Final Report:
MapForger

Collaborative DnD Map Visualizer — Distributed Systems Course 2025–2026

Eduard Toni Alexandru
eduardtoni.alexandru@studio.unibo.it

July 17, 2026

MapForger is a distributed microservice-based platform designed for real-time 2D tabletop role-playing game (RPG) management. The system is architected to handle complex campaign logic, character persistence, and live session synchronization across multiple players. Unlike a monolithic game server, the application logic is decomposed into specialized services—**Core** and **Gameplay**—coordinated via **Spring Cloud Eureka** for service discovery and routed through a **Spring Cloud Gateway**.

The project focuses on ensuring a consistent and resilient experience during gameplay sessions. The **CAP Theorem** trade-off is explicitly addressed: the system prioritizes **Consistency** over Availability within a single session, ensuring that no two actors can occupy the same map tile and that every action is atomically persisted before being broadcast. **Fault tolerance** is implemented through persistent state: every actor position, attack, and death event is recorded as an immutable row in **PostgreSQL 17**. Upon reconnection after a crash, the client re-fetches the full game state from the database, enabling seamless session recovery without data loss. Inter-service communication is handled via **OpenFeign** clients registered through **Eureka**, allowing the `Core` service to retrieve live session data from `Gameplay` and vice versa, while keeping each service's database fully isolated.

Contents

1 Concept

The project is a web-based distributed application consisting of a frontend developed in Angular and a backend organized into a microservice architecture. The backend logic is decomposed into two specialized services—`Core` and `Gameplay`—orchestrated by Spring Cloud Eureka for service discovery and a Spring Cloud Gateway for centralized routing.

Use Case Description

The system facilitates real-time Tabletop Role-Playing Game (TTRPG) sessions.

- **What:** The software allows users to create campaigns, manage character sheets, and participate in live, turn-based game sessions on a synchronized 2D grid map rendered in the browser.
- **Roles:** There are two primary roles: the **Game Master (GM)**, who creates and controls the campaign environment and governs NPC actors, and the **Players**, who each control one individual token and interact with the game logic during their assigned turn.
- **Interaction:** Users interact via a web browser. Gameplay interaction is real-time; a player moving a token or performing an attack sends a STOMP message to the backend via a persistent WebSocket connection, which is then validated, persisted, and broadcast to all other participants in the same game room.
- **Data Storage:** The system persists user profiles, campaign metadata, and character definitions in the `Core` PostgreSQL database. Every gameplay event (movement, attack, death) is stored as an immutable record in the `Gameplay` PostgreSQL database, forming a complete history of the session.

Why is distribution needed?

While a monolithic approach is technically feasible for a small game, distribution was deliberately chosen for the following reasons:

- **Fault Isolation:** By separating the `Gameplay` logic from the `Core` service, a crash or overload in the real-time WebSocket engine does not prevent users from browsing their campaign dashboard, managing characters, or logging in.
- **State Recovery:** Because the game state is fully persisted in the `Gameplay` PostgreSQL database, a service restart does not result in data loss. Reconnecting clients simply re-fetch the current state via HTTP before re-establishing the WebSocket channel.
- **Independent Scalability:** The `Gameplay` service—which bears the heavy cost of persistent WebSocket connections—can be scaled independently from `Core`, which handles comparatively infrequent REST requests.

- **Educational Purpose:** To implement and analyze real microservice patterns such as Service Discovery, API Gateway routing, inter-service Feign calls, schema migration with Liquibase, and CAP theorem trade-offs in a stateful real-time system.

2 Requirements Elicitation and Analysis

2.1 Functional Requirements

- **Registration & Authentication:** Users must be able to create an account, log in securely (JWT-based), and log out. Credentials are stored hashed with BCrypt.
- **Authorization:** The system must distinguish between a Game Master (campaign creator) and a Player, restricting game-start, pause, and NPC-control actions to the GM.
- **CRUD Character:** Users must be able to create, view, update, and delete character sheets, each with attributes such as armor, speed, and weapon damage.
- **CRUD Campaign:** GMs must be able to create and manage campaigns, including associating a map, a description, and a cover image. Campaign creation automatically seeds default NPC actors.
- **Campaign Joining:** Players must be able to join a campaign by selecting a character, which creates both a `CampaignMember` and a `CampaignActor` record in `Core`.
- **Room Presence:** Users must be able to join a live lobby linked to a campaign. The system must track which users are currently online in the lobby and broadcast presence updates to all members via WebSocket.
- **Session Lifecycle:** The GM must be able to start, pause, resume, and permanently finish a game session. Players receive these state transitions in real-time via WebSocket.
- **Turn-Based Gameplay:** The system must enforce a strict turn order. Only the actor whose turn it is may move or attack. After acting, the player may end their turn, advancing the sequence to the next actor.
- **Movement Validation:** The system must validate movement actions server-side, rejecting moves that exceed the actor's speed (Chebyshev distance), fall outside the map bounds, or target an already-occupied cell.
- **Combat:** Players must be able to attack another actor during their turn. The system must compute damage based on the attacker's `weaponDamage` stat and reduce the target's HP accordingly.

- **Death and Spectator Mode:** When an actor's HP reaches zero, a `DeathAction` is persisted, the actor is removed from the turn order, and all clients are notified. The controlling player transitions to a spectator view.
- **Campaign History:** The campaign detail page must display a chronological timeline of all turns and actions (moves, attacks, deaths) from completed or active sessions.
- **Session Persistence:** A paused session must be resumable in the future from the exact state it was left in—no data is lost between sessions.

2.2 Non-functional Requirements

1. **Consistency:** The system must maintain a consistent state of the game map across all connected clients.
   - *Acceptance Criterion:* If a player attempts an invalid action (e.g., moving to an occupied tile), the server must reject the request with an `IllegalArgumentException` and the client's board must not update.
2. **Availability:** The failure of the `Gameplay` microservice must not prevent users from accessing their profile, characters, or campaign data.
   - *Acceptance Criterion:* When the `Gameplay` service is manually shut down, the `Core` service remains reachable via the Gateway. The `Core` service uses a Feign fallback to return `null` gracefully when Gameplay is unavailable, rather than propagating a 503 error to the client.
3. **Low Latency Synchronization:** Real-time actions must be propagated to other players with minimal delay.
   - *Acceptance Criterion:* The round-trip time between a WebSocket move event being sent by one client and the corresponding actor position update being rendered on another client's Phaser board must not exceed 200ms under normal local network conditions.
4. **Recoverability:** The system must be able to restore the full state of a game session after a service crash or client reconnection.
   - *Acceptance Criterion:* Upon restarting the `Gameplay` service and reconnecting, the 2D map must display all actors in their last recorded positions, as read from the persisted `campaign_actors` table. No manual intervention is required.
5. **Schema Consistency:** Database schema changes must be applied automatically and consistently across all environments.
   - *Acceptance Criterion:* On startup, each microservice applies any pending Liquibase changesets before accepting traffic, ensuring the running code and the database schema are always in sync.

