Advanced OOP Project Report

Real-Time Multiplayer Jet Game using Spring Boot

Md Ali Maruf (112410221)
Maimun Hossain (112410242)
Jannatul Ferdous (112410357)
Sabbir Hossain (112310210)

Department of Computer Science

January 27, 2026

Contents

1 Introduction

This report details the design and implementation of a real-time multiplayer 2D shooter game ("JetGame"). The backend is built using **Java Spring Boot**, leveraging **WebSockets** for low-latency communication and **JPA/Hibernate** for persistent data storage.

The primary objective of this project is to demonstrate **Advanced Object-Oriented Programming (OOP)** concepts in a practical, distributed system scenario. The system handles game physics, collision detection, and state synchronization on the server side to prevent cheating and ensure consistency.

2 System Overview and Functionality

2.1 Game Features and Functionality

The "JetGame" is designed to provide a seamless competitive experience for two players. The core functional requirements and gameplay features include:

- **Real-Time Multiplayer Synchronization:** The system supports two concurrent players connected via WebSockets. Player movements and actions are synchronized in real-time (approx. 60 updates per second) to ensure a lag-free experience.
- **Combat Mechanics:** Players can fire projectiles (bullets) that travel across the game canvas. The server calculates trajectory, speed, and positioning to maintain consistency across all clients.
- **Collision Detection System:** A server-side physics engine detects intersections between entities (Bullet-to-Player). Upon collision, the system registers a "hit," updates health/score, and removes the projectile from the game state.
- **Score Tracking and Leaderboard:** The game tracks successful hits in real-time. Upon game completion (time expiry or health depletion), the final scores are persisted to a PostgreSQL database and displayed on a global leaderboard.
- **Session Management:** The application manages game sessions, handling player connection ('/join'), game start triggers, and automated game termination logic.

2.2 System Architecture

The application follows a standard **Model-View-Controller (MVC)** architecture, enhanced with a Service layer for business logic.

- **Configuration Layer:** Handles WebSocket message broker setup and endpoint registration.
- **Controller Layer:** Manages incoming HTTP requests (REST API) and WebSocket messages (STOMP protocol).
- **Service Layer:** Contains the core game loop, physics engine, and collision logic.
- **Model Layer:** Represents game entities (Players, Bullets) and database entities (Leaderboard).
- **Persistence Layer:** Uses Spring Data JPA repositories for database interactions.

3 Implementation Details

3.1 WebSocket Configuration

To enable real-time bidirectional communication, we configure the WebSocket endpoint and message broker. We use the STOMP protocol over SockJS.

```java
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.enableSimpleBroker("/topic");
        config.setApplicationDestinationPrefixes("/app");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/jet-game").withSockJS();
    }
}
```

Listing 1: WebSocketConfig.java

Explanation: This class decouples the connection logic from the business logic. By implementing the `WebSocketMessageBrokerConfigurer` interface, we override specific methods to define our messaging protocol, adhering to the Open/Closed Principle.

3.2 Game Controller and Thread Management

The Controller handles player actions and manages the game loop using a separate thread executor.

```java
@Controller
public class GameController {

    private final GameService gameService;
    private final SimpMessagingTemplate messagingTemplate;
    private ScheduledExecutorService executorService;

    @Autowired
    public GameController(GameService gameService, SimpMessagingTemplate
       messagingTemplate) {
        this.gameService = gameService;
        this.messagingTemplate = messagingTemplate;
    }

    @MessageMapping("/join")
    public void join(String playerName) {
        gameService.addPlayer(playerName);
        if (gameService.getGameState().getPlayers().size() == 2) {
            startGameLoop();
            messagingTemplate.convertAndSend("/topic/game", new GameMessage(
               MessageType.GAME_START, gameService.getGameState()));
        }
    }

    private void startGameLoop() {
        executorService = Executors.newSingleThreadScheduledExecutor();
        executorService.scheduleAtFixedRate(() -> {
            if (gameService.isGameOver()) {
                messagingTemplate.convertAndSend("/topic/game", new
                    GameMessage(MessageType.GAME_OVER, gameService.
                    getGameState()));
                executorService.shutdown();
            } else {
                gameService.update();
                messagingTemplate.convertAndSend("/topic/game", new
                    GameMessage(MessageType.SCORE_UPDATE, gameService.
                    getGameState()));
            }
        }, 0, 16, TimeUnit.MILLISECONDS);
    }
}
```

Listing 2: GameController.java (Snippet)

Explanation: The controller acts as an orchestrator. It uses Dependency Injection (via the constructor) to access the `GameService`. The `startGameLoop` method utilizes a `ScheduledExecutorServic` to run the game physics at approximately 60 frames per second (every 16ms), ensuring the main server thread remains unblocked.

