Group Project Assignment: RESTful API for Naval Battle Game

Overview

In this Group Project, you are tasked with developing both the client and server components for a standalone naval battle game, strictly adhering to the provided OpenAPI specification.
The project aims to demonstrate the implementation of a robust client/server architecture. The server will manage the game logic, while the client, operating autonomously, will engage in the game by sending requests and processing responses based on the server's RESTful API.
This setup will underline the significance of a well-defined interface in software architecture, ensuring compatibility and seamless interaction between different implementations of clients and servers.

Objectives

- **Develop a RESTful API**: Construct server-side logic to manage game actions, adhering to RESTful principles and the provided OpenAPI specification.
- **Create an Autonomous Client**: Build a client that autonomously interacts with the server, implementing the game strategy based on the API.
- **Understand Client/Server Architecture**: Illustrate the interaction between the client and server, emphasizing separation of concerns and network communication.

Game Rules

The naval battle game is a strategy game where the client automatically plays against the server on a 10x10 grid. The client's objective is to destroy the server's fleet of ships with as few shots as possible. The game is turn-based, with each turn allowing the client to attack a specific grid coordinate.

Game Board

- The game board is a 10x10 grid.
- Columns are labeled A to J and rows 1 to 10.

Ships

The server has a fleet of 10 ships of various types:
- **Aircraft Carrier**: Occupies 4 consecutive squares.
- **Cruisers**: Each occupies 3 consecutive squares. Two cruisers in total.
- **Destroyers**: Each occupies 2 consecutive squares. Three destroyers in total.

- **Torpedo Boats**: Each occupies 1 square. Four torpedo boats in total.

Placement of Ships

- Ships must be placed either vertically or horizontally on the grid, not diagonally.
- A ship cannot overlap with another ship, extend beyond the grid boundaries, or touch the sides of another ship. However, ships may touch at their corners.
- The server places its ships on the grid at the start of the game, and their locations are hidden from the client.

Game Objective

- The client autonomously sends firing commands to specific coordinates on the server's grid.
- The server responds with the result: 'hit', 'miss', or 'sunk'.
- The client's goal is to sink the entire server's fleet with the least number of shots.

Scoring

- The game is scored based on the number of shots fired by the client.
- The game ends when the client sinks all of the server's ships.

Game Loop

1. **Initialization**:
    - The client accepts a `suffix` as a parameter for unique session identification.
    - The initial message from the client to the `/game/start` endpoint includes the client's team name.
2. **Start the Game**:
    - The server responds to the start request with a `gameId` and its team name in the response message.
3. **Gameplay**:
    - The client enters the firing phase by sending requests to `/game/fire`, targeting specific cells (e.g., B2 or J10).
    - The server processes each request and returns the outcome (`hit`, `miss`, or `sunk`) along with the status of the targeted ship.
    - The client continues gameplay, strategizing based on the server's feedback.
4. **Winning Condition and Game Completion**:
    - The client aims to sink all of the server's ships.
    - After sinking all ships, the client stops the game and sends a termination request to `/game/stop` with the `gameId`.

5. **Documentation of Results**:
    - Client records the total number of shots fired.
    - This data is written to a file named `[clientname]-[servername]-[suffix].txt`
6. **Handling Errors**:
    - If the server detects an error (e.g., invalid request), it returns an error code and terminates the game.
    - Upon receiving an error code, the client must stop.
    - If the client detects an error (e.g., unexpected server response), it sends a termination request to `/game/stop` before stopping itself.
    - Both server and client should implement sufficient logging to determine the cause of any errors, ensuring accountability and traceability.

OpenAPI Specification

The RESTful API for the naval battle game is defined `battleships.yaml`

Use `openapi-generator-maven-plugin` or online tools like https://api.openapi-generator.tech/index.html to generate model/interface.

Plugin Configuration Example

```
<plugin>
  <groupId>org.openapitools</groupId>
  <artifactId>openapi-generator-maven-plugin</artifactId>
  <version>7.2.0</version>
  <executions>
    <execution>
      <goals>
        <goal>generate</goal>
      </goals>
      <configuration>
        <inputSpec>
          ${project.basedir}/src/main/resources/battleships.yaml
        </inputSpec>
        <generatorName>spring</generatorName>
        <apiPackage>com.example.battleships.v1.api</apiPackage>
        <modelPackage>com.example.battleships.v1.model</modelPackage>
        <supportingFilesToGenerate>
          ApiUtil.java
        </supportingFilesToGenerate>
        <configOptions>
          <sourceFolder>src/gen/java/main</sourceFolder>
          <!-- for clients -->
          <!-- <interfaceOnly>true</interfaceOnly> -->
        </configOptions>
      </configuration>
```

```
      </execution>
    </executions>
  </plugin>
```

Key Components

Server Implementation

Develop the server-side application to manage game logic, including boat placement, firing mechanics, and game state management.

Client Development

Construct a client application that can place boats, send fire commands, and receive game updates.

Game Mechanics

Implement game mechanics such as initial boat placement, hit and miss logic, and tracking of remaining boats.

API Endpoints

Create RESTful endpoints for starting a game, firing at an opponent, and ending a game.

Data Persistence

Implement a method to store and retrieve game states, possibly using a database or in-memory data structures.

Steps

Develop the Server

Implement the server application following the RESTful principles outlined in the OpenAPI documentation.

Create the Client

Develop the client application ensuring it can send requests and handle responses from the server.

Implement Game Logic

Code the mechanics of the naval battle game on both the server and client sides.

Test Endpoints

Ensure all RESTful endpoints are correctly functioning using Swagger.

Integrate Client and Server

Test the interaction between client and server, ensuring seamless communication and functionality.

Test the Application

Integration Tests

Conduct tests to verify the integration between client and server components.

End-to-End Tests

Perform comprehensive end-to-end tests to ensure the entire application works as expected.

Bonus Challenges

- **Improve API**: Create v2 of battleships API.
- **Battles between teams**: Best team gets a prize.

Appendix: Project Structure Requirements

Option 1: Separate Projects for Server and Client

1. **Two Independent Projects**:
   - Create two Maven projects: one for the server and another for the client.
   - Each project should have its own `pom.xml` and source code structure.
   - This approach allows for clear separation and independent management of each application.

Option 2: Single Project with Two Executables

1. **Single Project with Multiple Executables**:
   - Alternatively, set up a single Maven project to build two separate JAR files: one for the server (`server.jar`) and one for the client (`client.jar`).
   - Configure the `maven-jar-plugin` with multiple execution phases, each targeting the respective source sets for the server and client.

Server Application Requirements

1. **Building the Server JAR**:
   - For both project structures, ensure Maven is configured to create a `server.jar` with only the server-related classes and resources.
   - Specify the `Main-Class` attribute in the `MANIFEST.MF` file for the server's main class.
2. **Running the Server**:
   - The server should run on port `8080`.
   - Ensure the server can be started with `java -jar server.jar`.

Client Application Requirements

1. **Building the Client JAR**:
   - For both setups, configure Maven to build a `client.jar` containing only the client-related classes and resources.
   - The `MANIFEST.MF` file should specify the client's main class in the `Main-Class` attribute.
2. **Running the Client**:
   - The client should be configured to run on port `8081`.
   - Execution of the client should be possible with `java -jar client.jar`.

General Guidance

- **Clear Documentation**:
- Provide explicit build and run instructions for both the server and client applications.
- Document the steps for both the separate and single project structures.
- **Testing and Validation**:
- Regardless of the chosen structure, thoroughly test both the server and client applications to confirm they operate correctly on their respective ports and contain the necessary functionalities.