Joc Table Online

Documentație proiect Inginerie Software

Echipă: Cîrneală Darius, Botărel Patrik, Mărginean Alexandru
An/Grupă: *Anul 3, grupa 7*
Disciplina: *Inginerie Software*

Cuprins

1 Scurtă descriere a proiectului

Proiectul vizează dezvoltarea unei platforme web interactive pentru jocul de table. Utilizatorii pot crea conturi, se pot alătura unor camere de joc și pot juca împotriva altor utilizatori în timp real. Logica jocului este validată pe server, iar comunicarea se bazează pe protocolul WebSocket pentru a asigura o experiență de joc dinamică.

1.1 Obiective

- Implementarea unui sistem de autentificare securizat.
- Gestiunea meciurilor în timp real (aruncarea zarurilor, mutarea pieselor, capturarea pieselor).
- Persistența datelor referitoare la utilizatori și istoricul meciurilor.
- Interfață grafică responsivă adaptată pentru jocul de table.

2 Limbaje și tehnologii folosite

- **Backend**: Java Spring Boot (Spring MVC, Spring Data JPA, Spring Security).
- **Frontend**: Thymeleaf, HTML5, CSS3, JavaScript.
- **Real-time**: WebSockets (Infrastructură custom bazată pe `TextWebSocketHandler`).
- **Bază de date**: SQL (MySQL/PostgreSQL).
- **Gestiune Proiect**: Maven, Git.

3 Cerințe funcționale + diagrama Use-Case

3.1 Cerințe funcționale

1. **Autentificare și Profil**: Utilizatorii se pot înregistra și loga.
2. **Matchmaking**: Posibilitatea de a vedea camerele disponibile și de a intra în joc.
3. **Mecanica de joc**: Aruncarea automată a zarurilor, evidențierea mutărilor posibile, logica de final de joc.
4. **Dashboard Utilizator**: Vizualizarea statisticilor personale și a poziției în clasament direct în Lobby.
5. **Istoric Meciuri**: Afișarea listei cronologice a meciurilor jucate, incluzând oponentul, scorul și rezultatul (Victorie/Înfrângere).