2.3 Implementation Choices

1. **Technical Stack:** The backend is developed using Java 21 and Spring Boot 3.5.
   - *Justification:* Java 21 is the current Long-Term Support release and provides mature, battle-tested libraries for building distributed systems. Spring Boot 3's auto-configuration dramatically reduces boilerplate for REST endpoints, security, JPA, and service discovery. The Spring Cloud ecosystem (Eureka, Gateway, OpenFeign) is the de facto standard for microservice infrastructure on the JVM, offering tight integration and production-grade reliability.

2. **Database:** PostgreSQL 17 is used for data persistence, with two separate database instances—`MapForge` (Core) and `MapForgeGameplay` (Gameplay)—running in the same Docker container.
   - *Justification:* PostgreSQL offers strong ACID guarantees, which are necessary to enforce turn-based game logic atomically. Its robust support for UUID primary keys (native `gen_random_uuid()`), enum types, and composite primary keys matches the domain model precisely. Separating the two databases provides true data isolation between services without requiring separate infrastructure during development.

3. **Service Coordination:** Spring Cloud Eureka and Spring Cloud Gateway are used for infrastructure management, with OpenFeign for inter-service HTTP calls.
   - *Justification:* Eureka provides client-side service discovery, allowing services to locate each other by logical name rather than hardcoded IP addresses. The Gateway provides a single entry point for the frontend, handling CORS and routing transparently. OpenFeign generates HTTP clients from annotated interfaces, making cross-service calls as simple as local method invocations while benefiting from Eureka's load-balancing registry.

4. **Real-Time Communication:** WebSockets with the STOMP sub-protocol are used for all live gameplay communication.
   - *Justification:* STOMP over WebSocket provides a publish-subscribe messaging model that maps naturally onto the game's room concept: each campaign is a topic, and all players in the room subscribe to it. This eliminates the overhead of HTTP polling and allows the server to push state changes (turn transitions, actor movements, deaths) to all clients simultaneously.

5. **Schema Migration:** Liquibase is used for database schema versioning in both microservices.
   - *Justification:* In a distributed system where multiple services own separate databases, schema drift between environments is a critical risk. Liquibase's changelog files act as version control for the database, ensuring that every environment—development, Docker, CI—runs exactly the same schema. Each new feature that requires a schema change is expressed as a new, immutable SQL changeset.

6. **2D Map Rendering:** Phaser.js is used for the client-side game board.
    - *Justification:* Phaser is a mature, purpose-built 2D game framework for the browser. It provides a scene graph, input handling, camera zoom/pan, and tween animations out of the box—features that would require significant custom code with plain Canvas. Its dynamic import capability integrates cleanly with Angular's SSR architecture, ensuring the game engine only initializes in the browser, never on the server.

2.4 Relevant Distributed System Features

- **Fault Tolerance and Dependability:** This is a core focus of the project. The failure of the `Gameplay` node is isolated from `Core` by design. Because every gameplay action (movement, attack, death) is persisted in PostgreSQL before being broadcast, no in-flight state is ever lost. When a client reconnects after a crash, it calls `GET /campaigns/{id}/state` and `GET /campaigns/{id}/actors` to reconstruct the board from the database record, without requiring the previous server's memory. The `Core` service uses a Feign circuit-breaker fallback so that Gameplay being offline degrades gracefully rather than causing cascading failures.

- **Scalability:** The architecture is designed for horizontal scalability. The functional decomposition (`Core` vs. `Gameplay`) allows the Gameplay service—which handles expensive, persistent WebSocket connections—to be scaled into multiple instances independently. Eureka's registry and the Gateway's load balancer distribute incoming connections across all healthy instances automatically.

- **Resource Sharing and Concurrency:** The shared resource is the Game Room's map state. Multiple users interact concurrently with the same set of `CampaignActor` records. The turn-based protocol itself is the primary concurrency control mechanism: at any moment, only one actor is authorized to act, and the server validates the `actorId` in every action payload against the current turn record before processing it.

- **Performance:** WebSockets eliminate the polling overhead of HTTP for real-time updates. The `PresenceService` uses a `ConcurrentHashMap` for O(1) player lookups. HikariCP connection pooling minimizes per-request database overhead, which is critical for the high-frequency writes generated during an active game session.

- **Transparency:** Location transparency is fully realized via Spring Cloud Gateway and Eureka. All client requests are directed to a single entry point (`localhost:8080`), and the routing to `Core` or `Gameplay` is completely invisible to the Angular frontend. Inter-service Feign calls also use logical service names (`mapforge-core`, `mapforge-gameplay`) resolved through Eureka, not hardcoded hostnames.

- **Security and Trust:** Authorization is enforced at multiple levels. The Gateway validates JWT tokens before forwarding any request. Individual endpoints are

protected by Spring Security. The game logic enforces role boundaries: only the GM can start, pause, and finish a session, and only the actor whose turn it currently is can perform gameplay actions.

3 Design

3.1 Architecture

The system uses a **Microservices** architecture decomposed along functional boundaries. After analysis, three macro concerns were identified:

- **CRUD and Profile Logic:** Performed by the `Core` microservice. Handles users, characters, NPCs, campaigns, maps, and campaign membership. Exposes a REST API.
- **Real-Time Gameplay Logic:** Performed by the `Gameplay` microservice. Handles game sessions, turns, actor state, and all action events. Exposes both a REST API (for state queries) and a WebSocket endpoint (for real-time events).
- **Infrastructure:** Handled by `mapforge-eureka` (service registry) and `mapforge-gateway` (edge proxy).

This **Functional Decomposition** ensures that the volatile, high-traffic `Gameplay` service and the stable `Core` service can evolve, be deployed, and fail independently.

