Green Escape

L’implémentation

Victor Dallon - Roland Fontanes - Hugo Leyx-Valade - Mathieu Alassoeur

Sommaire

[Architecture du projet 3](#_Toc196765982)

[Structures des dossiers 3](#_Toc196765983)

[Dossiers principaux 3](#_Toc196765984)

[Flux de données 4](#_Toc196765985)

[Modèles de données 4](#_Toc196765986)

[Modèle CustomUser 4](#_Toc196765987)

[Modèle PlayerTimePerSeed 5](#_Toc196765988)

[Fonctionnalités principales 6](#_Toc196765989)

[Gestion des utilisateurs 6](#_Toc196765990)

[Génération de labyrinthes 6](#_Toc196765991)

[Classement et scores 6](#_Toc196765992)

[Sauvegarde 7](#_Toc196765993)

[Algorithmes 7](#_Toc196765994)

[Génération de labyrinthe 7](#_Toc196765995)

[Résolution de labyrinthe 7](#_Toc196765996)

[Diagrammes 10](#_Toc196765997)

[Diagramme de Classe 10](#_Toc196765998)

[Diagramme Séquentiel 10](#_Toc196765999)

## Architecture du projet

### Structures des dossiers

green-escape/

├── backend/

│ ├── GreenEscape-backend/

│ │ ├── api/ # Contient les fichiers liés à l'API Django

│ │ │ ├── \_\_init\_\_.py # Fichier d'initialisation du module

│ │ │ ├── views.py # Logique métier et gestion des requêtes HTTP

│ │ │ ├── models.py # Définition des modèles de la base de données

│ │ │ ├── urls.py # Définition des routes de l'API

│ │ │ ├── serializers.py # Sérialisation des données pour l'API REST

│ │ │ ├── tests.py # Tests unitaires pour l'API

│ │ ├── static/ # Fichiers statiques (CSS, JS, images)

│ │ │ ├── css/ # Feuilles de style CSS

│ │ │ ├── js/ # Scripts JavaScript

│ │ │ ├── images/ # Images utilisées dans le projet

│ │ ├── templates/ # Templates HTML pour le rendu des pages

│ │ │ ├── views/ # Pages HTML (login, profil, etc.)

│ │ ├── settings.py # Configuration principale du projet Django

│ │ ├── urls.py # Routes principales du projet

│ │ ├── wsgi.py # Point d'entrée pour le déploiement

│ │ ├── asgi.py # Point d'entrée pour les applications asynchrones

│ │ ├── manage.py # Script de gestion du projet Django

│ ├── requirements.txt # Liste des dépendances Python

├── README.md # Documentation du projet

### Dossiers principaux

#### api/ :

Contient toute la logique métier et les fonctionnalités principales de l'application.

1. views.py : Définit les fonctions qui gèrent les requêtes HTTP (par exemple, saveBestTime, update\_player\_stats).
2. models.py : Définit les modèles de données (par exemple, PlayerTimePerSeed, CustomUser).
3. urls.py : Associe les endpoints de l'API aux fonctions correspondantes dans views.py.
4. serializers.py : Sérialise les données pour les rendre compatibles avec l'API REST.

#### static/ :

Contient les fichiers statiques utilisés par le front-end.

1. css/ : Feuilles de style pour le design des pages.
2. js/ : Scripts JavaScript pour les interactions dynamiques (par exemple, génération de labyrinthes, enregistrement des scores).
3. images/ : Images utilisées dans l'interface utilisateur.

#### templates/ :

Contient les fichiers HTML utilisés pour le rendu des pages.

1. views/ : Pages HTML comme login.html, profile.html, scoreboard.html.

#### Fichiers de configuration :

1. settings.py :

Configure la base de données (MongoDB via Djongo).

Définit les applications installées, les middlewares, et les paramètres globaux.

1. urls.py :

Définit les routes principales du projet (par exemple, /api/save-best-time/, /api/update-stats/).

#### manage.py :

Script principal pour gérer le projet Django (exécuter le serveur, appliquer les migrations, créer un superutilisateur, etc.).

### Flux de données

#### Front-end :

L'utilisateur interagit avec l'interface via des pages HTML, CSS et JavaScript.

Les actions (comme résoudre un labyrinthe ou mettre à jour un profil) déclenchent des appels API via fetch.

#### Back-end :

Les requêtes API sont gérées par les vues définies dans views.py.

Les données sont récupérées ou mises à jour dans la base de données MongoDB via les modèles définis dans models.py.

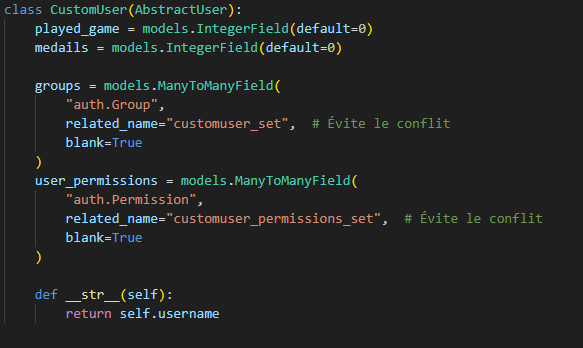
#### Base de données :

MongoDB stocke les informations des utilisateurs, les scores, et les labyrinthes générés.

