12/12/2024

Rapport project Java V

ZERO Bartlomiej

Table des matières

This project is designed to develop a **Multiplayer Game Management System** for an online multiplayer game. The system is split into two independent Spring Boot microservices:

1. **Player Management Service**: Manages player information, profiles, statistics, etc.
2. **Game Management Service**: Manages game sessions, scores, player participation, and game results.

The two services need to communicate with each other, specifically to update player statistics after a game ends based on performance.

---

1. Microservices Architecture

- The microservices architecture was chosen to separate the concerns of handling players and managing game sessions, making the system easier to scale and maintain. Each service is independently deployable and can be managed separately.
- Communication between the microservices is done using **REST APIs**.

Communication between Services:

- The Game Management Service communicates with the Player Management Service to create games by existing hosts and add existing users to the game.
- This communication is handled by **HTTP RESTful requests**.

Layers

- **Layered Architecture**: The system follows a layered architecture to ensure clear separation of concerns between **Controller**, **Service**, **Repository**, and **Entity** layers.
- **DTO Usage**: Data Transfer Objects (DTOs) are used for transferring data between layers, especially between the **Controller** and **Service** layers.
- **Spring Boot**: We have used Spring Boot to create a RESTful microservice architecture. **Spring Data JPA** is used to handle database interactions, while **Lombok** is used to minimize boilerplate code.
- **PostgreSQL**: PostgreSQL is used as the relational database due to its robustness and scalability.
- **Maven**: Maven is used as the build tool to manage dependencies and automate builds.
- **Etc**.

2. Database

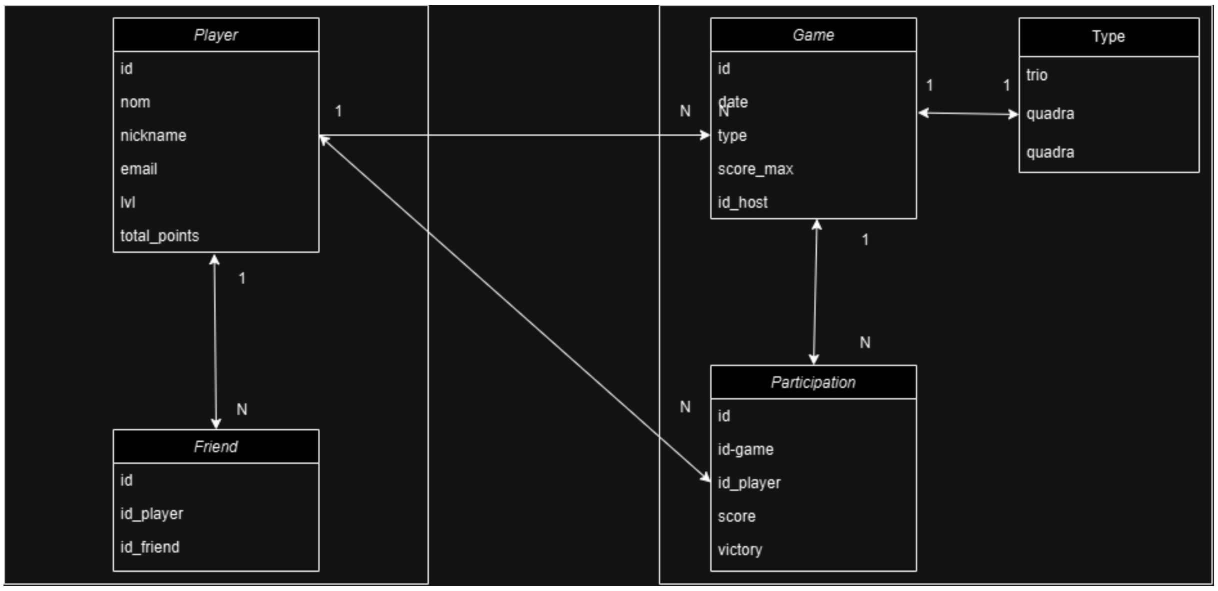
Player Management Service Database:

- The **Player Management Service** uses a relational database (PostgreSQL) to store player information. The following entities are included in the database:
  - Player: Contains fields like id, name, nickname, email, level, and total_points.

- Friend: Represents a friendship relation between players with fields like id, player_id, and friend_id.
- **Relationships**:
  - One player can have multiple friends (One-to-Many relationship).