3.2 Diagrama Use-Case

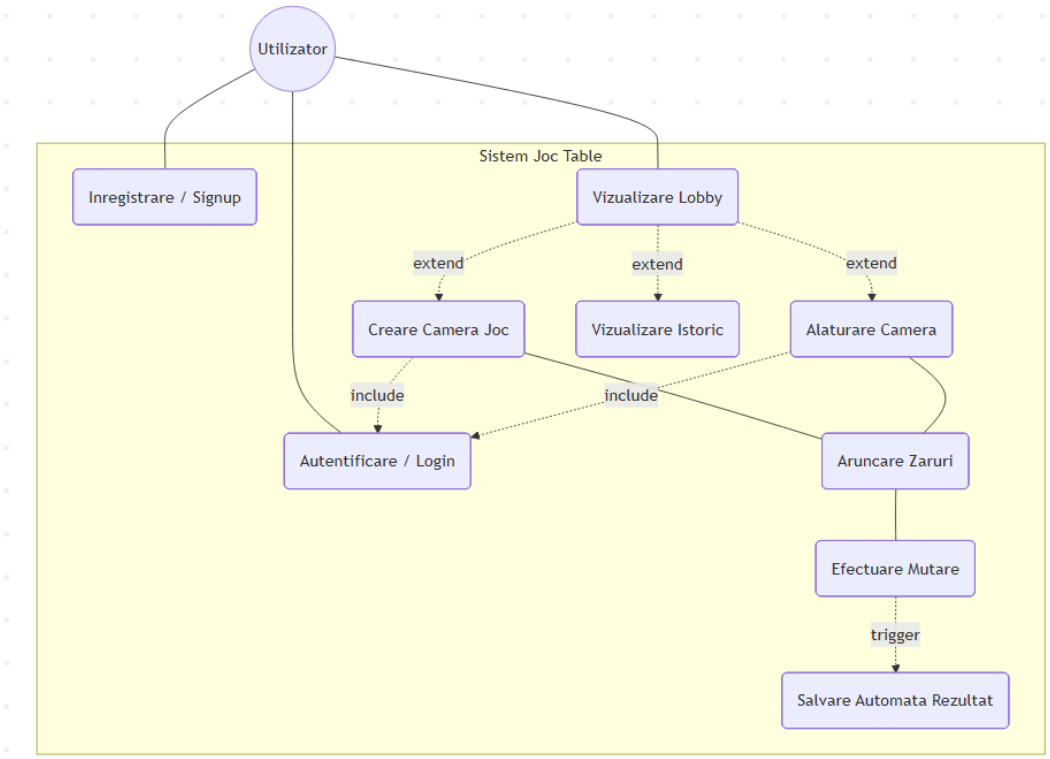

Figura 1: Diagrama Use-Case — Arena Table

4 Cerințe non-funcționale

- **Performanță**: Actualizarea tablei de joc trebuie să aibă o latență de sub 200ms.
- **Securitate**: Prevenirea mutărilor ilegale prin validarea acestora exclusiv pe backend.
- **Usability**: Interfața trebuie să permită drag-and-drop pentru mutarea pieselor.

5 Diagrame

5.1 Cîrneală Darius — Infrastructură WebSocket și Logica Jocului

Arhitectura WebSocket: Sistemul de comunicare utilizează o ierarhie de clase care decuplează managementul conexiunilor de logica aplicației. `BaseWebSocketHandler` gestionează obiectele de tip `Channel` și `Client`, delegând procesarea mesajelor JSON către interfața `BaseWebSocketListener`.

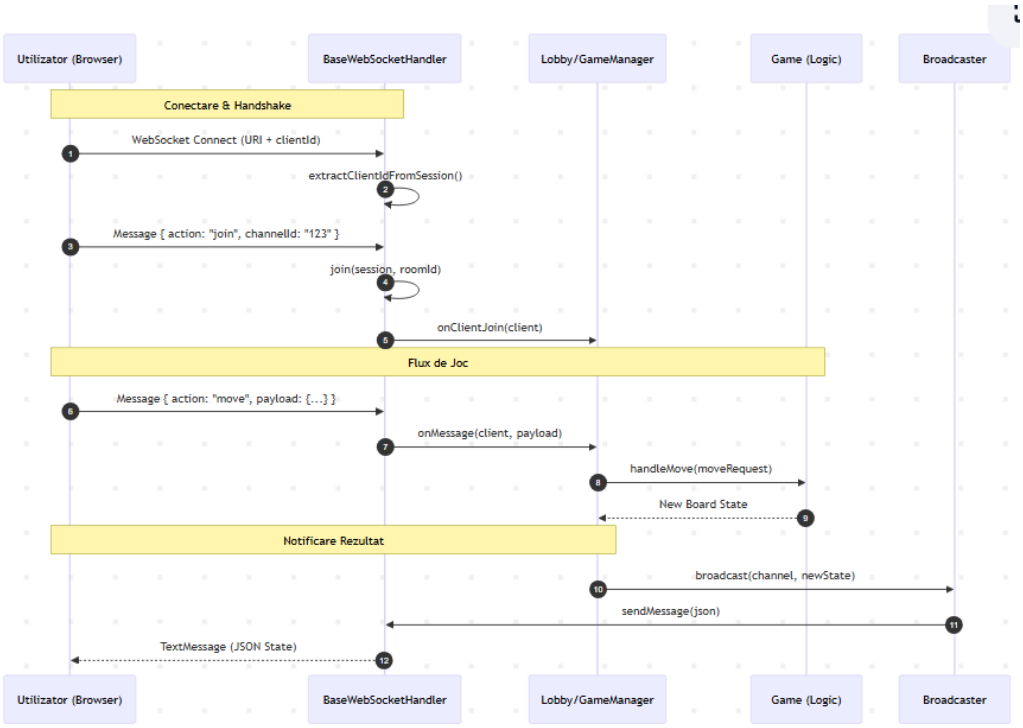

Figura 2: Diagramă de Clase — Infrastructura WebSocket

Procesul de Comunicare: Diagrama de secvență tehnică ilustrează cum mesajele de tip "move" sau "join" sunt extrase din sesiunea WebSocket, validate de handler și trimise către managerul de joc pentru procesare.

