Choix des technologies pour la partie serveur du projet

February 11, 2025

Introduction

Dans ce projet, le serveur joue un rôle central : il doit gérer la logique du jeu, la communication entre les joueurs, ainsi que l'accès à la base de données. Selon la technologie utilisée pour le client (front-end), nous avons plusieurs options pour structurer notre back-end. Le serveur devra assurer plusieurs fonctionnalités clés telles que la gestion des joueurs, des parties, la communication en temps réel via WebSockets et l'interaction avec la base de données.

Responsabilités du serveur

Le serveur devra assurer plusieurs fonctionnalités clés :

- **Gestion des joueurs** : Connexion, authentification, stockage des données des joueurs.
- **Gestion des parties** : Création des salles, synchronisation des actions des joueurs.
- **Communication en temps réel** : Via WebSockets pour permettre le multijoueur.
- **Interaction avec la base de données** : Stockage des scores, progression des joueurs, historique des parties.

Options de serveur

Option 1 : Serveur en Python (Flask/FastAPI + WebSockets)

Technologies utilisées :

- **Framework web** : Flask ou FastAPI (FastAPI est plus rapide et optimisé pour les WebSockets).
- **Communication temps réel** : WebSockets avec `websockets` ou `socket.io` en Python.
- **Base de données** : PostgreSQL/MySQL avec SQLAlchemy, ou MongoDB (NoSQL).

Avantages :

- Facilité d'intégration avec un client Godot (GDScript) ou Pygame.
- Bonne gestion des WebSockets avec FastAPI.
- Performance correcte pour un jeu 2D multijoueur.

Inconvénients :

- Moins performant que Node.js pour gérer de nombreuses connexions simultanées.
- Nécessite une bonne gestion des WebSockets en production.

Exemple de serveur WebSockets en Python avec FastAPI :

```
from fastapi import FastAPI, WebSocket
import uvicorn

app = FastAPI()

@app.websocket("/ws")
```

```
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()
    while True:
        data = await websocket.receive_text()
        await websocket.send_text(f"Message reçu : {data}")

if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Option 2 : Serveur en Node.js (Express + Socket.io)

Technologies utilisées :

- **Framework web** : Express.js pour gérer les routes et API REST.
- **Communication temps réel** : `socket.io` pour gérer les WebSockets.
- **Base de données** : PostgreSQL/MySQL (via Sequelize) ou MongoDB (via Mongoose).

Avantages :

- Très performant pour les WebSockets grâce à l'event loop de Node.js.
- Facilement compatible avec un client en Electron/React.
- Écosystème riche avec de nombreux modules pour le développement rapide.

Inconvénients :

- Consomme plus de mémoire que Python pour certaines tâches lourdes.
- Nécessite une bonne gestion des erreurs pour éviter les fuites de mémoire dans un projet multijoueur.

Exemple de serveur WebSockets avec Node.js + Socket.io :

```
const express = require('express');
const http = require('http');
const { Server } = require('socket.io');

const app = express();
const server = http.createServer(app);
const io = new Server(server);

io.on('connection', (socket) => {
    console.log('Un joueur s'est connecté !');

    socket.on('message', (msg) => {
        console.log(`Message reçu : ${msg}`);
        socket.emit('response', `Serveur a reçu : ${msg}`);
    });

    socket.on('disconnect', () => {
        console.log('Un joueur s'est déconnecté.');
    });
});

server.listen(3000, () => {
    console.log('Serveur lancé sur http://localhost:3000');
});
```

Selon notre projet

Voici un tableau récapitulatif des choix possibles pour le serveur en fonction du projet :

Option	Stack Serveur	Gestion des WebSockets	Base d
Python (Flask/FastAPI)	Python + Flask/FastAPI + WebSockets	`websockets` ou `socket.io`	SQLAlchemy (Postgre
Node.js (Express)	Node.js + Express + Socket.io	`socket.io`	Sequelize (SQL) ou

Conclusion

Choix du Serveur

Le choix du serveur dépend du choix de la technologie côté client :

- Si le frontend est fait avec Godot ou Pygame, alors il est recommandé d'utiliser Python avec **FastAPI** et **WebSockets**.
- Si le frontend est fait avec **Electron** ou **React**, alors il est préférable d'utiliser **Node.js** avec **Express** et **Socket.io**.

Recommandations

- Si l'objectif est la simplicité et une bonne compatibilité avec Godot/Python, un serveur en **Python** est un bon choix.
- Si l'on souhaite un serveur plus robuste et scalable, avec des WebSockets performants, alors **Node.js** est la meilleure solution.