# 1. Présentation du projet

## 1.1 Contexte

Ce projet consiste en la conception et le développement d'une application à destination des équipes compétitives du jeu vidéo **Mario Kart**.
L'application a pour objectif l'automatisation de nombreuses tâches jusqu'ici manuelles tout en valorisant les données collectées grâce à leur analyse et à la production de **statistiques avancées**, permettant ainsi d'optimiser les performances et la prise de décision des équipes.

L'application est séparée en deux entités :

- **Front-end** : un **Bot Discord** permettant aux équipes de saisir et consulter les résultats directement depuis leur serveur Discord.
- **Back-end** : une **API REST** connectée à une base de données **PostgreSQL**, chargée de centraliser, traiter et fournir les données.

## 1.2 Fonctionnalités de l'application

- **Gestion des matchs**
  - Calcul en temps réel d'un match.
  - Sauvegarde des résulats des joueurs.
  - Sauvegarde des résultats des courses.
  - Générer les tableaux de résultats.
  - Gérer l'historique des matchs.
- **Gestion des disponibilités des joueurs**
  - Permettre aux joueurs d'ajouter leurs disponbilités (disponible, peut-être, pas disponible ou remplaçant).
  - Afficher les disponibilités des équipes.
- **Gestion du contre-la-montre**
  - Ajouter ou modifier un temps de contre-la-montre.
  - Afficher classement d'une équipe pour une course.
  - Gérer les modes avec ou sans utilisations des objets.
- **Statistiques**
  - Afficher les statistiques des courses pour une équipe
  - Afficher les statistiques générales d'un joueur ou d'une équipe
- **Général**
  - Afficher code-ami Switch d'un joueur
  - Enregister son code-ami

# 2. outillage & organisation du travail

## 2.1 Protocol de déploiement continu - C2.1.1

### Environnement de développement

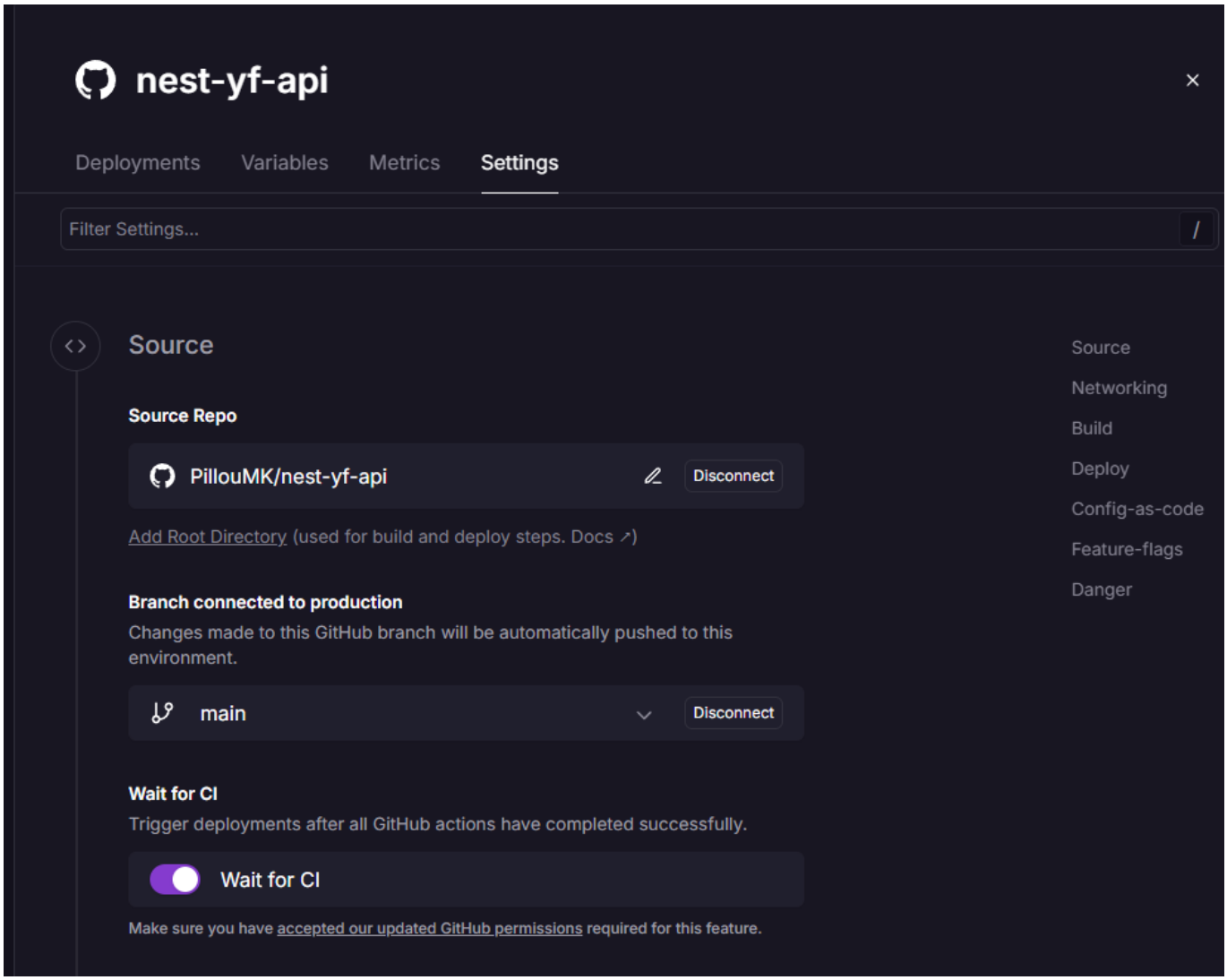
- **Éditeur de code** : Visual Studio Code, avec extensions (Prisma, ESLint, Prettier, Jest).

- **Langage** : TypeScript.
- **Compilateur** : `tsc` (TypeScript Compiler).
- **Serveur d'application** : Node.js (v22.14.0).
- **Gestion des dépendances** : npm.
- **Base de données** : PostgreSQL, pilotée via Prisma ORM.
- **Gestion de sources** : Git (hébergé sur GitHub).
- **Conteneurisation** : Docker, afin de reproduire un environnement homogène (API + base PostgreSQL).

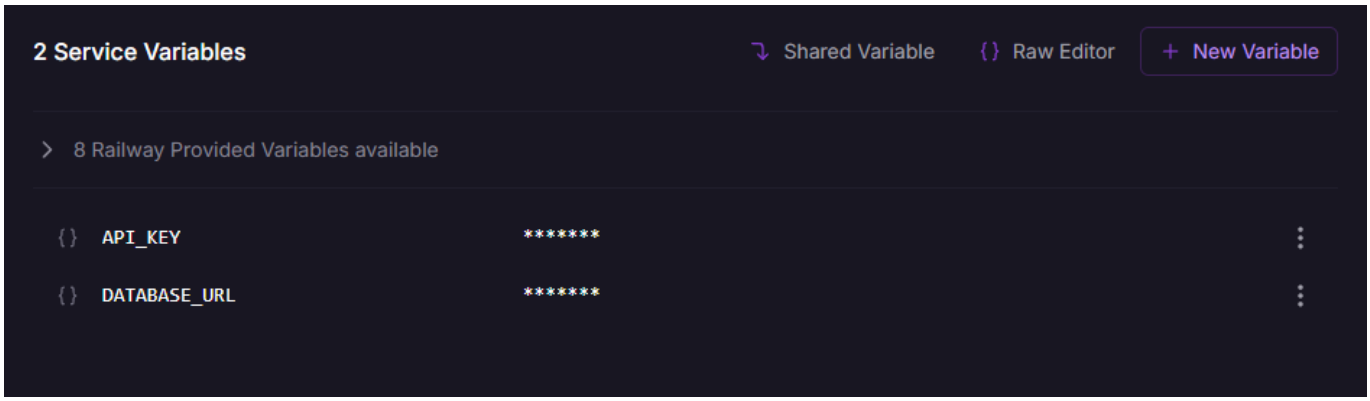
## Déploiement continu

Les applications est déployée et hébergée sur **Railawy** pour les environnements `production` et `staging`, **Railway** va détecter lorsque un push est effectué et va atteindre l'execution de la pipeline de Github Action *(Dont les détails seront fait dans la partie Intégration Continu C2.1.2)*.

Si celle-ci réussi, alors **Railway** va déployer la nouvelle version de manière automatique.




Les variables d'environnements sont également gérées directement sur Railway :