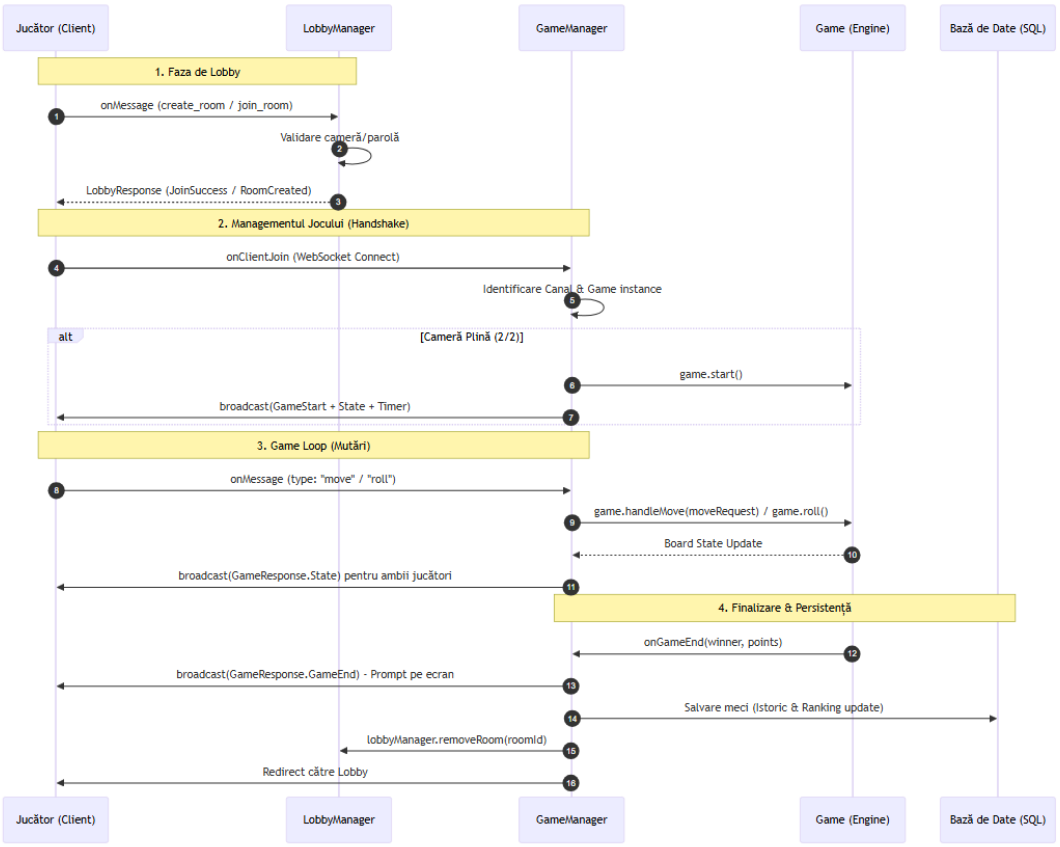

Figura 3: Diagramă de Secvență — Fluxul de mesaje WebSocket și Lobby

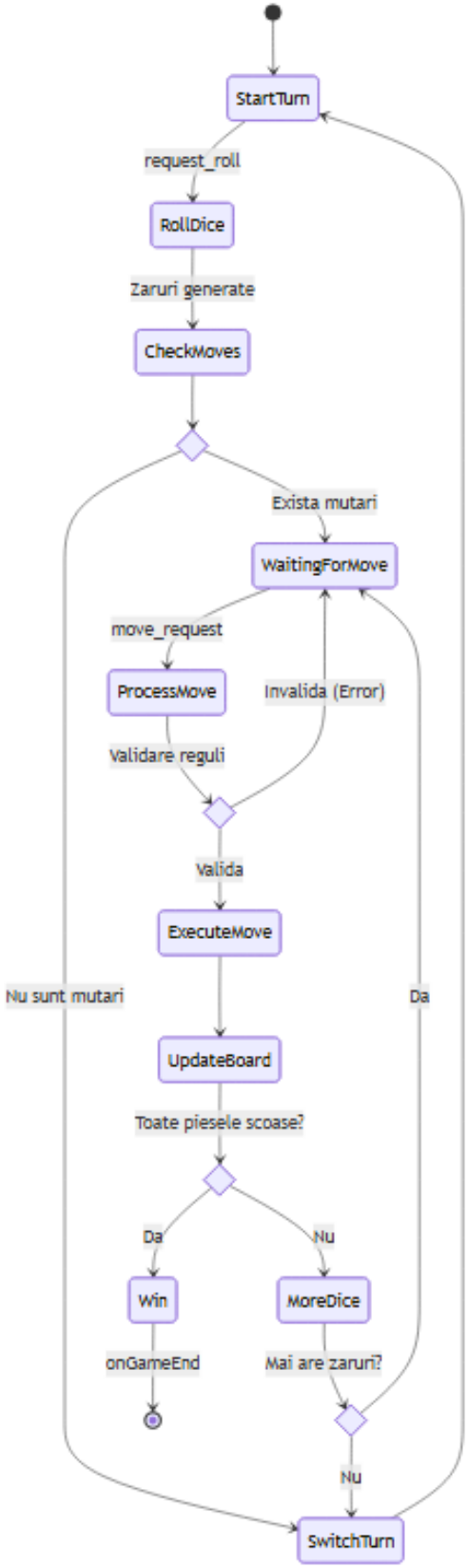

Figura 4: Diagramă de Activitate — Logica procesării unui turn de joc

5.2 Botărel Patrik — Persistență SQL și Servicii Backend

Modelul de Date: Pentru persistența datelor s-a utilizat un sistem de tabele SQL care urmărește profilurile utilizatorilor, detaliile