3.2 Infrastructure

The infrastructure is composed of several decoupled components:

- **Client-side:** An **Angular 19 SPA** providing the graphical interface. It communicates with the Gateway via REST (HTTP) for data fetching and via STOMP/WebSocket for real-time gameplay events.
- **Edge Layer:** A **Spring Cloud Gateway** instance acting as a reverse proxy. It provides a single entry point (`:8080`), hides the internal network topology, applies CORS policies globally, and strips service-name prefixes from incoming URLs before forwarding.
- **Discovery Layer:** A **Netflix Eureka** server (`:8761`). It maintains a live registry of all active service instances, enabling the Gateway and Feign clients to resolve logical service names to actual IP addresses.
- **Service Layer:** Two independent Spring Boot applications (`Core` on `:8082`, `Gameplay` on `:8081`). Both are stateless in their HTTP communication and register themselves with Eureka on startup.

- **Persistence Layer:** Two **PostgreSQL 17** Docker containers each one hosting a database: `MapForge` (owned by `Core`) and `MapForgeGameplay` (owned by `Gameplay`). Each service connects only to its own database.

Services discover each other through Eureka's registry using logical names, making the infrastructure **Location Transparent**. For development, all components run on the same host but on distinct ports. The system is designed to be **Cloud-Ready**: each component could be deployed on separate machines, and the system would continue to function as long as all services can reach the Eureka server.

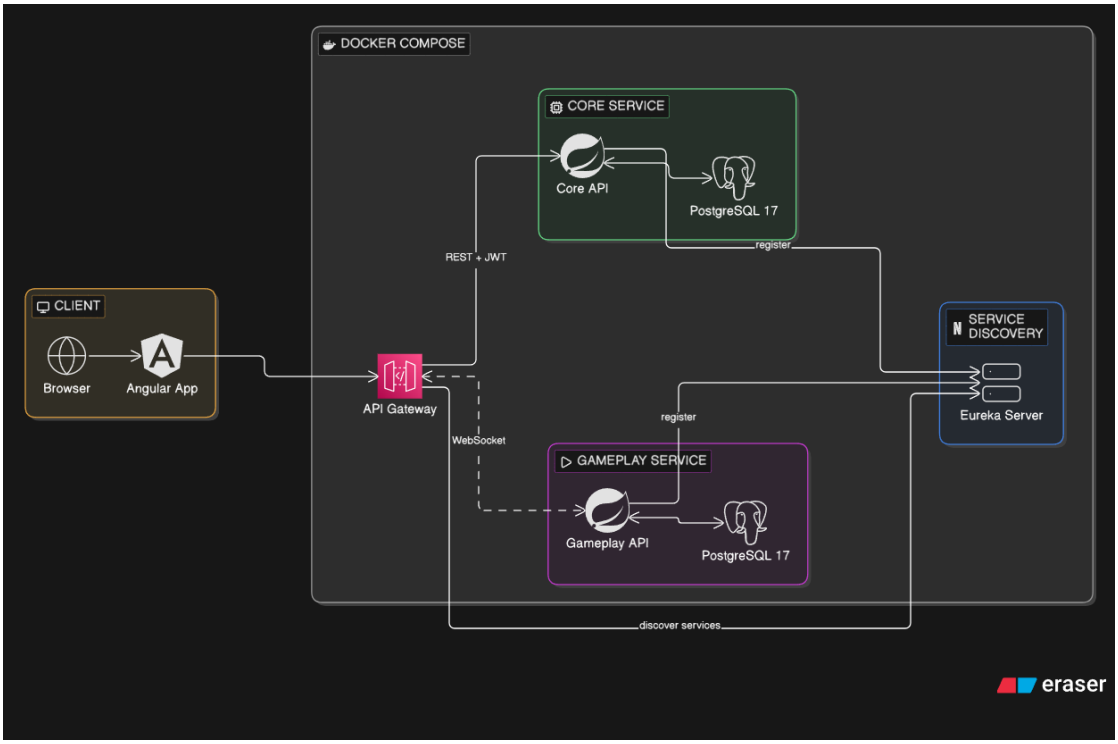

Figure 1: Component diagram of the infrastructure.

3.3 Modelling

3.3.1 Domain Entities

The domain is partitioned between the two services:

Core Service (MapForge DB)

- **User:** Represents the identity, credentials, and profile of a participant. Contains username, email, hashed password, and date of birth.

- **Character:** Defines the base attributes of a hero (name, armor, speed, weapon damage, race, alignment, backstory). Independent of any specific game session—a character is a template that can participate in multiple campaigns.
- **NPC:** A non-player entity (monster, ally) with combat stats (armor, speed, weapon damage, type). NPCs are seeded automatically when a campaign is created.
- **Map:** Defines the grid environment for a campaign (width, height, name, image reference).
- **Campaign:** The top-level container linking a GM, a Map, and a set of Members. Identified by a UUID primary key.
- **CampaignMember:** A junction entity (composite PK: owner ID + campaign UUID) that enrolls a User into a Campaign with a role (`MASTER` or `PLAYER`).
- **CampaignActor (Core copy):** Records which Character or NPC a Member brings to a campaign. Stores the *template* stats at the time of joining—this is the source of truth for bootstrapping a game session.