## Critères de qualité et de performance

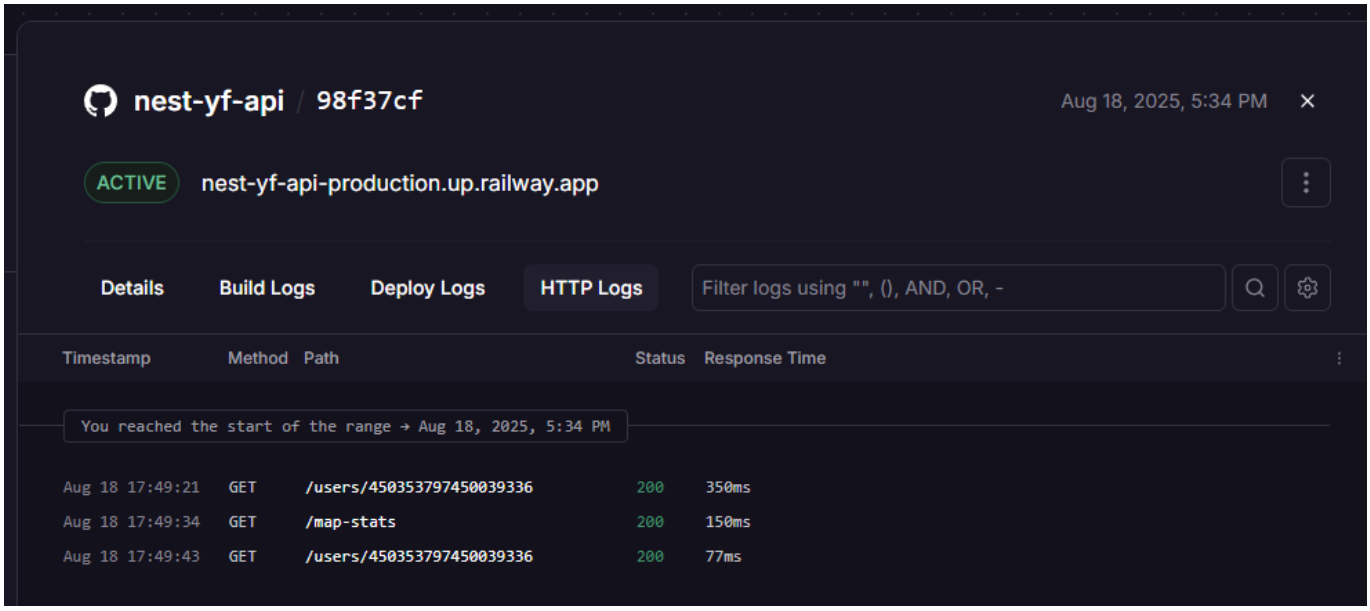
**Qualité du code :**

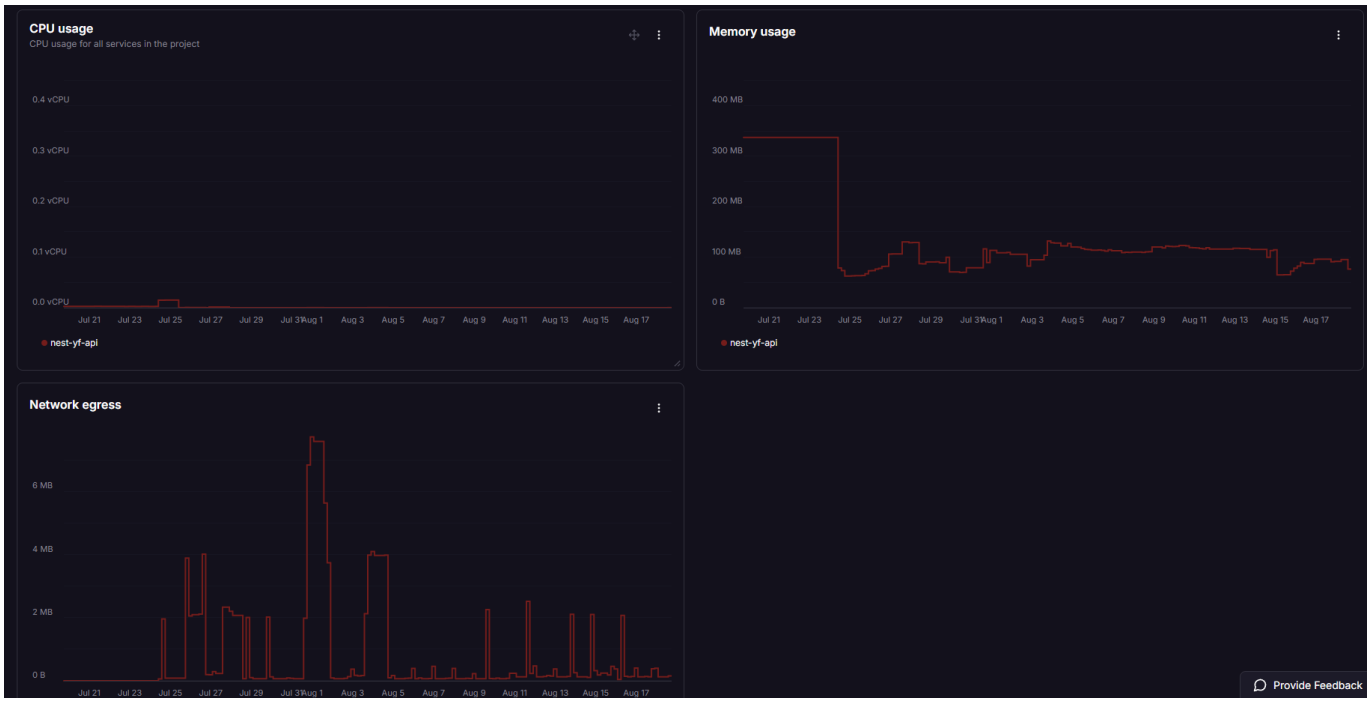
- Vérification **ESLint**.
- Vérification des dépendances inutilisées via **Depcheck**.
- Vérifification des vulnérabilités des dépendances **(audit)**
- Tests unitaires avec **Jest**.

**Performances**

- Majorité des requêtes sous les 200ms de temps de réponse
- Requêtes SQL optimisées et sécurisées avec PrismaORM

Railway met à disposition des outils permettant de suivre les performances de l'applications via des logs, graphiques, metrics, etc :






## 2.2 Protocol d'intégration continue - C2.1.2

Les projets sont hébergés sur GitHub, ce qui permet de centraliser le code source et d'assurer la traçabilité des modifications. Grâce à GitHub, il est possible de mettre en place un système d'intégration continue via GitHub Actions.

### Pipeline CI/CD :

Les deux projets possède une pipeline CI/CD assez similaire permettant d'automatiser les étapes de vie des logiciels, la pipeline est déclenché via Github action lors d'un push (ou d'un merge) sur une branche cible.

Pour prendre l'exemple de **l'API Rest**, il existe trois environnements :

- dev
- staging
- production (branche main)

La pipeline va executer les tâches suivantes :

- Récupérer le code et installer les dépendances
- Executer les tests unitaires **(via Jest)**
- Vérifier les dépendances utilisées **(via Depcheck)**
- Vérifier les vulnérabilités des dépendances **(audit)**
- Analyser le code avec un linter **(ESLint)**
- **Build** l'application
- Générer la **release de version** de manière automatisée avec **semantic-release** (uniquement pour la mise en production)

Voici le workflow de la branche main, `.github/workflows/main.yml` :

```
name: Main CI/CD
```

```yaml
on:
  push:
    branches:
      - main

permissions:
  contents: write
  issues: write
  pull-requests: write
  id-token: write

jobs:
  ci:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: 20

      - name: Install dependencies
        run: npm ci

      - name: Run Jest tests
        run: npm test

      - name: Run depcheck
        run: npm run depcheck

      - name: Run audit
        run: npm audit --audit-level=high

      - name: Run ESLint
        run: npm run lint

      - name: Build
        run: npm run build

      - name: Semantic Release
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN  }}
        run: npx semantic-release
```

# 3. Développement des fonctionnalités

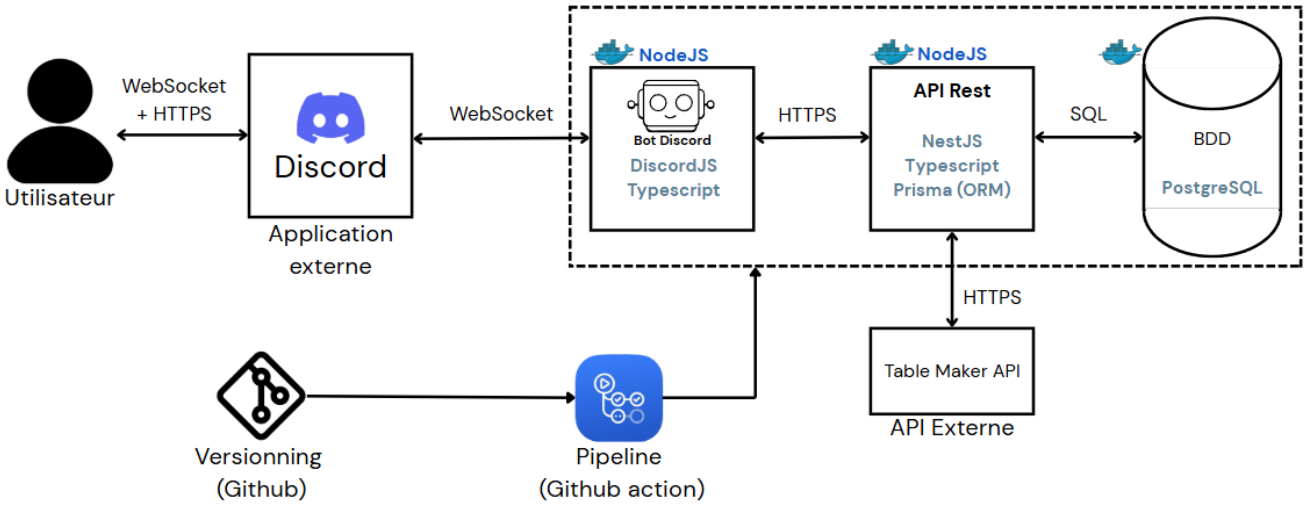
## 3.1 Prototype de l'application - C2.2.1

L'application est découpée en deux projets distincts :

- La partie back-end est une API REST basée sur le framework **NestJS (Node.js / TypeScript)**. La persistance des données est assurée par une base de données **PostgreSQL** qui est connectée à l'API grâce à l'**ORM Prisma.**
- La partie front-end est un Bot Discord, c'est un projet Node.js / TypeScript utilisant la librairie **discord.js**.

Les projets sont conteneurisés avec **Docker** pour garantir la reproductibilité et faciliter le déploiement sur les différents environnements.

### 3.1.1 Schéma de l'architecture logicielle :




**Note :** Table Maker API est une API externe permettant de générer des tableaux de résultats de matchs Mario Kart, elle a été développée par un développeur de la communauté Mario Kart.

### 3.1.2 Paradigmes et frameworks :

**Programmation orientée objet**

Utilisée pour structurer les services et contrôleurs de l'API.

**Exemple d'un controller :**

```
@Controller("teams")
export class TeamsController {
  constructor(private readonly teamsService: TeamsService) {}

  @Post()
  create(@Body() createTeamDto: CreateTeamDto) {
    return this.teamsService.create(createTeamDto);
  }

  @Public()
  @Get()
  findAllTeam() {
    return this.teamsService.findAllTeam();
  }
```

```
  @Public()
  @Get(":id")
  findTeam(@Param("id") id: string) {
    return this.teamsService.findTeam(id);
  }

  @Patch(":id")
  update(@Param("id") id: string, @Body() updateTeamDto: UpdateTeamDto) {
    return this.teamsService.update(id, updateTeamDto);
  }

  @Delete(":id")
  remove(@Param("id") id: string) {
    return this.teamsService.remove(id);
  }
}
```

---

**NestJS**

Utilisé pour l'architecture modulaire de l'API REST.
Chaque module est indépendant, et la modification de l'un n'a pas de conséquences sur les autres.

---

**Prisma ORM**

Prisma assure la connexion à la base de données PostgreSQL, avec une gestion **typée** des données et des migrations.

**Exemple : fichier** `schema.prisma`

```
generator client {
  provider = "prisma-client-js"
  binaryTargets = ["native", "debian-openssl-3.0.x"]
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id        String       @id
  name      String
  flag      String

  roster_id String?      @db.Uuid
  team_id   String?

  roster    Roster?      @relation(fields: [roster_id], references: [id])
  team      Team?        @relation(fields: [team_id], references: [id])
```

```
  timetrials Timetrial[]
  Match_User Match_User[]

  created_at  DateTime   @default(now())
  updated_at  DateTime   @updatedAt

}
[...]
```

---

**Docker**

Utilisé pour la portabilité des projets.

**Exemple : `docker-compose.yml` pour lancer l'API avec la base de données (développement)**

```
version: "3.9"

services:
  db:
    image: postgres:15
    container_name: postgres_db
    restart: always
    ports:
      - "5432:5432"
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: nest_db
    volumes:
      - db_data:/var/lib/postgresql/data

  api:
    build: .
    container_name: nest_api
    ports:
      - "3000:3000"
    depends_on:
      - db
    env_file:
      - .env
    environment:
      DATABASE_URL: ${DATABASE_URL}
    volumes:
      - .:/app
      - /app/node_modules
    command: >
      sh -c " npx prisma generate &&
              npx prisma migrate deploy &&
              npx prisma db seed &&
              npm run start:dev"
```