fiecărei partide (scoruri, câștigători) și un clasament calculat în timp real.

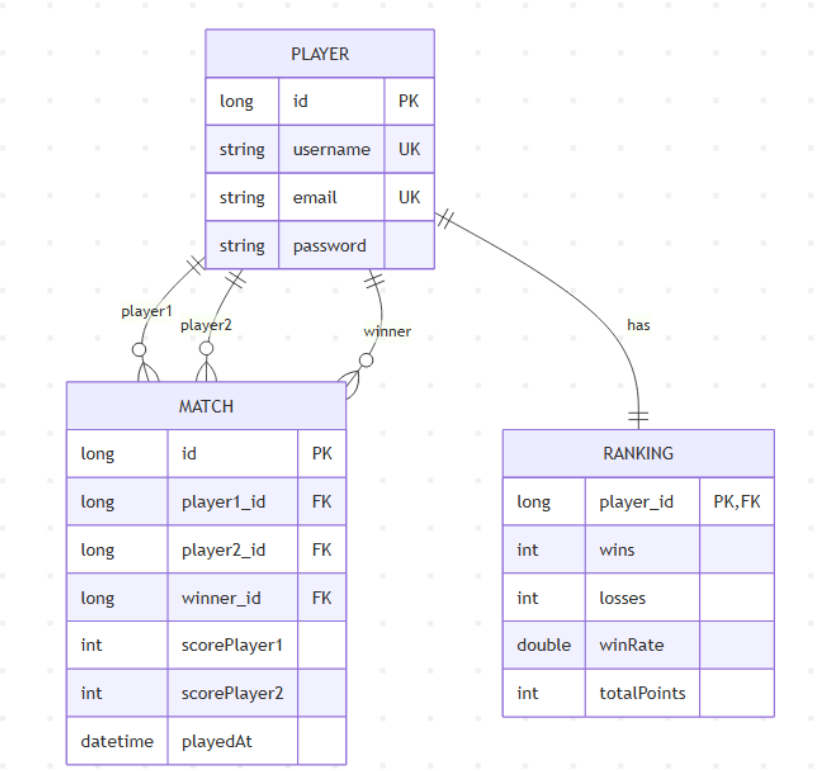

Figura 5: Diagramă Entity-Relationship (ERD)

Logica Serviciilor: Diagrama de clase pentru servicii ilustrează modul în care `MatchService` și `RankingService` utilizează pattern-ul Repository pentru a interacționa cu baza de date după finalizarea fiecărui eveniment de joc.