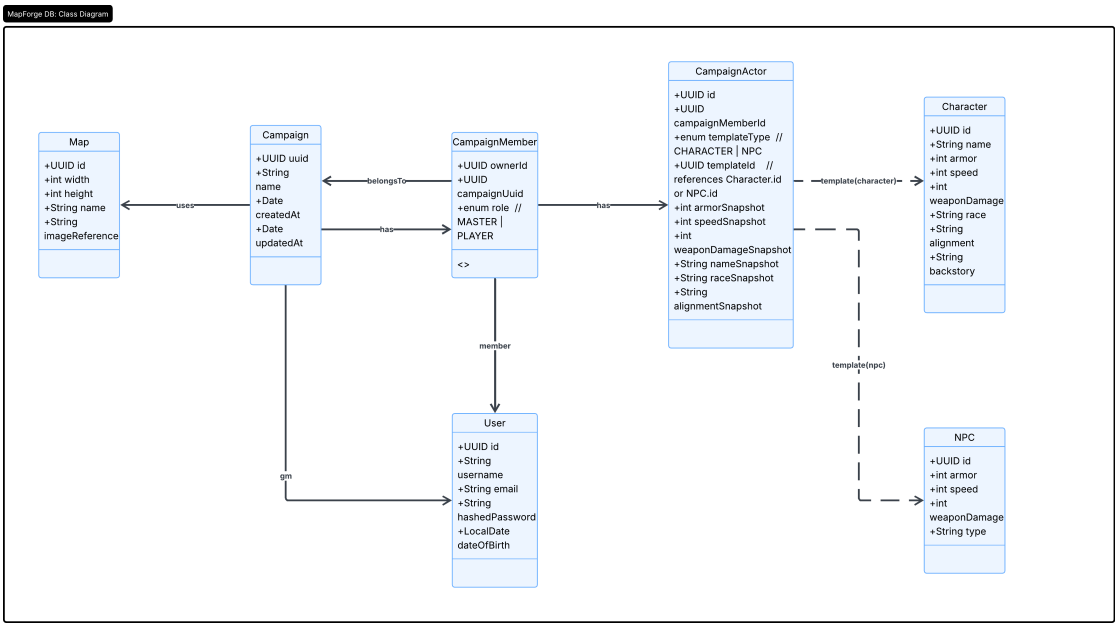

Figure 2: Core DB class diagram

Gameplay Service (MapForgeGameplay DB)

- **CampaignActor (Gameplay copy):** The *live runtime instance* of a character in an active session. Copied from Core at session start. Tracks mutable state:

current HP, XP, weapon damage, speed, and spatial coordinates (x, y) on the map grid.

- **GameSession:** Represents the state of a single play session for a campaign. Contains the current status (`WAITING`, `ACTIVE`, `PAUSED`, `FINISHED`), the current turn index, and the ordered turn list.
- **Turn:** A structural record (composite PK: index + campaign UUID) tracking which `CampaignActor` currently has priority to act.
- **MovementAction:** An immutable record that a `CampaignActor` moved to a specific (x, y) coordinate during a given turn.
- **AttackAction:** An immutable record of one actor attacking another, including the damage dealt.
- **DeathAction:** A state-change event indicating that an actor reached zero HP and was removed from the turn order. Records the killer and the turn index.
- **GameSessionTurnOrder:** A ordered collection table linked to `GameSession` that persists the turn order as an indexed list of `CampaignActor` IDs. Each row stores a `campaign_id`, an `actor_id`, and a `position` index that determines the sequence in which actors take their turns.
- **SessionPresence:** Records which users are currently active in a campaign's lobby or game room. Each row stores a `campaign_id`, a `username`, and a `joined_at` timestamp.

3.3.2 The CampaignActor Bridge

`CampaignActor` is the only entity that exists in both databases, with different purposes:

- **Core's CampaignActor:** Immutable after creation. Answers "who is in this campaign with which character/NPC?" It is the canonical enrollment record.
- **Gameplay's CampaignActor:** Mutable during a session. Answers "where is this actor now, and how much HP does it have?" It is bootstrapped from Core's copy at `startGame()` and is then fully owned by the Gameplay service.

This separation means that if a player updates their character's stats between sessions, the live session is unaffected—the session uses the snapshot taken at start time, which is the correct behavior for a game.

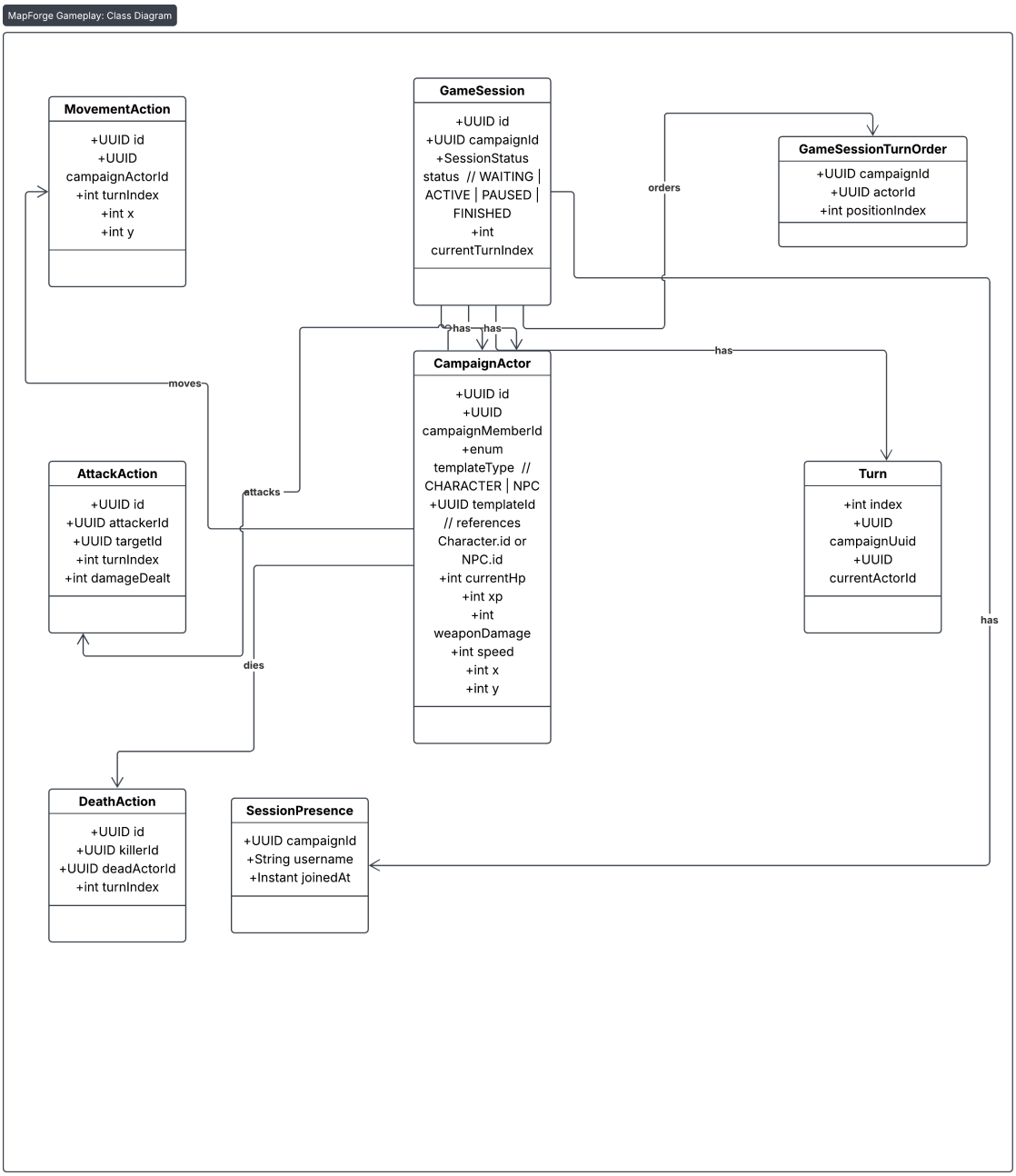

Figure 3: Gameplay DB class diagram

3.3.3 Session Lifecycle State Machine

A `GameSession` transitions through the following states:

$$\texttt{WAITING} \xrightarrow{\text{GM starts}} \texttt{ACTIVE} \xrightarrow{\text{GM pauses}} \texttt{PAUSED} \xrightarrow{\text{GM resumes}} \texttt{ACTIVE} \xrightarrow{\text{GM finishes}} \texttt{FINISHED}$$

A paused session retains all actor positions, HP values, turn indices, and action history. Resuming simply sets the status back to `ACTIVE`—no data needs to be replayed or reconstructed.

3.4 Interaction

The system uses two distinct communication patterns depending on the nature of the operation:

REST (HTTP/1.1): Used for all synchronous, stateless operations: authentication, character/campaign CRUD, initial state hydration on page load, and history queries. The frontend calls the Gateway, which routes to the appropriate service.

STOMP over WebSocket: Used for all real-time gameplay events. The frontend establishes a single persistent connection to the Gateway, which proxies it to the `Gameplay` service's STOMP broker. Messages are published to destinations prefixed with `/app/` (routed to `@MessageMapping` handlers) and broadcast to subscribers via `/topic/` destinations.

OpenFeign (internal HTTP): Used for inter-service calls at session boundaries:

- `Gameplay` → `Core`: Called once at `startGame()` to fetch the list of `CampaignActor` bootstrap data.
- `Core` → `Gameplay`: Called when the campaign detail page requests history or live actor stats, enriching the response with character names before returning it to the frontend.

Key Interaction Flows: *Joining a campaign:*

1. Frontend calls `POST /mapforge-core/api/v1/campaigns/{id}/join` with {`userId`, `characterId`}.
2. `Core` creates a `CampaignMember` and a `CampaignActor` record transactionally.
3. Frontend navigates to the gameplay view lobby.

Starting a game:

1. GM sends STOMP message to `/app/room.{id}.start`.

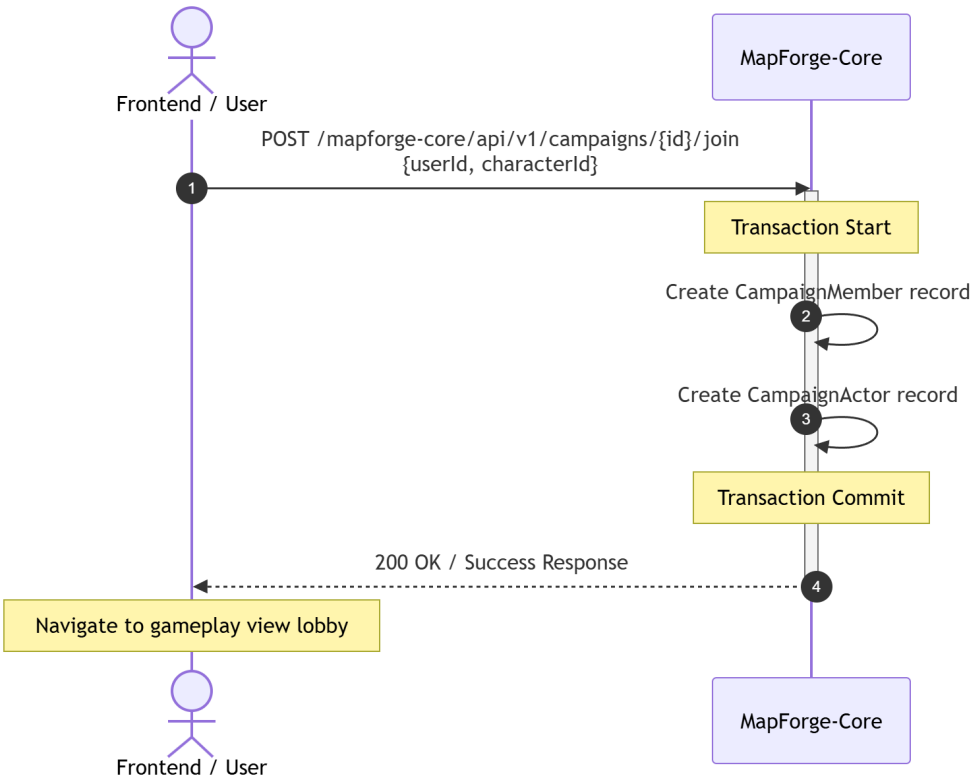

Figure 4: Joining a campaign sequence diagram.

2. `GameController` calls `GameService.startGame()`.
3. `GameService` calls `CoreClient.getActorsForCampaign()` via Feign.
4. Actors are copied into the Gameplay DB with their current stats.
5. A `GameSession` and the first `Turn` are persisted.
6. `GameStateDTO` is broadcast to all subscribers of `/topic/room.{id}.state`.
7. All clients receive `status: ACTIVE` and initialize the Phaser board.