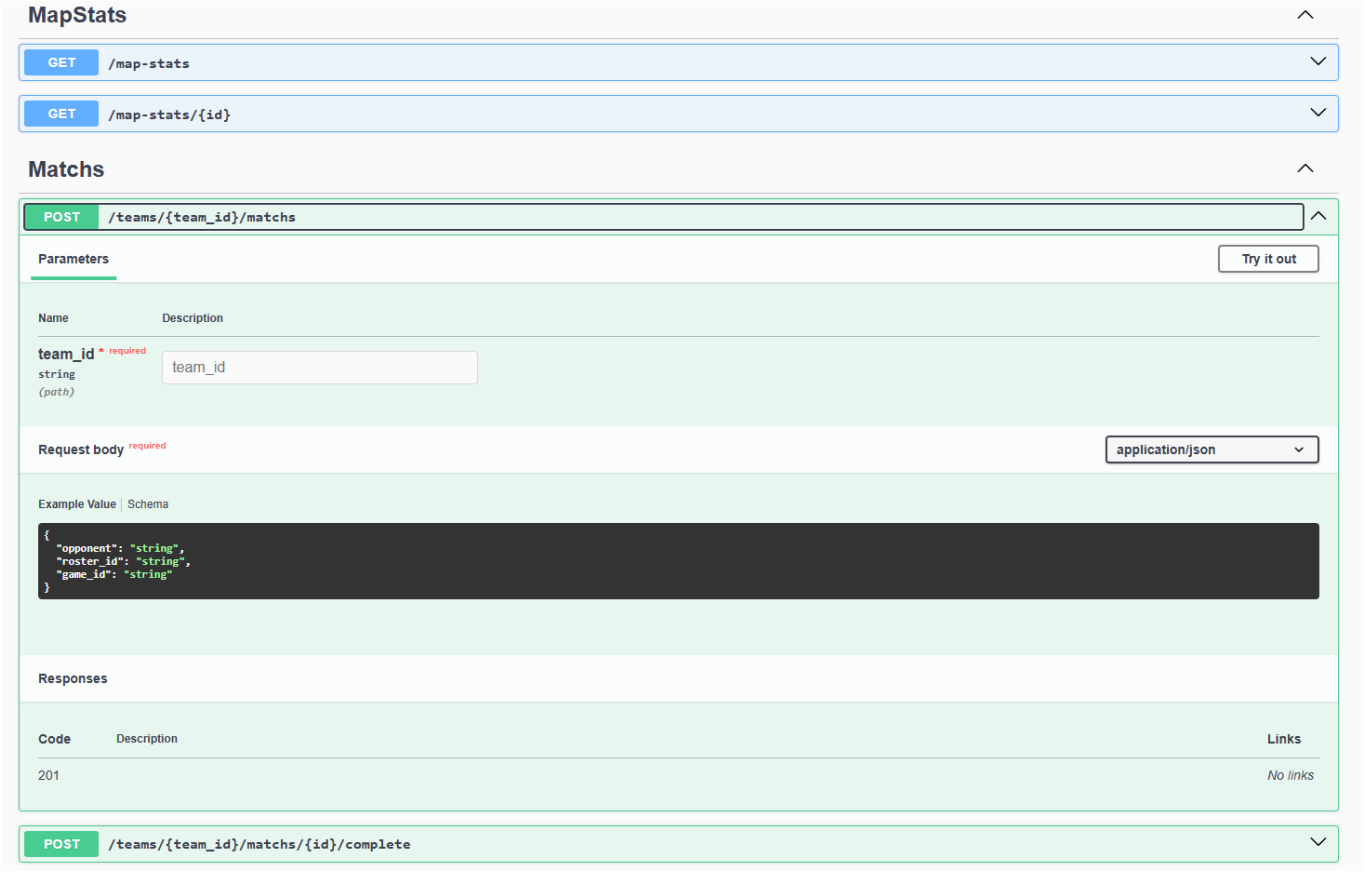
```
volumes:
  db_data:
```

---

**Swagger**

Swagger est utilisé pour la génération automatique de la documentation de l'API.

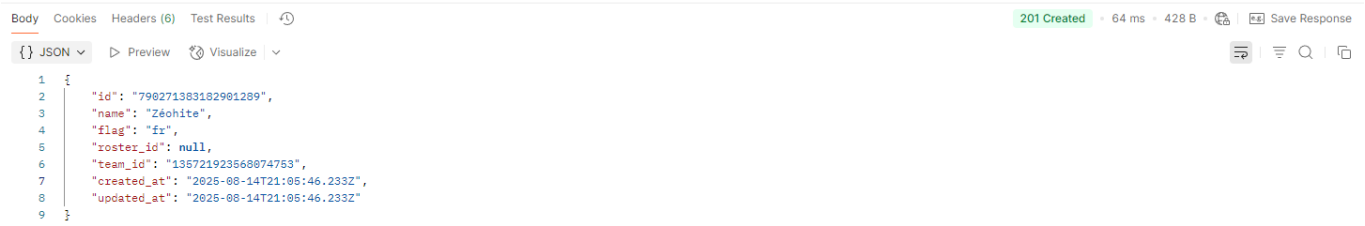
**Exemple du rendu :**




## 3.1.3 Prototype réalisé :

**API Rest**

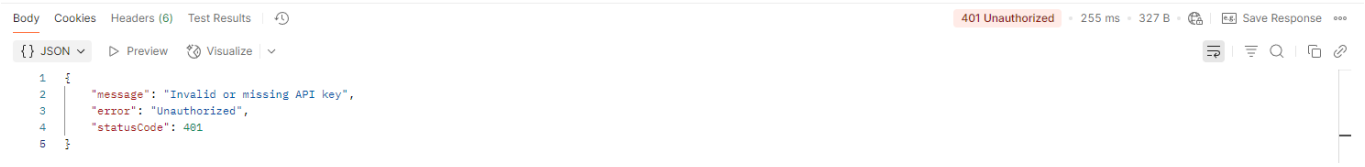
Utilisation de l'API via Postman, collection générée via le fichier JSON généré par Swagger automatiquement.
**Exemple pour l'endpoint** `users/:id` :


Lien de la documentation : Documentation API

L'API sécurise les endpoints via une clé API bloquant les requêtes ne possédant pas la clé : La clé est un UUID v4 (voir RFC 9562).

**Exemple :**

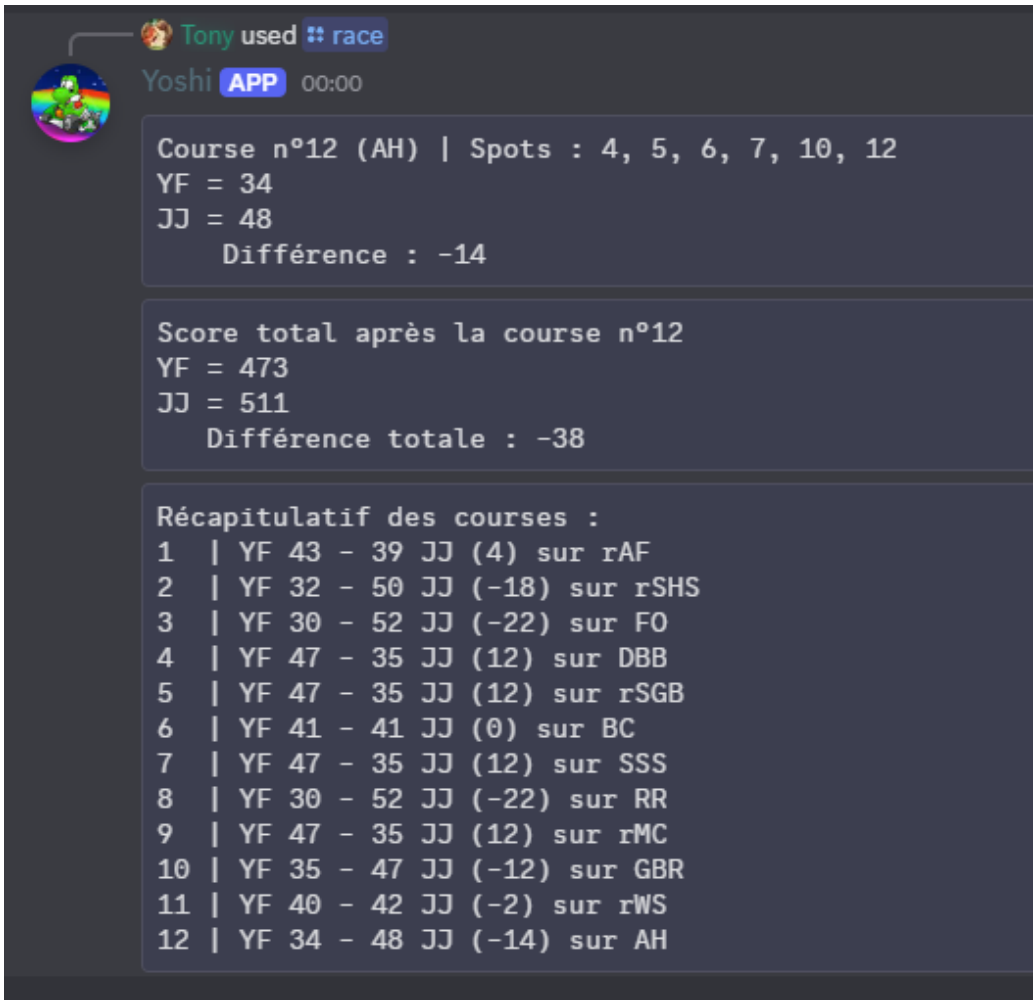



---

**Bot Discord**

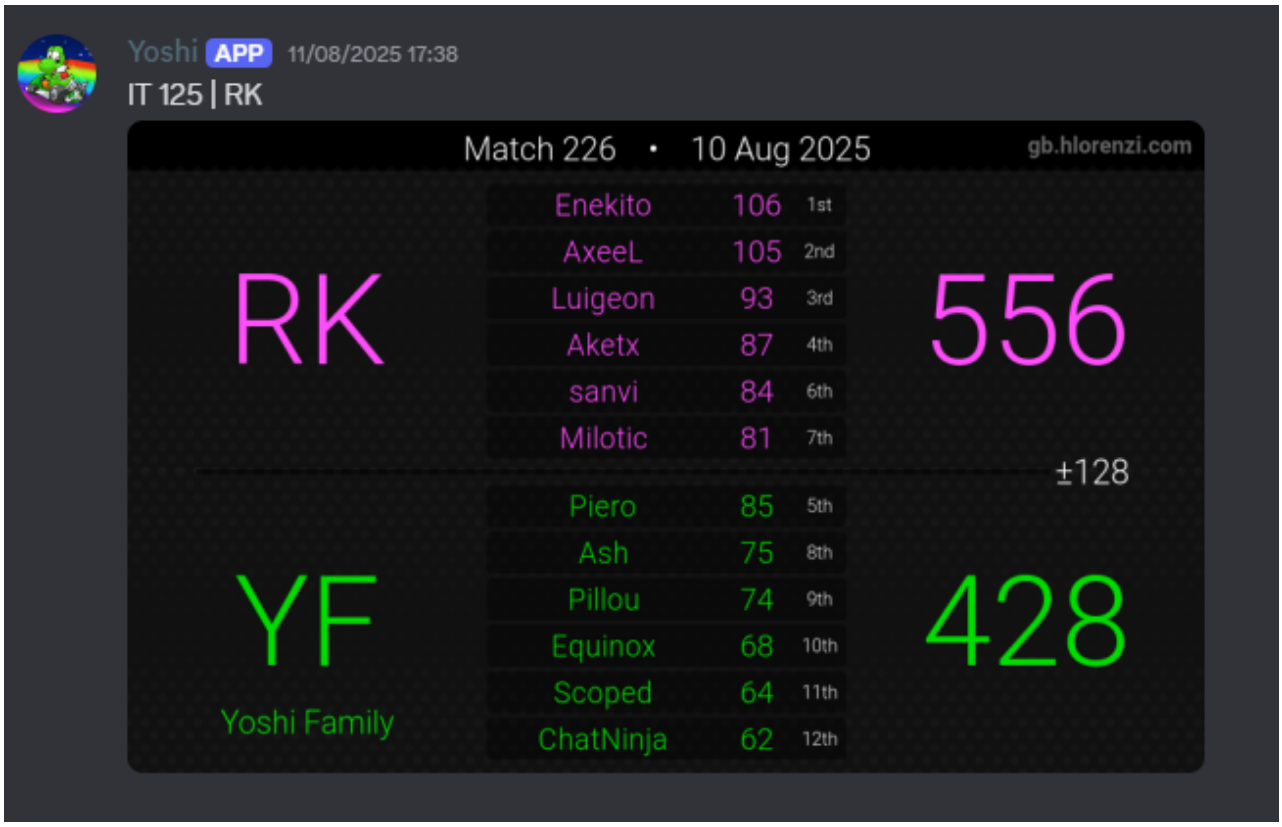
Exemple de la gestion d'un match :

