditeur intégré ;
- Visualiser le contenu des tables et vérifier les jointures.

Avant d'implémenter le schéma dans PGAdmin, nous avons d'abord conçu un modèle conceptuel (MCD), puis nous l'avons transformé en modèle logique (MLD) et modèle physique (MPD). Cette démarche nous a permis de bien structurer notre base dès le départ et d'assurer une bonne cohérence dans les relations entre les entités. [Voir annexe p.]

Commenté [JV3]: Il s'appelle comment le modèle qu'on a fait nous ?

Commenté [JV4]: ajouter annexe + italique

L'utilisation de PGAdmin nous a donc permis de garder une vue claire sur l'état de la base tout au long du projet, de corriger rapidement des erreurs, et de tester les interactions avec l'API backend.

Captures d'écran et démonstration (Frontend)

L'interface utilisateur a été développée avec Next.js et stylisée avec Tailwind CSS. Elle s'inspire du site u.gg pour offrir une navigation fluide et une organisation claire des informations.

Voici quelques exemples de pages réalisées :

Commenté [JV5]: ajouter capture d'écran (en annexe ?)

- Page d'accueil : permet de rechercher un joueur par pseudo et région.
- Page de profil joueur : affiche les statistiques globales, les champions les plus joués, le KDA, le taux de victoire, etc.
- Liste des champions : présentation en grille des champions du jeu, avec possibilité d'afficher plus d'informations par clic.

Processus de test

Les tests ont été réalisés principalement de manière manuelle durant la phase de développement. Voici notre approche :

- Tests de l'API avec Postman pour s'assurer que chaque route fonctionne comme prévu (test des statuts, des données retournées, des cas d'erreurs).

- Vérification des données dans DBeaver après insertion, modification ou suppression via l'API.

- Tests manuels côté frontend, pour s'assurer que les données s'affichent correctement, que les composants réagissent bien, et qu'il n'y a pas de problème de navigation ou d'affichage.

Nous avons également utilisé console.log dans le backend et le frontend pour repérer plus facilement les erreurs pendant le développement.

Bilan du projet

Difficultés rencontrées

Au cours du développement du projet LPTracker, plusieurs défis techniques et organisationnels sont apparus, qu'il s'agisse de la gestion de la base de données, de l'intégration frontend-backend, ou de la définition des endpoints pertinents.

Difficultés techniques

Problèmes avec la base de données : Lors de la création de la base de données, nous avons rencontré plusieurs difficultés, notamment avec l'outil DBeaver, qui ne répondait pas comme prévu pour certaines opérations. Finalement, nous avons opté pour PGAdmin, qui nous a offert une interface plus stable et adaptée à notre travail sur PostgreSQL. De plus, il a été difficile de construire toute la base de données à partir de rien, surtout en assurant une structure cohérente avec des relations entre les tables. Nous avons dû ajuster et modifier la structure de la base de données à plusieurs reprises au fur et à mesure de l'avancement du projet.

Lier le frontend et le backend : Nous avons eu quelques difficultés pour établir une communication fluide entre Next.js (frontend) et Express.js (backend). La gestion des appels API et la récupération des données depuis le backend vers le frontend ont nécessité des ajustements et des tests pour nous assurer que tout fonctionnait correctement, surtout pour les données dynamiques.

Trouver des endpoints pertinents : Définir des endpoints efficaces et pertinents pour notre API a été un challenge, notamment parce qu'un grand nombre d'endpoints (120) aurait été excessif et difficile à maintenir. Nous avons dû simplifier et nous concentrer sur les fonctionnalités les plus utiles et fréquemment utilisées.

Adaptation à Next.js : L'utilisation de Next.js pour le frontend nous a posé quelques défis, car ce Framework était nouveau pour nous. Nous avons dû apprendre à bien gérer les appels API côté serveur (SSR) et côté client pour avoir une bonne performance et une expérience utilisateur fluide.

Difficultés avec Postman : L'utilisation de Postman pour tester notre API nous a fait perdre du temps au début, car nous n'étions pas familiers avec certaines fonctionnalités avancées. Le paramétrage des requêtes et la gestion des environnements ont nécessité des ajustements, et le processus de test a pris plus de temps que prévu.