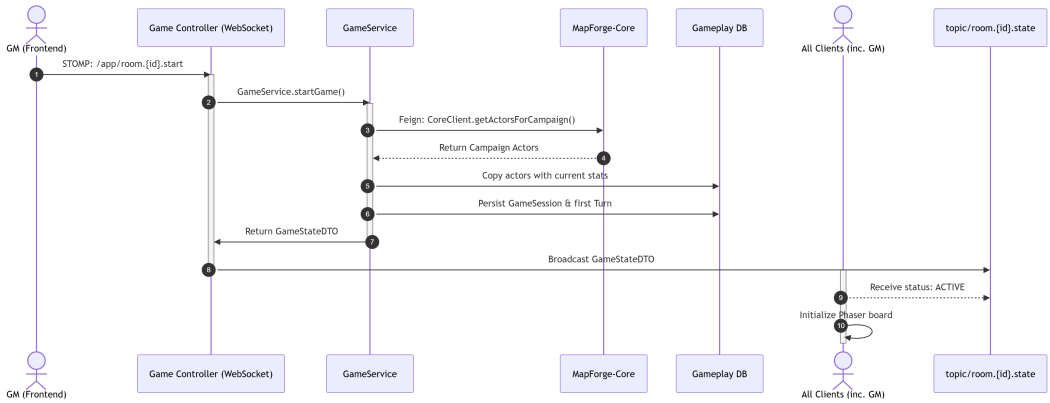

Figure 5: Starting a game sequence diagram.

Performing a move:

1. Player sends STOMP message to `/app/room.{id}.action` with {`actorId, type: "MOVE", x, y`}.
2. Server validates: is it this actor's turn? Is the destination within bounds? Is it within speed range? Is the cell unoccupied?
3. If valid: `CampaignActor` position is updated, `MovementAction` is persisted, result is broadcast to `/topic/room.{id}.action`.
4. All clients' Phaser boards animate the actor sprite to the new position.

3.5 Behaviour

Stateful components:

- **Gameplay service:** Stateful. Maintains `PresenceService` (in-memory player registry per room) and the Hibernate first-level cache. Persistent state (actors, sessions, turns, actions) lives in PostgreSQL.

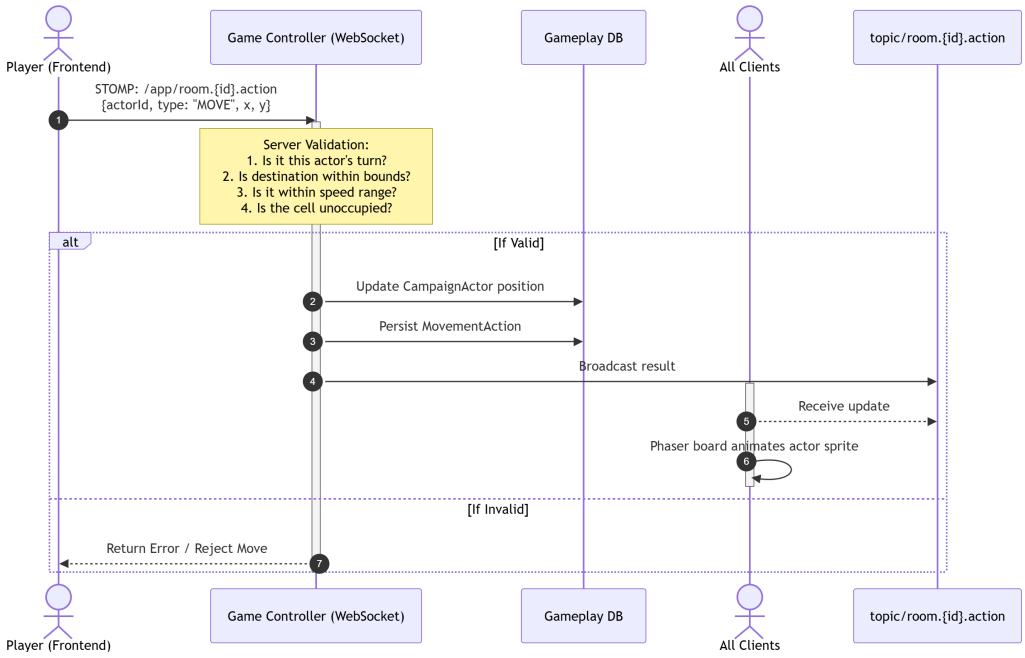

Figure 6: Perform a move sequence diagram.

- **Angular client:** Stateful. Maintains current game state, actor positions, action log, and the Phaser scene graph in memory. State is re-hydrated from HTTP on reconnection.

Stateless components:

- **Core service:** Stateless between requests. No in-memory session data; all reads and writes go directly to PostgreSQL.
- **Gateway:** Stateless. Pure routing and JWT validation.
- **Eureka:** Stateful infrastructure component managing service registry, but not application-level state.

3.6 Data and Consistency Issues

3.6.1 CAP Theorem Position

The system deliberately prioritizes **Consistency over Availability** (CP) within a game session. This is the correct choice for a turn-based game: it is unacceptable for the system to allow two actors to occupy the same tile or for a dead actor to continue acting, even if enforcing correctness requires momentarily rejecting a request.

During a network partition or service crash, active game sessions become **unavailable** until the service recovers. However, when it recovers, the state is **fully consistent**—every actor is exactly where the last successful action placed them.

An AP (available but eventually consistent) alternative was rejected because the domain does not tolerate conflicting states: two simultaneous attacks on the same NPC could result in contradictory HP values that cannot be cleanly merged.

3.6.2 Consistency Model

- **Turn-based serialization:** The primary consistency mechanism is the turn protocol itself. Since only one actor is authorized to act at any given moment, concurrent write conflicts are structurally prevented at the application layer. The server validates the `actorId` in every action payload against the current `Turn` record before processing.
- **Atomic transactions:** All gameplay actions and their side effects (e.g., an `AttackAction` combined with an HP update, or an attack leading to a `DeathAction` and a turn-order modification) are wrapped in a single `@Transactional` method, ensuring no intermediate broken states are ever visible.
- **Cross-DB eventual consistency:** The live HP and XP values visible on the campaign detail page (served by `Core` via a Feign call to `Gameplay`) may lag slightly behind the in-session reality. This is an explicit and acceptable trade-off: the campaign detail page is a reporting view, not a live game view. Clients in the active game session always see the authoritative state directly from `Gameplay`.

3.6.3 Shared Data and Synchronization

In a microservices environment, data sharing introduces coupling. The design minimizes this:

- **At session start only:** `Gameplay` calls `Core` exactly once—to bootstrap the actor snapshot. After this, `Gameplay` is fully autonomous for the duration of the session.
- **Read-only cross-service queries:** `Core` may call `Gameplay` for history enrichment (character names + action log), but `Core` never writes to the Gameplay database.
- **No shared tables:** Each service's database is entirely private. The `CampaignActor` duplication is intentional and the two copies serve different purposes (enrollment record vs. live session state).