**Suivi du match :**

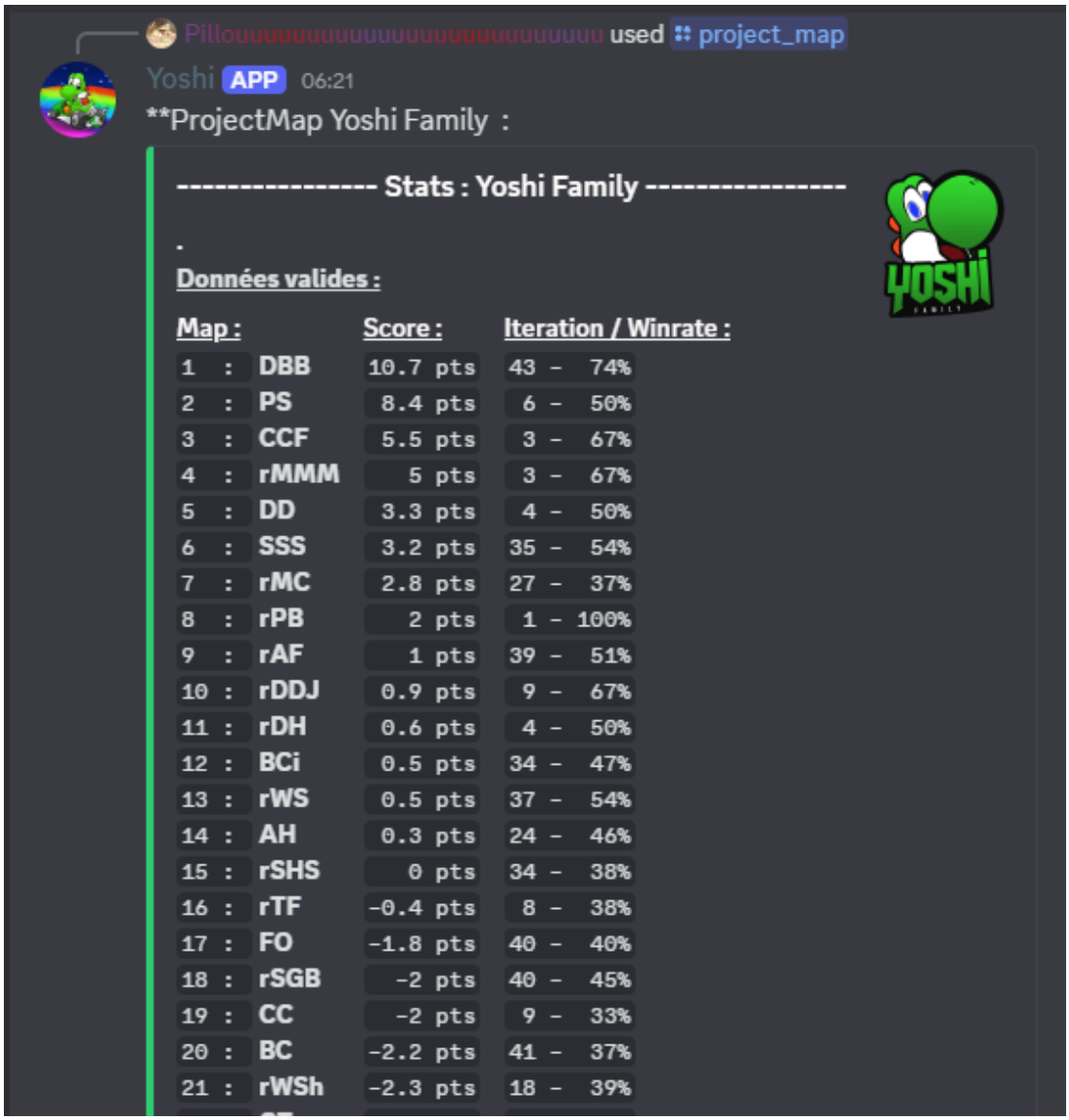



**Génération du résultat :**

**Post automatique du résultat (dans un channel précis) :**

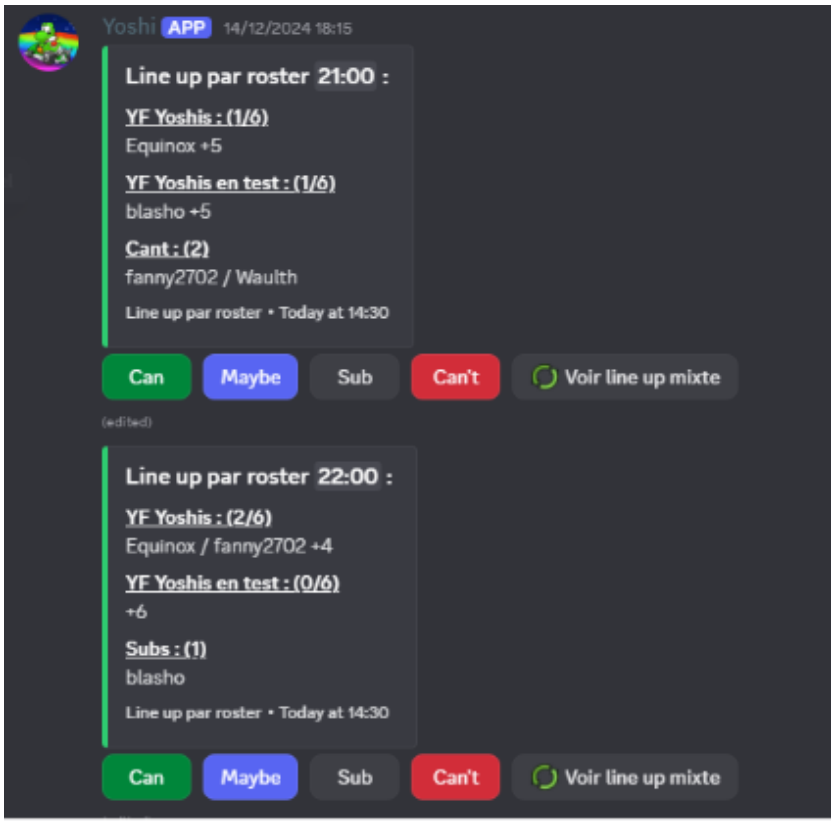


**Exemple des statistiques des matchs :**



**Exemple Contre-la-montre :**


**Exemple gestion des disponibilités par horaire :**




## 3.2 Développement des tests unitaires - C.2.2.2

Des tests unitaires ont été mis en place afin de prévenir les régressions et de s'assurer du fonctionnement correct des diverses fonctionnalités. Pour l'API Rest, les tests unitaires sont mis en place à l'aide de Jest, directement intégré avec NestJS.

Chaque service et controller est testé, un mock de Prisma est créé pour simuler une interaction avec la base de données.

**Exemple des tests unitaires pour le service des matchs :**

```
describe("MatchsService", () => {
  let service: MatchsService;

  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      providers: [
        MatchsService,
        { provide: PrismaService, useValue: prismaMock },
      ],
    }).compile();

    service = module.get<MatchsService>(MatchsService);
  });

  describe("createMatch", () => {
    it("should create a match with default values", async () => {
      const mockMatch = { id: 1 } as const;
      prismaMock.match.create.mockResolvedValueOnce(mockMatch);

      const dto = { opponent: "TeamX", game_id: "game1", roster_id: "roster1" };
      const result = await service.createMatch("team1", dto);

      expect(result).toEqual(mockMatch);
    });
  });
});
```

On utilise la librairie **Jest** pour les tests unitaires Cet exemple teste la création d'un match, on initialise un mock de Prisma (variable `prismaMock`) pour simuler une interaction avec la base de données.

Tous les controlleurs et services sont testés, chaque méthode des services possèdent au moins un test pour couvrir tous les cas possibles.

---

## 3.3 Normes de sécurité - C2.2.3

### 3.3.1 OWASP

Voici les mesures qui ont été mises en place pour répondre aux 10 failles de sécurité principales décrites par l'OWASP (Top 10 datant de 2021) :

| Failles OWASP | Description | Mesures / Solutions |
|---|---|---|
| A01 – Broken Access Control | Contrôles d'accès manquants ou mal configurés | **API** : L'API est sécurisée via une clé UUID v4.<br>**Bot Discord** : Les commandes Discord sensibles sont limitées d'accès via les rôles Discord (ex : administrateur). |

| Failles OWASP | Description | Mesures / Solutions |
|---|---|---|
| A02 – Cryptographic Failures | Failles liées au chiffrement des données sensibles | Stockage sécurisé des clés d'API et mots de passe via variables d'environnement.<br>L'API ne stocke aucune donnée sensible (pas de données confidentielles, personnelles, etc.). |
| A03 – Injection | Injection de code ou commandes dans les requêtes | **API** : Utilisation de l'ORM Prisma pour valider le format des données en entrée et éviter des injections SQL.<br>**Bot Discord** : Contrôle côté client des données fournies dans les commandes pour uniquement correspondre au format attendu. |
| A04 – Insecure Design | Conception globale non sécurisée | **API** : Isolation des ressources par modules.<br>**Bot Discord** : Contrôle de plausibilité sur les données fournies. |
| A05 – Security Misconfiguration | Mauvaise configuration des services ou API | Utilisation de Docker pour un déploiement automatisé et sans erreurs.<br>Ne pas laisser de fonctionnalités non nécessaires (ex : commandes qui ne sont plus utilisées pour le Bot, endpoint pour l'API). |
| A06 – Vulnerable and Outdated Components | Composants obsolètes ou vulnérables | Supprimer les dépendances non utilisées (ex : librairies, fichiers non utilisés, etc.) trouvé via Depcheck.<br>Vérifier les vulnérabilités des dépendances avec un audit.<br>**Bot Discord** : Mettre à jour la librairie DiscordJS (éviter une dette technique). |
| A07 – Identification and Authentication Failures | Mauvaise gestion des identités | **Bot Discord** : Pas d'authentification via le bot nécessaire pour l'utilisateur final (Discord gère l'authentification).<br>**API** : Clé API sécurisée (utilisation de UUID v4), nombre de requêtes limité sur un endpoint (erreur 429). |
| A08 – Software and Data Integrity Failures | Données ou logiciels corrompus ou non vérifiés | Utilisation de OWASP Dependency Check dans la pipeline de déploiement.<br>Audit des dépendances<br>**Bot Discord** : Les données ne proviennent que de l'API que nous avons développée.<br>**API** : Vérification de l'intégrité des données fournies par le bot (utilisation d'un DTO). Vérifier l'intégrité des données de l'API externe utilisée pour la génération de tableau de résultats. |
| A09 – Security Logging and Monitoring Failures | Absence de journalisation et surveillance | Mise en place de logs à chaque action.<br>Mettre en place une alerte en cas d'utilisation suspicieuse.<br>**API** : Logs stockés et visibles sur Railway.<br>**Bot Discord** : Logs stockés et visibles sur PebbleHost. Logs des commandes également postés sur un channel Discord privé. |
| A10 – Server-Side Request Forgery (SSRF) | Requêtes HTTP non sécurisées côté serveur | Limitation des appels externes à des URLs fiables.<br>Ne pas renvoyer de données sensibles au client lors d'une requête échouée (ex : format d'un ID, etc.). |

Bien que notre application ne gère pas de données personnelles critiques, il est important de mettre en place de bonnes pratiques de sécurités, afin que ceci deviennent une norme et non pas une exception

### 3.3.2 Accessibilité de l'application

Pour ce projet, nous sommes face à un cas un peu spécial concernant les normes d'accessibilité, nous n'avons pas d'interface web qui nous est propre sur laquelle nous pourrions développer les fonctionnalités en respectant les normes d'accessibilité.
Notre interface est le Bot Discord qui est utilisé sur **l'application Discord**, application sur laquelle nous n'avons pas la main pour tout ce qui est design ou structure. En revanche, nous pouvons **agir** sur la façon dont les commandes, les textes et les interactions sont conçus, afin de les rendre plus **clairs**, **simples** à utiliser et **accessibles** au plus grand nombre.