- **Game Management Service Database**:
  - The **Game Management Service** uses relational database (PostgreSQL). The following entities are stored:
    - Game: Contains fields like id, date, game_type, max_score, and host_id.
    - Participation: This is a join table representing the participation of players in games, with fields like id, game_id, player_id, score, and victory_status.
    - **Relationships**:
      - One game is played by multiple players (Many-to-Many relationship between Game and Player through Participation).

Below is the database schema for both services:



Script: spring.jpa.hibernate.ddl-auto=update

3. REST API Communication

- **Player Management Service**:
  - Provides endpoints to create, update, delete, and retrieve player information, including statistics and friends.
  - Key endpoints:
    - POST /joueurs: Create a new player.
    - GET /joeurs/{id}: Retrieve a player's profile and statistics.

        - POST /joueurs/{id}/amis: Add a friend to a player’s friend list.

- **Game Management Service**:
    - Provides endpoints to create, modify, and delete game sessions, register player participation, and retrieve game details.
    - Key endpoints:
        - POST /parties: Create a new game session.
        - POST /parties/{id}/participations: Register a player’s participation in a game along with their score.
        - GET /partie/{id}: Retrieve information about a game session and its participants.

Usage Guide

1. Running the Services

Both the Player Management Service and the Game Management Service should be run separately. Both services are running on different ports.

- **Start the Player Management Service**:
    - Make sure to configure the application properties (application.properties) with the appropriate database connection settings.
    - Run the service via an IDE like IntelliJ IDEA or Eclipse.
    - It will run on port: localhost:8080.
- **Start the Game Management Service**:
    - Similarly, configure the application properties for the database connection.
    - Run the service via an IDE.
    - This service will run on different port localhost:8081.

2. API Endpoints

Once both services are running, you can test the endpoints using tools like **Postman**. Below are some examples of how to interact with the system:

Player Management Service Endpoints:

- **Create a player**:
    - Method: POST /joueurs
    - Request Body: json

```
{
  "name": "John Doe",
  "nickname": "johnny",
  "email": "john.doe@example.com",
  "level": 1,
  "total_points": 100
}
```

- **Retrieve player profile**:
    - Method: GET /joueurs/{id}

- **Delete player profile**:
    - Method: DELETE /joueurs/{id}
- **Update a player**:
    - Method: PUT /joueurs
    - Request Body: json

```
{
  "name": "John upd",
  "nickname": "johnny",
  "email": "john.upd@example.com",
  "level": 10,
  "total_points": 10
}
```

- **Add a friend**:
    - Method: POST /joueurs/{id}/amis
    - Request Body: json

```
{
"idFriend": 3
}
```

Game Management Service Endpoints:

- **Create a game**:
    - Method: POST /parties
    - Request Body: json

```
{
  "date": "2024-12-12",
  " typePartie": "duo",
  " scoreMax": 700,
  " idHote": 2
}
```

- **Update a game**:
  - Method: PUT /parties/{id}
  - Request Body: json

```
{
  "date": "2024-12-15",
  " typePartie": "trio",
  " scoreMax": 700,
  " idHote": 2
}
```

- **Retrieve a game**:
  - Method: GET /parties/{id}
- **DELETE a game**:
  - Method: DELETE /parties/{id}
  - Request Body: json
- **Register player participation**:
  - Method: POST /parties/{id}/participations
  - Request Body: json

```
{
  "playerId": 3,
  "score": 500
}
```

Installation Guide

1. **Clone the Repositories**:
   - Clone the GitHub repository for both microservices
2. **Install Dependencies**:
   - install dependencies(mvn clean + mvn install)
3. **Database Configuration**:
   - Configure the database settings for both services in src/main/resources/application.properties.

```
spring.datasource.url=jdbc:postgresql://localhost:5433/javadb
spring.datasource.username=bartek
spring.datasource.password=bartek
spring.h2.console.enabled=true
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=update
```

   -

4. **Run the Services**:
   - Launch the Services via RUN button (to be sure it is working, first service shows Hello, world! And the second Hello, world2!).
5. **Testing**:
   - Use **Postman** to test the REST API endpoints for both services.

Unfinished Features

1. **DAO Layer**: Although not strictly required in this project due to the use of Spring Data JPA, implementing a **DAO Layer** can help for advanced queries and custom database interactions.
2. **Testing**: Unit tests and integration tests should be implemented to ensure that the application works correctly. This includes testing services, repositories, and controllers, as well as testing communication between microservices.
3. **Automatic Player Stats Update**: After the game finishes, player statistics (e.g., victories, total score) should be updated automatically via REST API calls between the Game Service and Player Service.