# Architecture de l'Application Wordle Multijoueur

## 1. Navigateur de l'utilisateur

- **React avec TypeScript** : Crée une interface utilisateur dynamique et réactive. TypeScript apporte une sécurité supplémentaire grâce au typage statique.
- **Context API** : Gestion légère de l'état de l'application, partage des données entre composants sans bibliothèques tierces.
- **Apollo Client** : Facilite la communication avec le backend GraphQL, gère les requêtes, le cache et les abonnements.
- **React Router** : Gère le routage côté client pour une navigation fluide sans rechargement complet.

## 2. VPS Hostinger

Fournit une solution d'hébergement abordable avec des performances satisfaisantes, permettant le déploiement de conteneurs Docker.

### 2.1. Docker Host

Environnement sur le serveur Hostinger qui exécute et orchestre les conteneurs Docker, assurant cohérence et facilitant le déploiement et la scalabilité.

### 2.2. NGINX (Reverse Proxy)

- **Rôle** : Reverse proxy et équilibreur de charge pour les requêtes entrantes.
- **Avantages** :
    - Gère les certificats SSL/TLS
    - Améliore la sécurité et les performances
    - Prépare l'infrastructure pour une éventuelle scalabilité

### 2.3. Conteneur Docker : Application Node.js

**Composants internes :**

- Express Server
- Apollo Server
- Prisma ORM
- Module d'Authentification
- Module d'Autorisation
- GraphQL Subscriptions
- Gestion des Logs (Morgan)
- Surveillance des Erreurs (Sentry Agent)

**Justifications :**

- **Express Server** : Framework web flexible pour construire l'API et gérer les middlewares.
- **Apollo Server** : Crée une API GraphQL puissante avec des fonctionnalités avancées.

- **Prisma ORM** : Simplifie les opérations CRUD et gère les migrations de schéma.
- **Module d'Authentification** : Sécurise l'accès via JWT et OAuth 2.0.
- **Module d'Autorisation** : Contrôle l'accès aux fonctionnalités selon les rôles.
- **GraphQL Subscriptions** : Gère les communications en temps réel.
- **Morgan** : Facilite le débogage et le suivi des activités.
- **Sentry Agent** : Surveille les erreurs en temps réel pour améliorer la fiabilité.

### 2.4. Conteneur Docker : Redis

**Rôle** : Base de données en mémoire pour le cache et la gestion des sessions, améliorant les performances.

### 2.5. Conteneur Docker : PostgreSQL

**Rôle** : Base de données relationnelle robuste pour le stockage persistant des données.

## 3. Services Externes

- **Mailgun API** : Envoi d'emails transactionnels.
- **Google OAuth 2.0** : Authentification externe via Google.
- **Let's Encrypt** : Fournit des certificats SSL/TLS gratuits.
- **Sentry Service** : Surveillance des erreurs et des performances.

## 4. CI/CD (Intégration Continue / Déploiement Continu)

**GitHub Actions** : Automatise les tests, la construction et le déploiement de l'application.

## 5. Contrôle de Version

**Dépôt GitHub** : Gestion du code source avec stratégie GitFlow (main, dev, epic).

## Relations et interactions entre les composants

1. L'utilisateur interagit avec l'Application React, qui communique avec le backend via HTTPS.
2. NGINX reçoit les requêtes et les achemine vers le Serveur Express.
3. Express Server utilise Apollo Server pour traiter les requêtes GraphQL.
4. Apollo Server interagit avec Prisma ORM pour les opérations sur la base de données.
5. Prisma ORM communique avec PostgreSQL pour exécuter les requêtes SQL.
6. Express Server utilise Redis pour la gestion des sessions et le cache.
7. Le Module d'Authentification interagit avec Google OAuth 2.0.
8. Le conteneur Node.js utilise Mailgun API pour l'envoi d'emails.
9. Express Server utilise Morgan pour les logs HTTP.
10. Sentry Agent capture les erreurs et les envoie au Sentry Service.
11. NGINX utilise Let's Encrypt pour les certificats SSL/TLS.
12. Les modifications dans le Dépôt GitHub déclenchent des GitHub Actions pour le CI/CD.