Pour ces raisons, nous avons choisi de nous baser sur le référentiel **OPQUAST**, bien que celui-ci soit à l'origine pensé pour les sites et applications web, il reste tout à fait applicable pour mesurer la qualité et l'accessibilité d'une interface, y compris dans un contexte un peu particulier comme celui d'un bot Discord.
Discord respecte les normes d'accessibilité pour son application, et a mis en place beaucoup de possibilités pour les Bots Discord afin que ceux-ci respectent (dans la mesure du possible) un maximum de ces normes. Les efforts pour ces normes d'accessibilité se sont donc concentrés sur l'interaction avec le Bot Discord ainsi que ses réponses. Voici un tableau présentant les bonnes pratiques Opquast mises en œuvre dans le contexte d'un Bot Discord :

| **Règle Opquast** | **Mise en place avec le Bot Discord** |
|---|---|
| L'application est utilisable et navigable en utilisant uniquement le clavier. (règle 161) | Discord est entièrement utilisable avec uniquement le clavier. La norme pour un Bot Discord est désormais d'utiliser les slash commandes, qui permettent d'interagir avec le bot, elles sont compatibles avec le clavier à 100%. |
| L'application offre un ou plusieurs mécanismes pour s'adapter à une vue mobile (règle 189) | Le contenu généré par le bot peut prendre différentes formes : texte, images, embeds, etc. Pour les contenus qui seraient trop larges et donc déformés sur un téléphone, nous avons mis en place un bouton "Vue mobile" pour modifier le contenu. Il n'est malheureusement pas possible de détecter de manière fiable si l'utilisateur est sur mobile ou non, ainsi nous avons estimé que donner la possibilité à l'utilisateur de modifier directement le contenu via ce bouton est la meilleure solution disponible pour ce genre de cas. |
| Chaque champ indique le format de données attendu (règle 70) | Chaque champ de formulaire indique s'il est requis ou non (règle 69). Les options dans les slash commandes d'un bot agissent comme des champs de formulaire. Il est possible d'indiquer le format attendu ainsi que s'il est requis ou non. Dans notre cas, nous précisons pour chaque option si elle est requise ou non ainsi que le format demandé directement dans le code de la commande. |

| Règle Opquast | Mise en place avec le Bot Discord |
|---|---|
| Chaque image décorative possède un texte alt (règle 111) | Chaque lien d'image possède un texte alt (règle 112). Depuis 2021, Discord permet d'ajouter à une image postée sur Discord une description (qui fait office de alt texte). Il est possible de renseigner cette description avec le Bot Discord lorsque celui-ci utilise une image. Dans notre cas, le bot Discord attache toujours une description pertinente aux images qu'il utilise. **Cela est pertinent pour les lecteurs d'écrans** |

**Note :** Une évolution possible de l'application serait une traduction automatique des commandes qui serait adaptée à la langue de l'utilisateur, il est en effet possible de connaître la langue dans laquelle l'utilisateur utilise Discord. Cela permettrait de respecter la règle 80 de Opquast. Cela nécessiterait d'investir plus de moyen dans la traduction de l'application dans diverse langue.

Les règles Opquast ont aussi une pertinence au niveau de l'API Rest dont voici quelques exemples :

| Règle Opquast | Mise en place au niveau de l'API |
|---|---|
| Toutes les pages utilisent le protocole HTTPS (règle 192) | L'API est hébergée sur Railway, celui-ci force l'utilisation du protocole HTTPS pour toutes les URL et endpoints de l'API. En cas de requête HTTP, le serveur envoie une redirection HTTP 301 ou 302 pour rediriger vers le protocole sécurisé. |
| Le serveur renvoie une erreur 404 lorsque qu'une ressource n'est pas trouvée (règle 215) | Si une ressource n'est pas trouvée (ou qu'un endpoint n'existe pas), le serveur renverra une erreur 404 avec un message pertinent (voir cf.404). |

## 3.4 Historique des différentes versions - C2.2.4

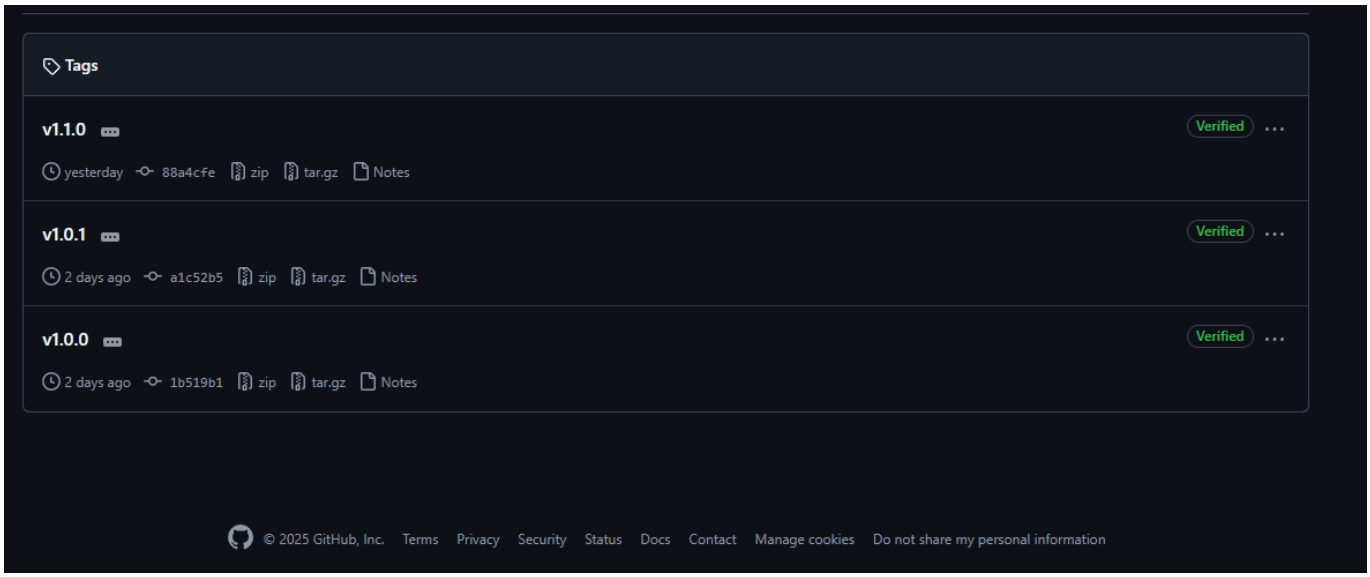
Pour les deux projets, un système de gestion de version a été mis en place afin de suivre l'évolution du code et de maintenir une traçabilité complète des modifications. Nous utilisons **Semantic-release**, un outil qui permet de générer automatiquement un historique des versions et de gérer les montées de version de manière automatisée, en se basant sur les commits qui respectent une nomenclature précise.

Le fonctionnement est intégré directement dans notre **CI/CD** via un workflow sur GitHub. L'historique des versions est mis à jour automatiquement lors des merges dans la branche `main` (équivalente à l'environnement de production). Ainsi, chaque version publiée est cohérente avec les changements apportés et documentée de manière structurée.

### 3.4.1 Historique de version actuelle

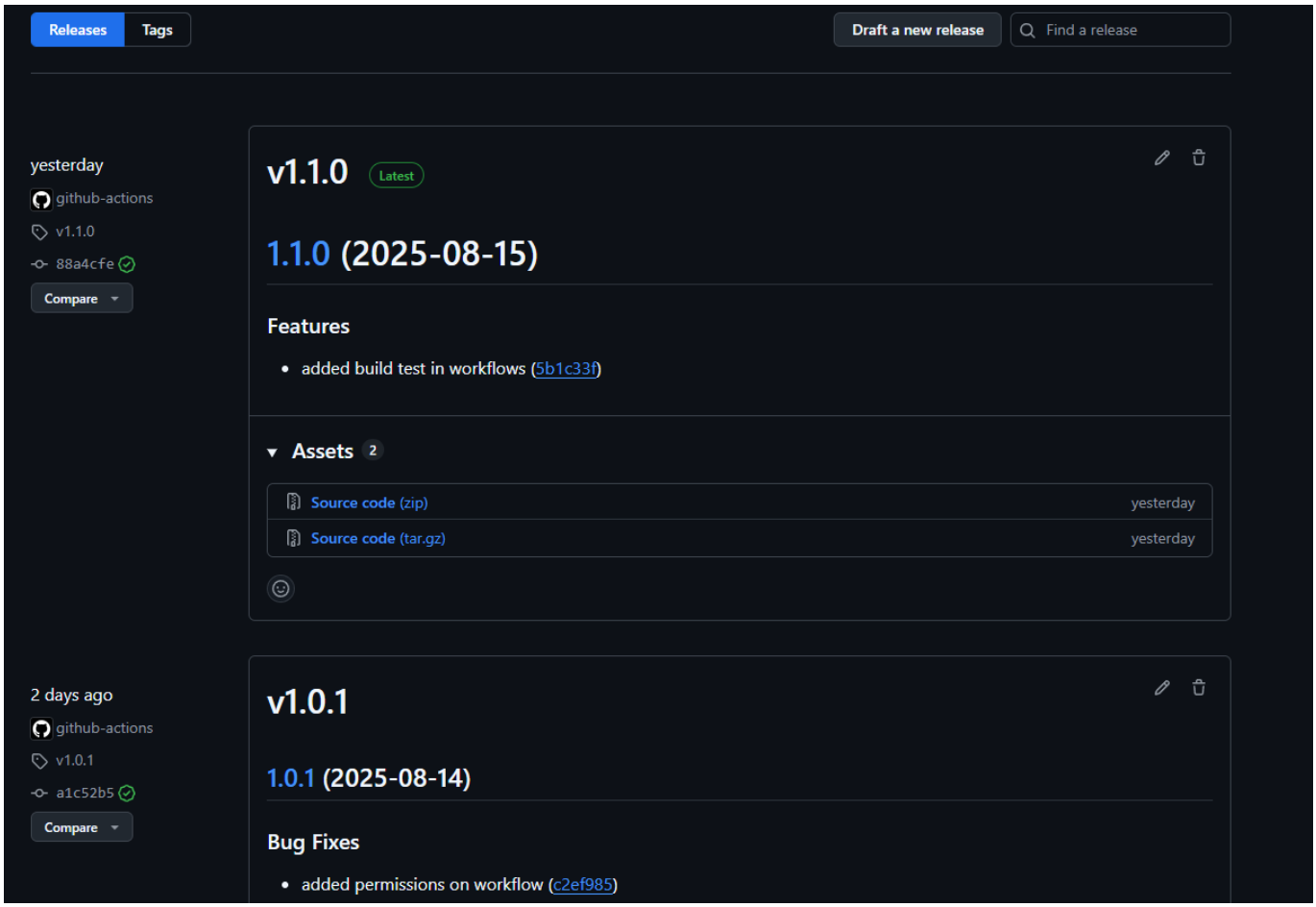
Semantic-release crée automatiquement des **tags Git** pour chaque version, ce qui permet de savoir précisément quelle version du projet est déployée. Il est également possible d'inclure des notes de version détaillant les modifications, corrections ou nouvelles fonctionnalités apportées par chaque release, facilitant ainsi la lecture de l'historique pour les développeurs et les utilisateurs.

Voici l'historique de version pour l'API :




### 3.4.2 LTS de l'application

Cette gestion des versions permet de mettre en production la dernières version la plus stable À ce jour, pour l'API, la version stable utilisée en production est la **1.1.0**.




# 4. Recette de fonctionnalité

## 4.1 Cahier de recette - C2.3.1

Cahier de recette de l'API

| ID | Scénario | Pré-condition | Action | Résultat attendu |
|---|---|---|---|---|
| U1 | Création d'un utilisateur | Aucun utilisateur requis | POST `/users` avec payload valide | Utilisateur créé et renvoi des informations de l'utilisateur (id, nom, email, etc.) |
| U2 | Création multiple d'utilisateurs | Aucun utilisateur requis | POST `/users/`bulk avec liste d'utilisateurs | Tous les utilisateurs sont créés, renvoi d'une liste avec leurs ids |
| U3 | Récupération d'un utilisateur | Utilisateur existant | GET `/users/{id}` | Renvoi des informations de l'utilisateur correspondant à l'id |
| U4 | Mise à jour d'un utilisateur | Utilisateur existant | PATCH `/users/{id}` avec modifications | Informations de l'utilisateur mises à jour correctement |
| M1 | Récupération des circuit d'un jeu | Circuits existant | GET `/maps/{id}` | Renvoi la liste des circuits d'un jeu |
| T1 | Création d'une équipe | Aucun prérequis | POST `/teams` avec payload valide | Equipe créée et renvoi des informations de l'équipe |
| T2 | Récupération de toutes les équipes | Equipes existantes | GET `/teams` | Liste complète des équipes |
| T3 | Récupération d'une équipe | Equipe existante | GET `/teams/{id}` | Renvoi des informations de l'équipe correspondante |
| T4 | Mise à jour d'une équipe | Equipe existante | PATCH `/teams/{id}` avec modifications | Informations de l'équipe mises à jour correctement |
| T5 | Suppression d'une équipe | Equipe existante | DELETE `/teams/{id}` | Equipe supprimée, réponse 200 |