## Modèles de données

### Modèle CustomUser

Le modèle **CustomUser** est une extension du modèle utilisateur par défaut de Django. Il contient des informations supplémentaires spécifiques au projet, comme le nombre de parties jouées et les médailles gagnées.

Le voici :

Champs principaux :

played\_game : Nombre de parties jouées par l'utilisateur.

medails : Nombre de médailles gagnées par l'utilisateur.

Hérite également des champs du modèle utilisateur par défaut de Django, comme :

username

email

password

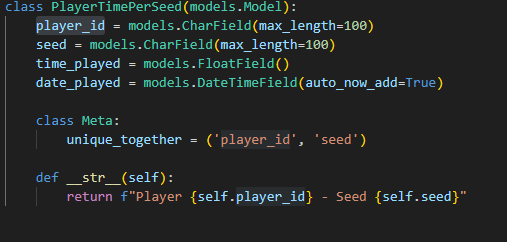
is\_staff

is\_superuser

date\_joined

### Modèle PlayerTimePerSeed

Ce modèle enregistre les meilleurs temps des joueurs pour chaque seed utilisée pour générer un labyrinthe.

Le voici :

## Fonctionnalités principales

### Gestion des utilisateurs

#### Inscription et connexion

Les utilisateurs peuvent créer un compte avec un nom d'utilisateur, un mot de passe et une adresse e-mail.

Authentification sécurisée avec Django.

#### Modification de Profil

Les utilisateurs peuvent mettre à jour leur nom d'utilisateur, leur mot de passe et d'autres informations personnelles.

#### Statistiques Personnelles

Chaque utilisateur peut consulter ses statistiques, comme le nombre de parties jouées ([played\_game](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)) et le nombre de médailles gagnées ([medails](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)).

### Génération de labyrinthes

#### Génération basée sur une seed

Les labyrinthes sont générés de manière déterministe en utilisant une graine aléatoire ou fournie par l'utilisateur.

#### Algorithmes de résolution

Les labyrinthes peuvent être résolus automatiquement par des algorithmes comme A\*, Dijkstra ou BFS.

#### Personnalisation de la seed

Les utilisateurs peuvent choisir une graine spécifique pour générer un labyrinthe.

### Classement et scores

#### Classement Global

Affiche les 10 meilleurs joueurs en fonction du nombre de médailles gagnées.

#### Meilleur temps par seed

Enregistre et affiche les meilleurs temps pour chaque graine.

### Sauvegarde

#### Parties jouées / Médailles

Ajoute automatiquement la partie jouée et les médailles éventuellement gagnées à la base de données

#### Meilleurs temps

Après avoir terminé un labyrinthe, le temps du joueur est comparé au meilleur temps existant pour cette graine.

Si le nouveau temps est meilleur, il est enregistré dans la base de données.

## Algorithmes

### Génération de labyrinthe

La génération des labyrinthes repose sur une seed pour garantir un résultat déterministe. Cela signifie que la même graine produira toujours le même labyrinthe.

Le labyrinthe est généré en divisant l'espace en sous-sections et en creusant des chemins de manière récursive.

Une seed est utilisée pour initialiser un générateur pseudo-aléatoire, garantissant que le labyrinthe est reproductible.

Etapes de l’algorithme :

1. Initialiser une grille vide.
2. Choisir un point de départ aléatoire (basé sur la graine).
3. Creuser des chemins en visitant les cellules voisines de manière récursive.
4. Ajouter des murs pour séparer les chemins.

Toute l’implémentation est disponible dans le fichier Green-Escape\backend\GreenEscape-backend\api\algorithms\algo.py

### Résolution de labyrinthe

Tous l’implémentation de ces algorithmes est disponible dans le fichier Green-Escape\backend\GreenEscape-backend\api\algorithms\solver.py

#### A\*

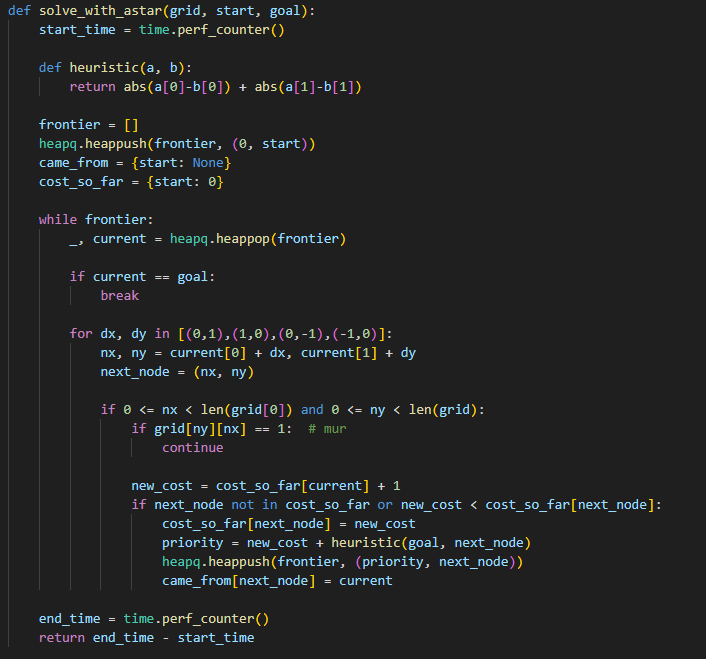
A\* est un algorithme de recherche de chemin optimal qui utilise une fonction heuristique pour guider la recherche.