3.7 Fault-Tolerance

The system is designed to handle **Fail-Stop** failures, where a component ceases to function but does not send malicious or corrupted data.

3.7.1 Service Isolation via Microservices

Because `Core` and `Gameplay` are independent processes with independent databases and independent HikariCP connection pools, the failure of one cannot directly cause the other to crash. A database connection saturation in `Gameplay` does not affect `Core`'s connection pool.

3.7.2 Service Discovery and Health Monitoring

Netflix Eureka monitors service liveness through a heartbeat protocol:

- Every service instance sends a periodic heartbeat to the Eureka server every 30 seconds.
- If a service fails to renew its lease within the configured timeout, Eureka removes it from the registry.
- The `Spring Cloud Gateway` continuously refreshes its routing table from Eureka, stopping traffic to failed instances automatically.

3.7.3 Feign Circuit Breaker and Fallback

The `Core` service uses a Feign client with a circuit-breaker fallback to call `Gameplay`. When `Gameplay` is unavailable:

- `getHistory()` returns `null`, and the frontend displays a "No history yet" placeholder.
- `getRuntimeActors()` returns an empty list, and the campaign detail page shows static data from `Core`'s own records.

This ensures that the `Core` service remains fully operational even when `Gameplay` is down, satisfying the availability non-functional requirement.

3.7.4 State Recovery on Reconnection

When a client reconnects after a `Gameplay` service restart:

1. The Angular component calls `GET /mapforge-gameplay/campaigns/{id}/state`.
2. The response contains the current `GameSession` status, turn index, and turn order—all read directly from PostgreSQL.
3. The component then calls `GET /mapforge-gameplay/campaigns/{id}/actors` to retrieve all actor IDs, types, HP values, and (x, y) positions.
4. The Phaser board is initialized with this data, rendering all actors exactly where they were before the crash.

5. The WebSocket connection is re-established, and the game continues.

No action log replay is needed because actor positions are stored directly on the `CampaignActor` row and updated on every move, making the latest state immediately queryable.

3.7.5 Turn Timeout (Design)

A scheduled job is designed to auto-advance the turn if the active player disconnects and their turn does not complete within a configurable timeout (e.g., 5 minutes). This prevents the game from hanging indefinitely due to a single disconnected player. The scheduled check runs every 30 seconds and compares the `Turn` creation timestamp against the current time.

3.8 Availability

3.8.1 Caching

The system currently does not implement an explicit distributed cache (e.g., Redis). The primary availability mechanisms are:

- **HTTP re-fetch on reconnect:** The Angular client re-fetches full state on component initialization, acting as a client-side cache miss that is always resolved from the authoritative database.
- **Hibernate first-level cache:** Within a single request's transaction, repeated reads of the same entity are served from Hibernate's session cache, reducing database round-trips.

3.8.2 Load Balancing

Client-side load balancing is provided by **Spring Cloud LoadBalancer** (integrated with Eureka). The Gateway uses a `Round-Robin` strategy to distribute requests across multiple instances of the same service. Feign clients also resolve service names through Eureka and benefit from the same load balancing.

3.8.3 Network Partition Behavior

In the event of a network partition:

- **Client cannot reach Gateway:** The Angular frontend cannot make any requests. The user sees a connection error. No data is corrupted on the server side.
- **Gateway cannot reach Gameplay:** REST requests to Gameplay endpoints return 503 (or the Feign fallback). Active game sessions for already-connected clients continue until their WebSocket connection drops.

- **Gameplay cannot reach Core:** Only `startGame()` is affected, as it is the only point where Gameplay calls Core. In-progress sessions are unaffected since they do not require Core after the bootstrap phase.
- **Gameplay cannot reach its DB:** Gameplay rejects all state-mutating operations (moves, attacks) with a database error. The session is effectively frozen until the partition heals, consistent with the CP choice.

3.9 Security

3.9.1 Authentication and Identity Management

The system implements centralized, stateless authentication:

- **Token-Based Auth:** Upon successful login, the `Auth` endpoint issues a **JSON Web Token (JWT)** signed with an HMAC-SHA256 algorithm using a shared secret.
- **Statelessness:** `Core` and `Gameplay` validate the JWT's cryptographic signature locally without contacting any external session store, ensuring horizontal scalability of the authentication check.

3.9.2 The Gateway as a Security Shield

The **Spring Cloud Gateway** acts as the system's security perimeter:

- **Global JWT Filter:** Every inbound request is intercepted and its `Authorization` header is validated. Requests with missing, expired, or tampered tokens are rejected with `401 Unauthorized` before they reach any internal service.
- **CORS Policy:** The Gateway applies a global CORS configuration, allowing the Angular frontend origin while rejecting unauthorized cross-origin requests.

3.9.3 Authorization and Role-Based Access Control

Once authenticated, role-based access control is enforced at the service level:

1. **User:** Can manage their own profile and create characters or campaigns.
2. **Game Master:** The creator of a specific campaign. Has exclusive rights to start, pause, resume, and finish game sessions. Controls all NPC actors during their turns.
3. **Player:** A user enrolled in a campaign. May perform actions only on the `CampaignActor` whose turn it currently is, validated server-side on every action.

3.9.4 Cryptographic Details

- **Password Hashing:** User passwords are hashed using **BCrypt** with a cost factor of 10, providing resistance to brute-force and rainbow table attacks.
- **JWT Secret:** The JWT signing secret is injected at runtime via an environment variable (`JWT_SECRET`), never hardcoded in source code.

4 Implementation

4.1 Network Protocols and Data Representation

The system uses a multi-protocol approach:

- **HTTP/1.1 (REST):** All synchronous operations: user registration, login, character/campaign CRUD, initial state queries, and campaign history. Responses are JSON-encoded DTOs.
- **WebSockets (STOMP):** All real-time gameplay events. The `Gameplay` service registers a STOMP endpoint at `/gameplay-ws`. The application destination prefix is `/app`; broadcast topics use the `/topic` prefix.
- **JSON (UTF-8):** All inter-service and client-server data is serialized as JSON using Jackson. DTOs are defined per service, and shared data structures (e.g., `CampaignActorBootstrapDTO`) are duplicated across service boundaries rather than shared as a library, preserving service independence.