| ID | Scénario | Pré-condition | Action | Résultat attendu |
|---|---|---|---|---|
| R1 | Création d'un roster | Aucun prérequis | POST `/rosters` avec payload valide | Roster créé et renvoi des informations |
| R2 | Récupération de tous les rosters | Rosters existants | GET `/rosters` | Liste de tous les rosters |
| R3 | Mise à jour d'un roster | Roster existant | PATCH `/rosters/{id}` avec modifications | Roster mis à jour correctement |
| R4 | Suppression d'un roster | Roster existant | DELETE `/rosters/{id}` | Roster supprimé, réponse 200 |
| TT1 | Création d'un timetrial | Aucun prérequis | POST `/timetrials` avec payload valide | Timetrial créé, renvoi des informations |
| TT2 | Récupération d'un timetrial | Timetrial existant | GET `/timetrials/{map_tag}` | Renvoi des informations du timetrial correspondant |
| G1 | Récupération des jeux | Jeux existants | GET `/games` | Liste complète des jeux |
| MS1 | Récupération des statistiques des circuits | MapStats existants | GET `/map-stats` | Renvoi de toutes les stats des circuits pour une équipe |
| MS2 | Récupération des statistique d'un circuit | MapStats existant | GET `/map-stats/{id}` | Renvoi des stats correspondantes d'un circuit pour une équipe |
| MA1 | Création d'un match | Equipe et payload valide | POST `/teams/{team_id}/matchs` | Match créé avec état initial |
| MA2.1 | Complétion d'un match | Match en cours | POST `/teams/{team_id}/matchs/{id}/complete` | Match mis à jour et marqué comme terminé, résultat des circuits (map-stats) enregistrés |
| MA2.2 | Refuser complétion d'un match | Match n'existe pas | POST `/teams/{team_id}/matchs/{id}/complete` | Erreur 404 |

| ID | Scénario | Pré-condition | Action | Résultat attendu |
|---|---|---|---|---|
| MA2.3 | Refuser complétion d'un match | paramètres des scores absents | POST `/teams/{team_id}/matchs/{id}/complete` | Erreur 400 |
| MA3.1 | Prévisualisation du résultat d'un match | Match existant | POST `/teams/{team_id}/matchs/{id}/preview` avec payload | Renvoi du tableau de prévisualisation du match qui devra être validé |
| MA3.2 | Refuser prévisualisation du résultat d'un match | Match n'existe pas | POST `/teams/{team_id}/matchs/{id}/preview` avec payload | Erreur 404 |
| MA3.3 | Refuser prévisualisation du résultat d'un match | Résultat ne correspondent pas | POST `/teams/{team_id}/matchs/{id}/preview` avec payload | Erreur 400 |
| MA4 | Publication d'un match | Match terminé | POST `/teams/{team_id}/matchs/{id}/publish` | Match publié avec validation du tableau de résultat, enregistrement des scores des joueurs (match_user) |
| MA5 | Liste des matchs terminés | Matchs terminés existants | GET `/teams/{team_id}/matchs/done` | Liste des matchs terminés |
| MA6 | Liste des matchs publiés | Matchs publiés existants | GET `/teams/{team_id}/matchs/published` | Liste des matchs publiés |
| MA7 | Récupération d'un match | Match existant | GET `/teams/{team_id}/matchs/{id}` | Renvoi des informations complètes du match |

## Cahier de recette du Bot Discord

| ID | Scénario | Pré-condition | Action | Résultat attendu |
|---|---|---|---|---|

| ID | Scénario | Pré-condition | Action | Résultat attendu |
|---|---|---|---|---|
| M1 | Créer un match | X | Commande `/startwar` | Le bot fait une requête à l'API et confirme la création du match |
| M2.1 | Ajouter résultat d'une course | Match en cours requis | Commande `/race` avec spots et circuit | Met à jour le score du match en cours |
| M2.2 | Refuser résultat d'une course | Match en cours requis, résultats incohérents | Commande `/race` | Refuse de mettre à jour le match, Bot Discord prévient de l'erreur |
| M2.3 | Refuser résultat d'une course | Match en cours requis, circuit n'existe pas | Commande `/race` | Refuse de mettre à jour le match, Bot Discord prévient de l'erreur |
| M3 | Modifier résultat d'une course | Match en cours requis | Commande `/edit_race` avec spots + circuit (et numéro de la course à modifier) | Modifie le résultat d'une course, met à jour les scores |
| M4 | Ajouter une pénalité | Match en cours requis | Commande `/pena` | Ajoute une pénalité à l'équipe indiqué, met à jour les scores |
| M5 | Finir un match | Match en cours requis | Commande `/stopwar` | Met fin au match avec requête à l'API, Bot Discord confirme l'arrêt du match |
| M6.1 | Preview un match | Match terminé | Commande `/preview_match` | Affiche un tableau de résultat qui doit être validé |
| M6.2 | Echec de la preview un match | Match terminé, résultats ne correspondent pas | Commande `/preview_match` | Indique à l'utilisateurs que les résultats ne concordent pas |
| M7.1 | Publier un match | Match terminé, preview_match a réussi | Bouton `Valider` sous le message à l'issue du `preview_match` | Valide la publication du match, poste le résultat du match dans le salon de discussion configuré |
| M7.2 | Annuler publication d'un match | Match terminé, preview_match a réussi | Bouton `Annuler` sous le message à l'issue du `preview_match` | Annule la commande en cours et ne publie pas le match |

| ID | Scénario | Pré-condition | Action | Résultat attendu |
|---|---|---|---|---|
| M7.3 | Echec publication d'un match | Match terminé, preview_match a réussi, salon de discussion de résultat non-configuré | Bouton `Valider` sous le message à l'issue du `preview_match` | Bot Discord indique l'échec de la commande et recommande de configurer le salon de résultat |
| M7.4 | Echec publication d'un match | Match terminé, preview_match a réussi, Bot n'a pas l'autorisation de psoter dans le salon de discussion indiqué | Bouton `Valider` sous le message à l'issue du `preview_match` | Bot Discord indique qu'il n'a pas les permissions requises |
| T1.1 | Ajouter/modifier un temps contre la montre | Users existant | Commande `/set_tt` | Le Bot fait une requête à l'API et confirme l'ajout/modification du temps renseigné |
| T1.2 | Refuser l'ajout/modification un temps contre la montre | Users existant, temps renseigné non-valide | Commande `/set_tt` | Le Bot prévient que le temps n'a pas le format attendu |
| T1.3 | Refuser l'ajout/modification un temps contre la montre | Users existant, circuit n'existe pas | Commande `/set_tt` | Le Bot prévient que le circuit renseigné n'est pas valide |
| T2.1 | Afficher classement d'un circuit | Circuit existant | Commande `/classement` | Le Bot affiche le classement des joueurs de l'équipe (si aucun temps, classement vide) |
| T2.2 | Refuser affichage du classement | Circuit n'existe pas | Commande `/classement` | Le Bot prévient que le circuit renseigné n'est pas valide |
| MS1.1 | Afficher statistique des circuits | X | Commande `/projectmap` | Le Bot affiche les statistiques des circuits sous forme de classement |
| MS1.2 | Echec de l'affichage des statistique | Aucune données | Commande `/projectmap` | Le Bot indique l'absence de données |
| LU1.1 | Affichage des disponibilité des joueurs | X | commande `lineup` | Affiche les disponibilités pour un horaire donné |

| ID | Scénario | Pré-condition | Action | Résultat attendu |
|---|---|---|---|---|
| LU1.2 | Echec de l'affichage des disponibilité des joueurs | heure voulu non conforme | commande `lineup` | Le Bot Discord indique que l'horaire renseigné n'est pas valide |
| LU2 | Mise à jour de la disponibiltié d'un joueur | Message de disponibilité exitant | Interaction avec les boutons du message de disponibilité | Met à jour la disponibilité du joueurs (disponible, non-disponible, peut-être disponible) |
| MOB1 | Affichage mobile | Message ayant un bouton "Vue mobile" | Interaction avec le bouton | Modifie le message pour avoir un affichage mobile-friendly |

### Tests de Sécurités

L'API est protégé des attaques par injection SQL grâce à Prisma car on utilise uniquement des requêtes paramétrées ($1, $2, $3...) côté base de données, ce qui empêche un input utilisateur de modifier la structure SQL.
Cependant par mesure de sécurité, nous pouvons tout de même faire des tests de sécurités.
Egalement, tous les endpoints critiques sont protégés par une clé d'API, afin de ne pas avoir de modification extérieur par des personnes extérieurs.

| ID | Scénario | Action | Résultat attendu |
|---|---|---|---|
| S1 | Requête sans fournir la clé d'API | Requête **POST/PATCH/DELETE** sur n'importe quelle ressource, sans le paramètre `x-api-key` | Erreurs 401 |
| S2 | Tentative d'injection SQL | GET `users/' OR 1=1 --` | Erreurs 404 |
| S3 | Tentative d'injection SQL | GET `users/'; DROP TABLE User; --` | Erreurs 404 |

## 4.2 Plan de correction des bogues - C2.3.2

### Procédure :

- Signalement du bug :
    - Si bug découvert durant le développement, je créé une issue sur le repo github
    - Si découvert par un utilisateur/testeur, signalement dans le salon de discussion Discord prévu à cet effet avec un template pré-établi. Puis je créé l'issue github.

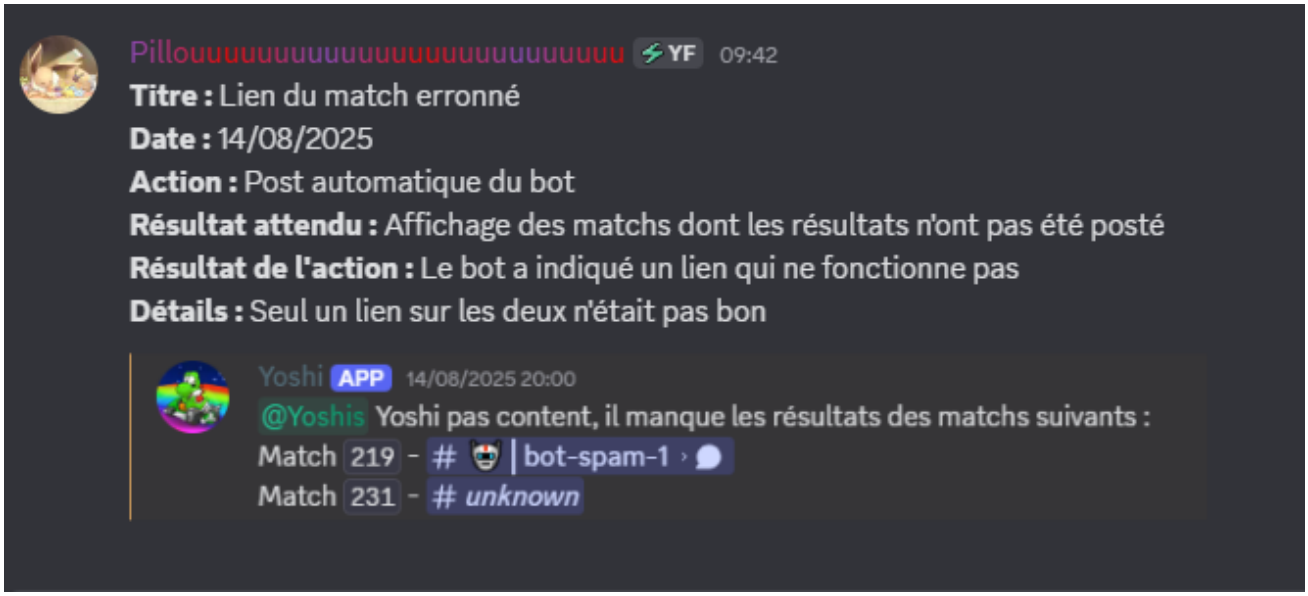
  Template :

```
Titre :
Date :
Action :
Résultat attendu :
Résultat de l'action :
Détails :
```