Leçons apprises et améliorations futures

(Idées d'évolutions pour aller plus loin.) : à rédiger avec le groupe

Conclusion

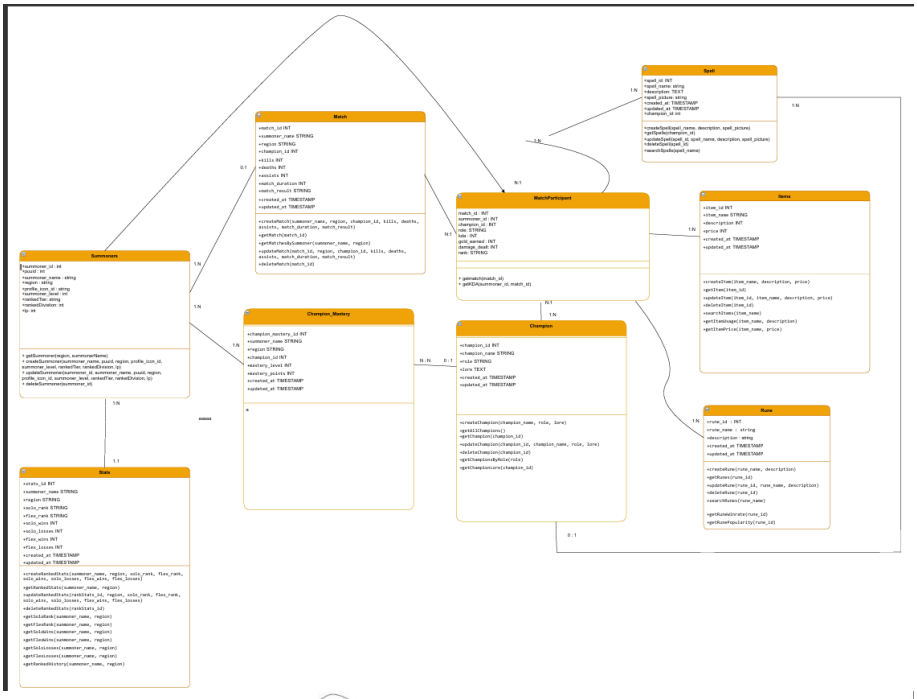
Rappel sur les objectifs atteints et la vision globale : à rédiger avec le groupe

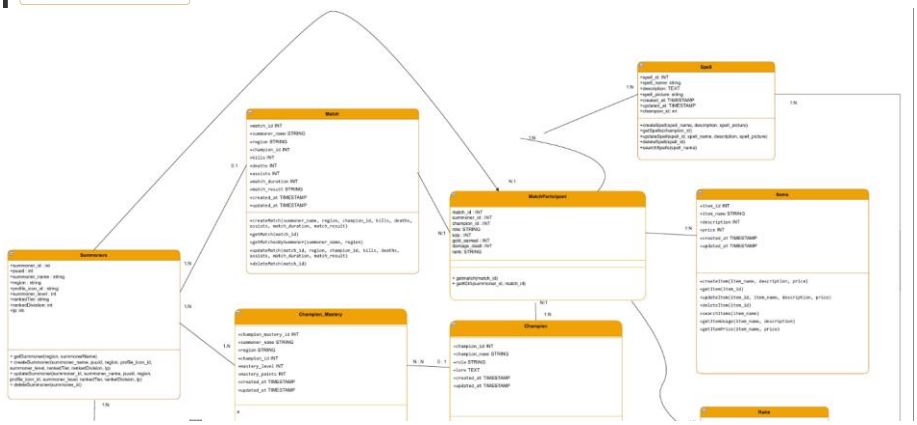
Annexes

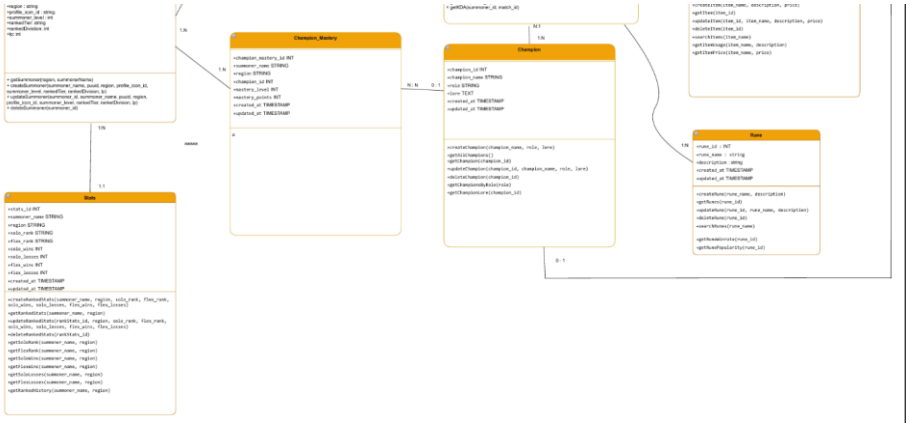
Liens Git : Dépôts et documentation.

Scripts ou consignes d'installation : Pour lancer rapidement l'application.

Autres documents : Diagrammes additionnels, références externes…


Spell
Match
MatchParticipant
Items
Summoners
Champion_Mastery
Champion
Rune
Stats


Spell
Match
MatchParticipant
Items
Summoners
Champion_Mastery
Champion
Rune



1. **Bases de données** :

Nous avons modélisé notre base avec un MCD, optimisé avec des index et des clés étrangères.