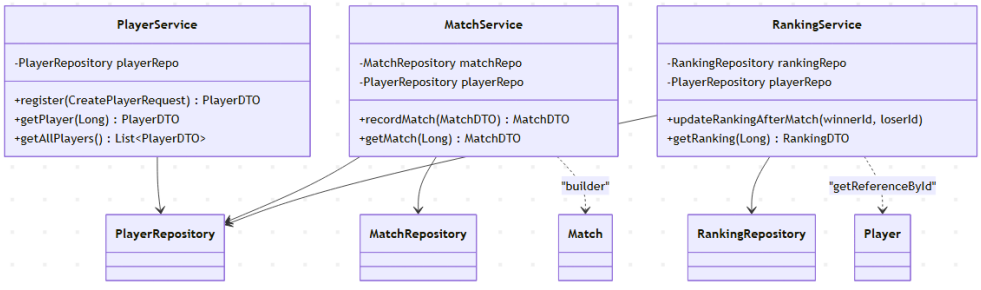

Figura 6: Diagramă de Clase pentru Servicii și Repositories

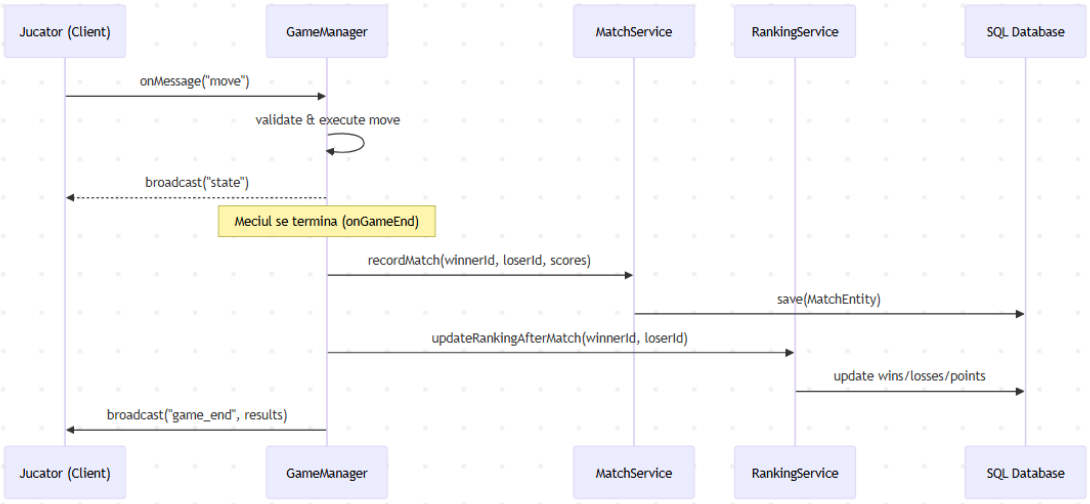

Figura 7: Diagramă de Secvență — Fluxul de Joc și Persistența Datelor

5.3 Mărginean Alexandru — Sistemul de Autentificare și Managementul Sesiunilor

Logica de Autentificare (Backend): Am implementat `AuthController`, care gestionează fluxurile de securitate ale aplicației. Sistemul permite înregistrarea utilizatorilor noi prin `signup` și validarea identității acestora prin `login`. Parolele și datele de profil sunt gestionate prin `PlayerRepository`, asigurând o integrare directă cu baza de date MySQL.