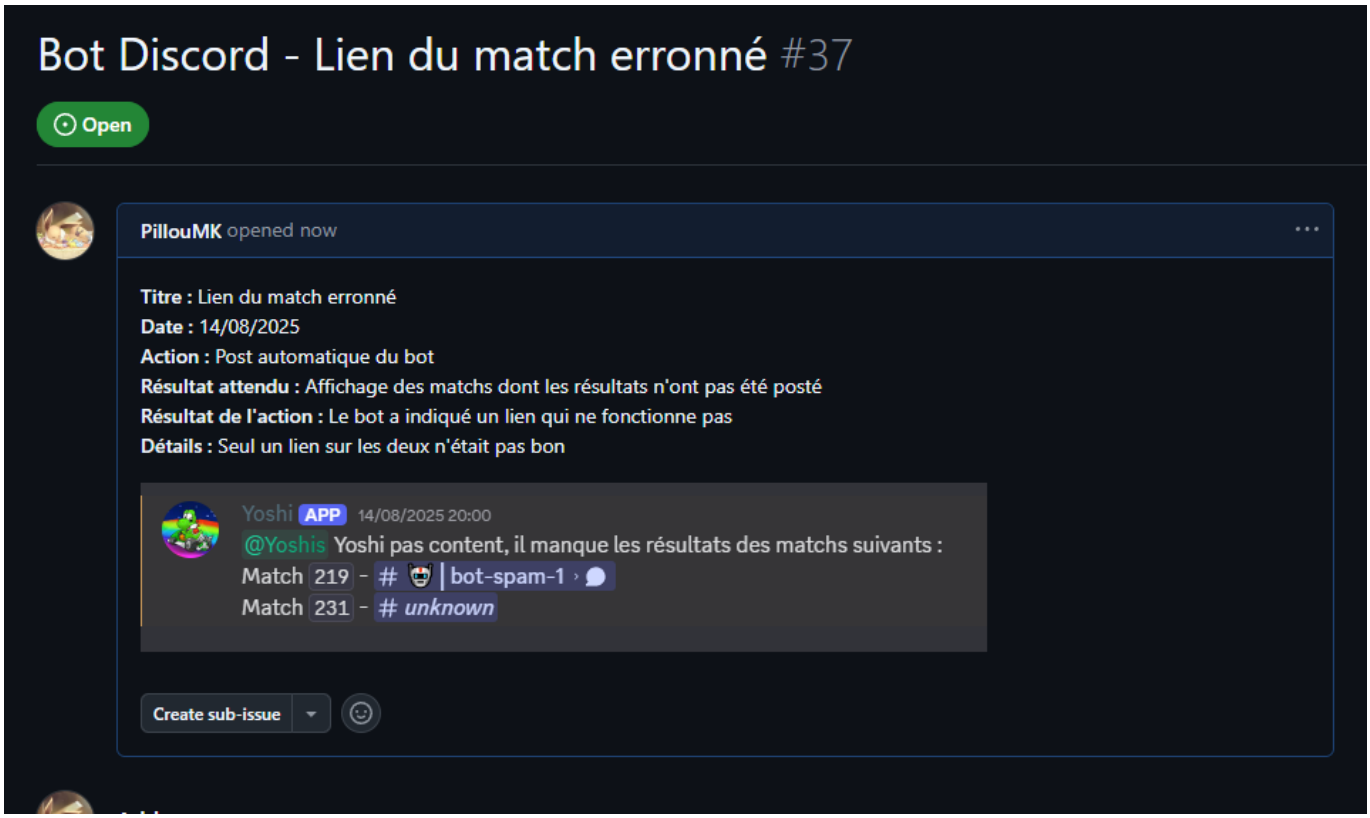
- Analyse :
  - Analyse du signalement
  - Lecture des logs
  - Tentative de reproduction du bug
  - Analyse du codé concerné lorsque celui-ci est identifié
- Correction :
  - Correction du bug sur une branche dédié (fix/nom-du-bug), commit "fix: xxx #Numéro issue"
  - Ajout de nouveaux tests
- Validation :
  - Le correctif doit valider les tests
  - Merge dans la branche `staging`
  - Déploiement en `production` lors du prochain patch
  - Issue fermée automatiquement lorsque le correctif est déployé

### Exemple d'un bug corrigé :

Signalement sur le canal de discussion :




Création de l'issue Github :




Analyse :

Seul un lien sur deux ne fonctionnait pas, il a fallit donc rechercher la différence qu'il y a eu entre les deux matchs, On peut remarquer dans les logs que le match concerné par le bug s'est terrminé avec une commande `edit_race` (modification du résultat d'une course, probablement suite à une erreur lors de la commande précédente)

Reproduction de l'action :

- Création d'un match
- Ajoute les courses
- Fini sur une modification

Constat :

- Le lien du match est aussi erronné, l'erreur provient donc de la commande `edit_race`

Correction :

- La commande `edit_race` ne sauvegardait pas correctement le lien. Ajout de la même fonction qui est utilisé dans `race` pour le sauvegarder
- Ajout du test vérifiant la sauvegarde du lien lors de la commande `edit_race`

Validation :

- Correctif ajouté sur la branche stagging
- Tests validés, déploiement lors du prochain patch en production
- Déploiement fait, fermeture automatique du ticket

# 5. Documentation technique - C2.4.1

## 5.1 Manuel de déploiement

Voici le manuel de déploiement, permettant de rapidement lancer les projets en version locales, ce manuel est divisé en deux parties, la partie API Rest et la partie du Bot Discord

### API Rest

Version node du projet : `v22.14.0`

Au préalable, vous devez avoir installé et configuré sur votre ordinateurs les logiciels et outils suivants :

- NodeJS
- Docker
- PostgreSQL

Pour commencer, vous devez initialisé le fichier .env, vous trouverez le fichier .env.local qui présentera l'exemple à suivre. Les variables sont prête à l'emploi :

```
DATABASE_URL="postgresql://postgres:postgres@db:5432/nest_db"
API_KEY=api_key_in_uuidv4
```

Ensuite, installer les dépendances :

```
npm install
```

Et maintenant, lancer l'application avec la commande suivante :

```
docker compose up -d
```

Voila l'API est maintenant lancé ! Vous pouvez utiliser Postman pour tester les endpoits, pour configurer Postman, allez à l'endpoint : `http://localhost:3000/docs-json` et téléchargez le ficchier JSON

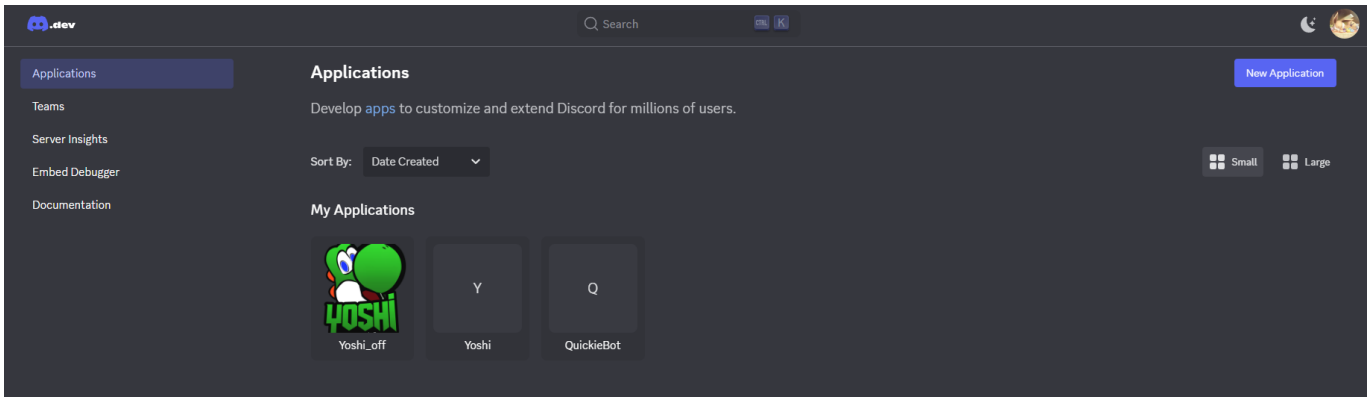
Maintenant dans Postman, vous pouvez charger la collection à partir de ce fichier, tout sera configuré pour les endpoits, vous n'aurez qu'à renseigner l'URL et l'API_KEY

**Note :** Lors du lancer du projet avec Docker, la base de données est automatiquement remplit avec des données factice grâce au script prisma/seed.ts

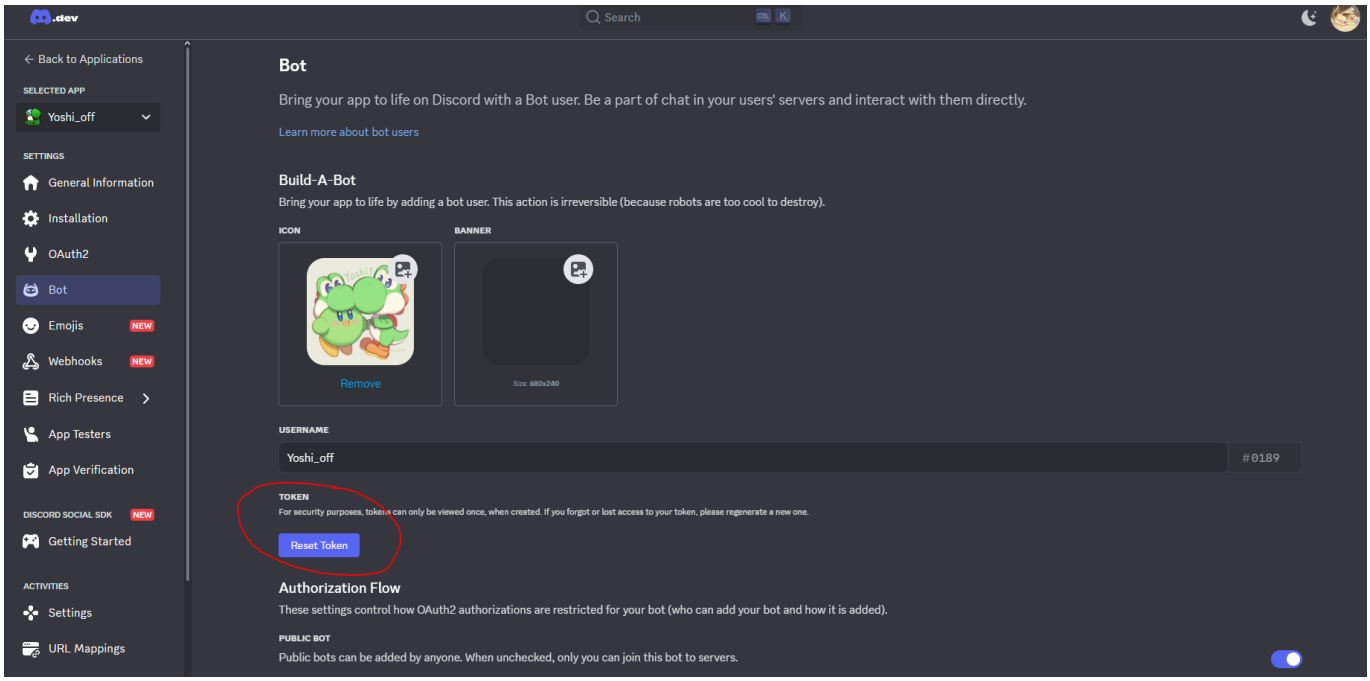
### Bot Discord

Pour lancer le projet du Bot Discord, vous devez au préalable posséder un Bot Discord, si ce n'est pas le cas, rendez vous ici : https://discord.com/developers

Connectez vous et allez dans l'onglet "Applications" et faites "New Application"




Ensuite, allez dans l'application que vous venez de créer, puis dans "Bot", vous trouverez le Token de votre bot nouvellement créé. Attention, si ce token vient à se retrouver en ligne, il sera automatiquement reset par Discord. Copiez ce token et enregistrez le dans le fichier .env dans la variable `DISCORD_TOKEN`




Enfin, vous devez générer le lien d'invitation pour inviter votre bot sur le serveur discord de développement. Pour ceci, allez dans l'onglet "OAuth2", et dans l'encadré OAuth2 URL Generator, cochez "Bot", donnez lui les permissions que vous souhaitez et copier l'URL généré.