3.3 Domain Models (Encapsulation)

The `Bullet` class demonstrates encapsulation. All properties are private and accessed via public getters/setters.

```java
public class Bullet {
    private double x;
    private double y;
    private double speed;
    private double width;
    private double height;
    private String shooterName;

    public Bullet(double x, double y, double speed, double width, String
        shooterName) {
        this.x = x;
        this.y = y;
        this.speed = speed;
        this.width = width;
        this.height = 5;
        this.shooterName = shooterName;
    }
}
```

Listing 3: Bullet.java

3.4 Data Persistence

The `Leaderboard` class is an Entity mapped to a database table.

```java
@Entity
@Table(name = "leaderboard")
public class Leaderboard {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "player_name")
    private String playerName;

    @Column(name = "score")
    private int score;

    @Column(name = "match_date")
    private Timestamp matchDate;

    public Leaderboard(String playerName, int score, Timestamp matchDate) {
        this.playerName = playerName;
        this.score = score;
        this.matchDate = matchDate;
    }
}
```

Listing 4: Leaderboard.java

The `LeaderboardController` exposes a REST API to retrieve these records.

```java
@RestController
@RequestMapping("/api/leaderboard")
public class LeaderboardController {
    @GetMapping
    public List<Leaderboard> getLeaderboard() {
        return leaderboardRepository.findTop10ByOrderByScoreDesc();
    }
}
```

Listing 5: LeaderboardController.java

3.5 Core Logic: Game Service

This is the most complex class, handling collision detection, power-up spawning, and state management.

```java
@Service
public class GameService {
    public static final double GAME_WIDTH = 800;

    public void update() {
        if (System.currentTimeMillis() - gameStartTime > 30000) {
            gameOver = true;
            saveGameResults();
            return;
        }

        List<Bullet> bullets = gameState.getBullets();
        List<Bullet> bulletsToRemove = new ArrayList<>();

        for (Bullet bullet : bullets) {
            bullet.setX(bullet.getX() + bullet.getSpeed());

            for (Player player : gameState.getPlayers()) {
                if (bullet.getShooterName().equals(player.getName()))
                    continue;

                if (checkCollision(bullet, player)) {
                    updateScore(bullet.getShooterName());
                    bulletsToRemove.add(bullet);
                }
            }
        }
        bullets.removeAll(bulletsToRemove);

    }

    private boolean checkCollision(Bullet b, Player p) {
        return b.getX() < p.getX() + p.getWidth() &&
               b.getX() + b.getWidth() > p.getX() &&
               b.getY() < p.getY() + p.getHeight() &&
               b.getY() + b.getHeight() > p.getY();
    }
}
```

Listing 6: GameService.java (Logic Snippet)

4 OOP Concepts Analysis

4.1 Encapsulation

The game state is heavily encapsulated. Direct access to player coordinates or scores is restricted. For instance, the `Bullet` and `Player` classes maintain their internal state (x, y coordinates) and only allow modification through specific methods (e.g., `movePlayer` in `GameService`), ensuring data integrity.

4.2 Single Responsibility Principle (SRP)

The application adheres to SRP by separating concerns:

- **GameService:** Handles physics and rules. It does not know about HTTP or WebSockets.
- **GameController:** Handles communication. It does not calculate physics.
- **LeaderboardRepository:** Handles database SQL operations.

4.3 Dependency Injection (DI)

Spring's Inversion of Control (IoC) container is used throughout. For example, `GameController` requests a `GameService` in its constructor. This makes the code testable and loosely coupled; we could easily swap `GameService` for a `MockGameService` during unit testing without changing the controller code.

4.4 Concurrency

While not strictly a standard OOP concept, the object-oriented handling of threads via `ScheduledExecutor` allows us to treat the game loop as an object-managed task, rather than raw procedural thread management.

5 Conclusion

This project successfully implements a real-time multiplayer game using Spring Boot. By adhering to Object-Oriented principles and the MVC design pattern, the system is modular, scalable, and easy to maintain. The separation of the communication layer (WebSockets) from the physics engine (Service layer) demonstrates a robust architectural approach suitable for complex interactive applications.