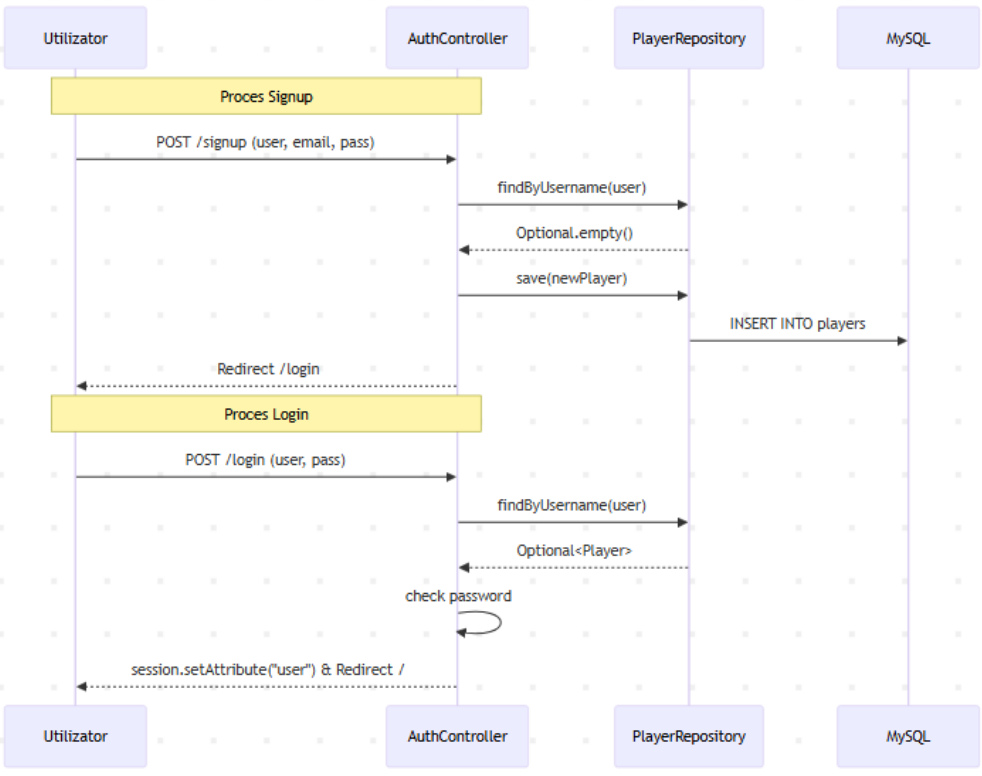

Figura 8: Diagramă de Secvență — Procesul de Înregistrare și Autentificare

Managementul Sesiunilor: Pentru a menține starea utilizatorului între pagini (Lobby, Game

Board), am utilizat `HttpSession`. La o autentificare reușită, obiectul `Player` este stocat în sesiune, permițând celorlalte componente (precum WebSockets) să identifice în mod unic utilizatorul curent. Procesul de `logout` asigură invalidarea sesiunii și curățarea datelor sensibile.

Validarea Datelor și Feedback: Am implementat mecanisme de validare pentru a preveni duplicarea numelor de utilizator. În cazul unor erori (parolă incorectă sau utilizator existent), controller-ul transmite mesaje de feedback către interfața grafică prin intermediul modelului `Thymeleaf`, oferind o experiență de utilizare fluidă.

Securitatea la nivel de Controller:

- **Signup**: Utilizează pattern-ul `Builder` pentru a crea entitatea `Player` înainte de salvare.
- **Login**: Verifică existența utilizatorului și potrivirea credențialelor înainte de a redirecționa către pagina principală.
- **Logout**: Dezalocă resursele sesiunii pentru a preveni accesul neautorizat.

Experiența Utilizatorului (UX): Am implementat stiluri CSS condiționale pentru a diferenția vizual rezultatele partidelor (victorii vs. înfrângeri) și am utilizat `Thymeleaf` pentru a randa dinamic aceste date, oferind utilizatorului un feedback instantaneu asupra progresului său în joc.

6 Design Patterns

6.1 Observer / Listener Pattern

Implementat prin interfața `BaseWebSocketListener`. Componenta `BaseWebSocketHandler` acționează ca subiectul care notifică managerii de joc sau de lobby la primirea mesajelor de la clienți, decuplând infrastructura de rețea de logica de business.

6.2 Strategy Pattern

Folosit pentru gestionarea diverselor tipuri de acțiuni primite prin socket (`ROLL`, `MOVE`, `REMOVE`). Sistemul alege și execută logica specifică în funcție de tipul mesajului JSON, fără a folosi structuri `if-else` repetitive și greu de întreținut.