Mettez cette URL dans votre navigateur, vous pourrez ainsi inviter le bot dans les serveurs où vous êtes administrateur.

Une fois cela fait, vous pourrez lancer le projet

Installer les dépendances :

```
npm install
```

Initialisez le fichier .env :

```
DISCORD_TOKEN="TOKEN"
CLIENT_ID="Bot Discord ID"
```

```
GUILD_ID="Dev Server ID"
API_KEY="API Rest Key"
```

Pour avoir accès à l'ID du Bot Discord et du Serveur Discord de développement vous devez avoir activé le mode développeur sur Discord. Une fois cela fait, vous pouvez récupérez l'ID de n'importe quel élément en faisant clique droit dessus.

Enfin, lancez la commande :

```
npm run dev
```

Et voila ! Le bot est désormais en ligne !

## 5.2 Manuel d'utilisation

Voici le manuel d'utilisation de l'application (Bot Discord), ce guide va vous guidez pas à pas pour installer et configurer le bot sur votre serveur

Vous aurez ensuite une présentation de toutes les commandes disponibles

### Installation :

Pour inviter le bot, utilisez l'URL suivante : `https://discord.com/oauth2/authorize?client_id=801177258920247307&permissions=563948533705792&integration_type=0&scope=bot`

Entrez cette URL dans votre navigateur, vous pourrez à partir de là inviter le bot sur votre serveur (Vous devez posséder les droits nécessaires)

Une fois sur le serveur, vous devez configuré votre équipe avec la commande suivante : `/set_team` name: "Nom de l'équipe" result channel: "Canal reservé aux résultats"

Une fois cela fait, vous pourrez utiliser pleinement le bot

**Voici la liste des commandes :**

**Match :** Ensembles des commandes destinées à la gestion d'un match

| Commande | Objectif | Paramètres |
|---|---|---|
| `/startwar` | Créer un match | **team1:** "Votre équipe",<br>**team2:** "Opposant"<br>**game:** "Jeu Mario kart (par défaut: **Mario kart World**)" |
| `/race` | Ajouter une course au match | **spots:** Les 6 spots de votre équipe à l'issue de la course, exemple : "1 2 5 6 8 9",<br>**map:** Le tag de la course jouée |

| Commande | Objectif | Paramètres |
|---|---|---|
| `/edit_race` | Modifier une course du match | **spots:** Les 6 spots de votre équipe à l'issue de la course, exemple : "1 2 5 6 8 9",<br>**map:** Le tag de la course jouée,<br>**race: (optionnel)** Le numéro de la course à modifier, par défaut cela modifie la dernière course |
| `/pena` | Ajouter une pénalité à une équipe | **team:** "Vous ou l'adversaire",<br>**pénalité:** Le montant de point à retirer |
| `/stopwar` | Finir un match, cela indiquera l'identifiant du match qui servira à la publication | **force (optionnel):** "Vrai ou faux", pour forcer l'arrêt d'un match qui n'est pas arrivé à terme. **(Attention, c'est définitif)** |

**Résultat d'un match :** Ensembles des commandes destinées à la gestion des résultats des matchs

| Commande | Objectif | Paramètres |
|---|---|---|
| `/lineup_war` | Lister les joueurs d'un match, cela ouvrira un menu de sélection où vous pourrez sélectionner les joueurs de votre équipe ayant jouer un match<br>Cela générera un texte à copier et remplir selon les isntruction donné par le bot | Pas de paramètres |
| `/preview_match` | Génère le tableau de résultat qui devra être validé | **id:** identifiant du match donné par `/stopwar`,<br>**text:** Le texte généré par `/lineup_war` que vous avez remplit |

Si /preview_match fonctionne, le tableau sera affiché avec les boutons "Valider" et "Annuler" **Valider**: Valide le tableau et publie le résultat dans le canal des résultats **Annuler**: Annule la commande et supprimer les boutons, vous devrez donc recommencer `/preview_match`

**Lineups :** Ensembles des commandes destinées aux disponibilités des membres

| Commande | Objectif | Paramètres |
|---|---|---|
| `/lineup` | Lister les joueurs disponibles (ou non) pour un horaire donné pour l'organisation des matchs | **hour:** horaire voulu, il est possible d'en indiquer plusieurs, exemple : "21 22" |
| Boutons de `/lineup` | Inscrire l'utilisateur à la lineup | **Can:** Ajoute le joueur comme étant disponible<br>**Maybe:** Ajoute le joueur comme étant peut-être disponible<br>**Cant:** Ajoute le joueur comme n'étant pas dispobile<br>**Voir line up mixte/roster:** Permet d'afficher les disponibilités par roster s'ils existent |

**Statistiques des courses :**

| Commande | Objectif | Paramètres |
|---|---|---|
| `/projectmap` | Afficher les statistiques des courses de l'équipe | **Month: (optionnel)** Nombre de mois de données à prendre en compte<br>**roster: (optionnel)** Filtrer par roster si existants<br>**game: (optionnel)** Jeu à prendre en compte (par défaut : **Mario Kart World**) |

**Contre-la-montre :** Gestion du mode contre-la-montre de l'équipe

| Commande | Objectif | Paramètres |
|---|---|---|
| `/set_tt` | Ajouter ou modifier un temps | **time:** Temps réalisé au format **"x.xx.xxx"**<br>**roster:** Filtrer par roster si existants<br>**map:** tag de la course jouée<br>**game: (optionnel)** Jeu Mario kart (par défaut: **Mario Kart World**)<br>**no_item: (optionnel)** Enregistre le temps comme étant "Sans objets" |
| `/classement` | Afficher le classement de l'équipe d'une course | **map:** tag de la course jouée<br>**game: (optionnel)** Jeu Mario kart (par défaut: **Mario Kart World**)<br>**no_item: (optionnel)** Affiche le classement du monde "Sans objets" |

## 5.3 Manuel de mise à jour

### 5.3.1 Manuel de mise à jour de l'API Rest :

Ce manuel a pour but de guider le développeur sur les différentes points de l'API qui peuvent être mis à jours.

- Evolutions de la BDD et migrations
- Evolutions des endpoints
- Evolutions des services
- Evolutions des ressources

### Evolutions de la BDD et migrations

Le projet NestJS est lié à la BDD via l'ORM Prisma, c'est à travers lui qu'est faites l'interaction avec la base de données. Le fichier `prisma/schema.prisma` est le modèle de la base de données, c'est ce fichier qui permet de modifier le modèle.

**Ajouter une nouvelle table ou un champ**

Modifier le fichier `prisma/schema.prisma` :

```
model Example {
  id        String   @id @default(uuid())
```

```
  name      String
  createdAt DateTime @default(now())
}
```

Générer la migration :

```
npx prisma migrate dev --name add_example_table
```

Vérifiez que le fichier de migration SQL a bien été créé dans `prisma/migrations`

**Note :** Attention, certains changements destructifs (DROP COLUMN) peuvent entraîner une perte de données.

### Evolutions des endpoints

Les endpoints REST sont définis dans les Controllers NestJS `src/[resource]/[resource].controller.ts`.

Pour ajouter un endpoint, il faut ajouter une nouvelle méthode dans le controller

```
@Get(':id')
findOne(@Param('id') id: string) {
  return this.exampleService.findOne(id);
}
```

**Note :** Faites attention à ne pas créer un endpoint qui aurait le même type d'URL

Par exemple :

`api/match/:id`
`api/match/complete`

Si deux endpoints sont déclarés dans cet ordre, la seconde URL ne serait jamais atteinte, puisque `complete` serait interpreté comme `:id`

Documentation de l'endpoint :

- Documenter l'endpoint avec les décorateurs @ApiTags, @ApiResponse de `@nestjs/swagger` pour que la documentation de l'API, générée automatiquement, soit correcte.

### Evolutions des services

La logique métier se trouve dans les Services `src/[resource]/[resource].service.ts`.

La fonction doit être ajouter dans la classe du service, elle pourra être utilisé ensuite dans le controlleur associé

```
async findOne(id: string) {
  return this.prisma.example.findUnique({ where: { id } });
}
```

### Evolutions des ressources

Les ressources correspondent aux modules NestJS et regroupent les controllers, services et entités associées.

Pour créer une nouvelle ressource, vous pouvez utiliser la commande suivante dans le terminal :

```
npx nest g resource example
```

### Bonne pratique de mise à jour :

- Créer une branche dédiée à la modification, exemple : `feature/add-example`
- Lancer les tests unitaires après chaque évolution, pour éviter toute regression ou dysfonctionnement, avec :

```
npm test
```

### 5.3.2 Manuel de mise à jour du Bot Discord :

Voici l'architecture du projet du Bot Discord :

- `src/`
  - `buttons/` - Script lors d'interaction avec un bouton discord
  - `commands/` - Script lors de l'utilisation d'une slash command
  - `controller/` - Logique métier
  - `database/` - Fichiers JSON de stockage local
  - `model/` - Typage
  - `select_menus/` - Script lors d'interaction avec un menu de selection discord
  - `config.ts`
  - `deploy-commands.ts` - Script pour déployer les slash commands
  - `global.ts` - Variables globales
  - `index.ts`
  - `.env` - variables d'environnements

Ce manuel a pour but de guider le développeur sur les différentes points de l'API qui peuvent être mis à jours.

- Evolutions des slash commandes
- Evolutions des boutons
- Evolutions des controller

### Evolutions des slash commandes

Pour ajouter une slash commandes, vous devez créer un fichier typescript dans `src/commands/**/new-command.ts`, la structure d'un tel fichier est la suivante :

```
module.exports = {
  data: new SlashCommandBuilder()
    .setName("name")
```

```
    .setDescription("description")
    .addStringOption((option) =>
      option.setName("argument").setDescription("argument").setRequired(true)
    ),

  async execute(interaction: ChatInputCommandInteraction) {
    // Script executé lors qu'un utilisateur utilise la slash commande
  },
};
```

L'option `setName` permet de nommer la commande, ce nom doit être unique et en minuscule sans caractères spéciaux

La fonction `execute` est la fonction executé lorsqu'un utilisateur utilise la commande

Lorsque vous avez créé la commande, vous devez la **déployer** pour que celle-ci soit utilisable sur Discord :

Pour déployer instantanément la commande sur le serveur de test :

```
npm run deploy:dev
```

Pour déployer la commande de manière globale :

```
npm run deploy:prod
```

Attention, le déploiement global peut prendre jusqu'à 1 heure

## Evolutions des boutons

De manière similaire aux slash commands, pour ajouter un bouton, vous devez créer un fichier typescript dans `src/buttons/**/new-button.ts`, la structure d'un tel fichier est la suivante :

```
module.exports = {
  data: {
    name: "new_button",
  },

  async execute(interaction: ButtonInteraction, args: string[]) {},
};
```

La structure du fichier est très similaire, la fonction `execute` est la méthode executé lorsque l'utilisateur interragit avec le bouton

C'est le bot qui génère un bouton, le bouton peut être attaché à un message posté par le bot, le bot surveille ensuite les interactions qu'il reçoit sur les boutons, chaque bouton possède un `custom_id`, c'est comme ça que le bot va savoir quoi faire

Pour générer une liste de bouton :

```
const makeButtonList = (): ActionRowBuilder<ButtonBuilder> => {
  let param1 = "param1";
  let param2 = "param2";
  return new ActionRowBuilder<ButtonBuilder>().addComponents(
    new ButtonBuilder()
      .setCustomId(`new_button-${param1}-${param2}`)
      .setLabel("Button label")
      .setStyle(ButtonStyle.Primary)
  );
};
```

`setCustomId` permet de renseigner le nom du bouton et les paramètres de celui-ci, séparés par des "-", sous la forme de strings. Le premier string est le nom du bouton, il doit correspondre au name du script src/buttons/**/**.ts associé

## Evolutions des controller

La logique métier est centralisé dans les controllers dans `src/controller/**.ts`

Chaque ressource possède son controller, de manière générale, un controller est associé à une catégorie de slash commands. Dedans, on y export les méthodes qui seront ensuité utilisés dans les scripts des commandes/boutons/select_menus