La distance de Manhattan est utilisée comme heuristique : h(n) = |x1 - x2| + |y1 - y2|.

Il combine le coût réel du chemin (g(n)) et une estimation du coût restant (h(n)).

Étapes principales :

1. Ajouter le point de départ à une file de priorité.
2. Explorer les voisins du nœud actuel.
3. Mettre à jour les coûts et ajouter les voisins à la file.
4. Répéter jusqu'à atteindre le point final ou épuiser les nœuds.

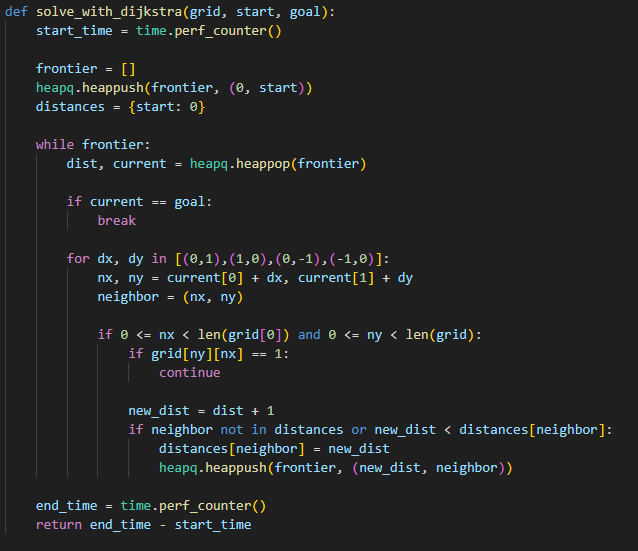
Le voici :

#### Djikstra

Dijkstra est un algorithme de recherche de chemin qui trouve le chemin le plus court entre un point de départ et tous les autres points.

Étapes principales :

1. Initialiser les distances de tous les nœuds à l'infini, sauf le point de départ (distance = 0).
2. Ajouter le point de départ à une file de priorité.
3. Explorer les voisins du nœud actuel et mettre à jour leurs distances.
4. Répéter jusqu'à atteindre le point final ou épuiser les nœuds.

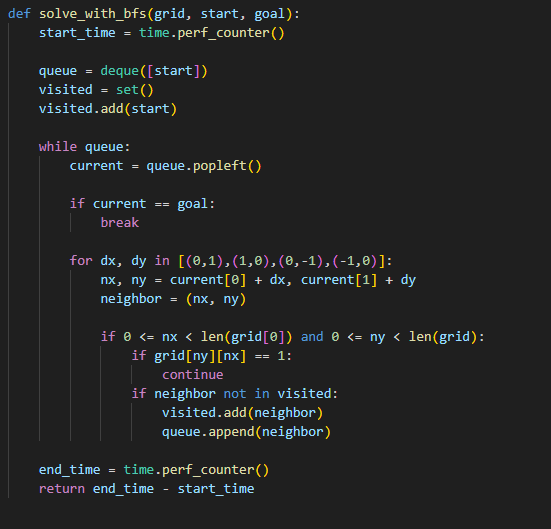
Le voici :

#### Breadth-First Search (BFS)

Le BFS explore tous les nœuds à une distance donnée avant de passer à la distance suivante.

Étapes principales :

1. Ajouter le point de départ à une file.
2. Explorer les voisins du nœud actuel et les ajouter à la file.
3. Répéter jusqu'à atteindre le point final ou épuiser les nœuds.

Le voici :

## Diagrammes

### https://lh7-rt.googleusercontent.com/docsz/AD_4nXfp6zgqRhBeJSMNN2RjouT7BezDsQ2axoj3EP6DGI_GVHBz0V8ZTphj9kR8tMapHVYjFmcTaY92XYU52ixaatD2L0kwZBPtAeIQpTIbAHXJ5Kgxh-XgsHkp6V5Iz1ZDm9XpDKsp1A?key=xPmP8VnZTsGZlsEN5DP3iJLP Diagramme de Classe

### **https://lh7-rt.googleusercontent.com/docsz/AD_4nXeSR0abDsgZtjQgDreWzeuSqGJ_uVkIjPtCmlR3dNt0AjAlijxeyGHjCTVCAibGQfd4opzGj6ZVPjyO8inDev4aRXd9U6bdQcIwomBhtq9o7Y2IpkhizHv1bPXvRgkqwk6iG36JRA?key=xPmP8VnZTsGZlsEN5DP3iJLP** Diagramme Séquentiel