6.3 Spring Singleton Scope

Deși nu este un Singleton clasic, am utilizat mecanismul de **Dependency Injection** din Spring Boot. Clasele adnotate cu `@Service` (ex: `GameManager`, `LobbyManager`) sunt gestionate ca instanțe unice (Singleton Bean Scope). Acest lucru asigură că starea camerelor de joc și a jucătorilor conectați este partajată corect între toate cererile WebSocket.

6.4 Builder Pattern

Implementat prin adnotarea `@Builder` din librăria Lombok pe entitățile de bază de date (`Match`, `Player`). Acest pattern facilitează crearea obiectelor complexe într-o manieră lizibilă, esențială pentru procesul de salvare a rezultatelor meciurilor în `MatchService`.

7 Scurt README

7.1 Structură proiect (Pachete)

Organizarea codului respectă structura standard Spring Boot:

- `com.example.proiectis.controller` — Endpoint-uri REST pentru interfața web.
- `com.example.proiectis.dto` — Obiecte pentru transferul de date între straturi.

- `com.example.proiectis.entity` — Modelele de date pentru JPA (Match, Player, Ranking).
- `com.example.proiectis.game` — Engine-ul jocului și managerii de sesiune (`GameManager`, `LobbyManager`).
- `com.example.proiectis.repository` — Interfețe pentru accesul la baza de date.
- `com.example.proiectis.service` — Logica de business (MatchService, PlayerService, RankingService).
- `com.example.proiectis.websocket` — Managementul conexiunilor în timp real.

7.2 Rulare

1. Configurați baza de date în `application.properties`.
2. Rulați `mvn spring-boot:run`.
3. Accesați aplicația în browser pe portul 8080.

8 Specificația Protocolului WebSocket

Comunicarea între client și server se realizează prin mesaje asincrone în format JSON. Toate mesajele urmează o structură standard definită de clasa `Response<T>`, asigurând un protocol predictibil pentru ambele părți.

8.1 Structura Generală a Mesajelor

Orice pachet de date (atât cerere, cât și răspuns) este compus din două câmpuri principale:

- **type**: Un șir de caractere care identifică tipul acțiunii sau al evenimentului.
- **payload**: Un obiect generic care conține datele specifice evenimentului respectiv.

8.2 Mesaje de la Client (Requests)

Clienții trimit mesaje către server pentru a interacționa cu mediul de joc. Acțiunile sunt procesate de `BaseWebSocketHandler`.

Acțiune (Type)	**Sursă**	**Descriere Payload**
`create_room`	Lobby	Conține opțional parola camerei (`password`).
`join_room`	Lobby	Conține `roomId` și parola pentru acces.
`roll_request`	Game	Nu necesită payload; solicită generarea zarurilor.
`move`	Game	Conține `from`, `to` și `color` (ID-ul piesei).
`reenter`	Game	Folosit pentru reintroducerea pieselor lovite pe tablă.

Tabela 1: Principalele cereri trimise de utilizator

8.3 Mesaje de la Server (Responses)

Serverul notifică clienții despre schimbările de stare prin transmisiuni de tip *Unicast* sau *Broadcast*.

- **state**: Trimis după fiecare mutare validă. Conține întreaga structură a tablei de table serializată (poziția pieselor, zarurile rămase).

- **timer**: Actualizare periodică (tick) a timpului rămas pentru fiecare jucător.
- **game_end**: Trimis la detectarea unui câștigător. Conține statisticile finale: `winner`, `points`, `username`.
- **invalid_move**: Trimis în caz de eroare. Conține un câmp `reason` (ex: "Not your turn" sau "Illegal move").

8.4 Exemplu de Mesaj JSON

Mai jos este prezentat un exemplu de mesaj de tip `state`, generat de server pentru a sincroniza tabla de joc:

```
{
 "type": "state",
 "payload": {
  "board": [ [2, 0], [0, 0], ... ],
  "dice": [4, 2],
  "currentTurn": 1,
  "whiteCaptured": 0,
  "blackCaptured": 1
 }
}
```

Figura 9: Vizualizarea ierarhică a protocolului de comunicare