4.2 Data Persistence and Querying

- **JPA / Hibernate:** All database interactions use **Spring Data JPA**. Repositories extend `JpaRepository`, providing standard CRUD operations and the ability to define derived query methods (e.g., `findByCampaignIdAndTurnIndex`).
- **Liquibase:** Database schema is managed via SQL changesets. Each service has an independent `db.changelog-master.xml` that includes numbered SQL files (e.g., `001-initial-schema.sql`, `002-add-position-to-campaign-actors.sql`). On startup, Spring Boot auto-runs Liquibase before accepting traffic, ensuring schema and code are always in sync.
- **HikariCP:** Connection pooling is provided by HikariCP (Spring Boot's default), managing a pool of reusable database connections per service to minimize the overhead of connection establishment under load.
- **Environment-based configuration:** Database URLs, credentials, and the JWT secret are never hardcoded. Each service reads them from environment variables at startup (`DOCKER_DB_URL`, `POSTGRES_USER`, `POSTGRES_PASSWORD`, `JWT_SECRET`), injected by Docker Compose in production and by `.env` files during local development.

4.3 Authentication and Authorization

- **JWT:** The `Auth` endpoint issues JWTs signed with HMAC-SHA256. Services validate the signature using a shared secret injected via environment variable.
- **Spring Security:** Each service's security filter chain is configured to validate JWTs on protected endpoints. The `Gameplay` service's WebSocket endpoint requires a valid JWT in the STOMP `CONNECT` frame headers.

4.4 Frontend Implementation

- **Angular 19 & TypeScript:** The frontend is a Single Page Application built with Angular's standalone component architecture. TypeScript enforces type safety on all API response models, catching deserialization errors at compile time.
- **Angular Signals:** Reactive state management uses Angular Signals (`signal()`, `computed()`, `effect()`) for fine-grained reactivity without the overhead of zone-based change detection on the game board.
- **RxJS:** WebSocket events from the STOMP client are published to RxJS `Subject` streams (`gameState$`, `action$`, `death$`, `presence$`). Components subscribe to these streams and update Angular signals accordingly.
- **Phaser.js:** The 2D game board is rendered using Phaser, loaded via dynamic `import()` to prevent SSR execution on the Node.js server. The Phaser scene is initialized only after the Angular component confirms it is running in a browser context (`isPlatformBrowser`). Actor sprites are stored in a `Map<number, any>` keyed by actor ID, allowing O(1) lookups for position updates received over WebSocket.
- **DaisyUI & Tailwind CSS:** The UI uses Tailwind CSS utility classes and DaisyUI components for responsive layout, dark theming, and interactive elements (tabs, cards, badges, stats).

4.5 Containerization and Orchestration

Every system component is containerized using Docker:

- **Docker Containers:** Each Spring Boot service has a multi-stage `Dockerfile`: a build stage using `gradle:jdk21` to compile the fat JAR, and a runtime stage using `eclipse-temurin:21-jre` to minimize the final image size.
- **Docker Compose:** A single `docker-compose.yml` defines the entire system. It configures the internal `microservices-net` bridge network, injects environment variables (DB URLs, credentials, Eureka address), and enforces startup dependencies using health checks (e.g., `Core` waits for both Eureka and PostgreSQL to be healthy before starting).

- **PostgreSQL Volume:** A named Docker volume (`postgres_data`) persists the database across container restarts, preventing data loss during development cycles.

4.6 Technological Details

- **Java 21:** The current LTS release. Used for its mature concurrency model and compatibility with the Spring Boot 3 ecosystem.
- **Spring Boot 3.5:** Provides auto-configuration for JPA, Security, WebSocket, and Actuator health checks.
- **Spring Cloud 2025:** Provides Eureka Client, Gateway, and OpenFeign. The Feign client generates HTTP clients from annotated interfaces and resolves service names through Eureka's registry.
- **Lombok:** Eliminates boilerplate `@Getter`, `@Setter`, `@AllArgsConstructor`, and `@RequiredArgsConstructor` annotations from all entity and DTO classes.
- **Liquibase 4.23:** Manages database migrations via SQL changesets. The Gradle Liquibase plugin is configured in each subproject's `build.gradle.kts` to run migrations against the local database during development.

5 Validation

5.1 Automatic Testing

Integration tests for the `Gameplay` service are implemented using **JUnit 5**, **Spring Boot Test**, **Testcontainers**, and **Mockito**.

5.1.1 Test Infrastructure

Testcontainers spins up a real **PostgreSQL 17** Docker container for each test run. This means tests execute against the actual database engine—not an in-memory H2 substitute—ensuring that Liquibase migrations, UUID types, and PostgreSQL-specific behavior are all exercised. The container URL is injected into the Spring context via `@DynamicPropertySource`:

```
@Container
static PostgreSQLContainer<?> postgres =
    new PostgreSQLContainer<>("postgres:17")
        .withDatabaseName("MapForgeGameplay")
        .withUsername("postgres")
        .withPassword("postgres");

@DynamicPropertySource
static void configureProperties(DynamicPropertyRegistry registry)
   {
    registry.add("spring.datasource.url", postgres::getJdbcUrl);
```

```
    registry.add("eureka.client.enabled", () -> "false");
}
```

Mockito is used to mock the two external dependencies that cannot be tested in isolation:

- `CoreClient` (Feign): Mocked via `@MockitoBean` to return a predefined set of `CampaignActorBootstrapDTO` objects without requiring a running `Core` service.
- `SimpMessageSendingOperations` (WebSocket broker): Mocked to prevent broadcast calls from failing when no STOMP broker is running.

5.1.2 Test Coverage

Tests are organized in `GameplayIntegrationTest` and cover the following scenarios:

Movement Validation (Requirement: Gameplay Logic)

- **Valid move within bounds and speed:** Asserts that a `MovementAction` is persisted and the actor's coordinates are updated in the DB.
- **Move outside map bounds:** Asserts that an `IllegalArgumentException` is thrown with message "out of bounds" and no `MovementAction` is written.
- **Move to negative coordinates:** Same rejection behavior for $(x < 0)$ or $(y < 0)$.
- **Move to occupied cell:** Moves actor 1 to $(3, 5)$, ends turn, then asserts that actor 2 cannot move to $(3, 5)$, with an "occupied" exception.
- **Move exceeding speed limit:** Moves actor 1 to a known position, then asserts that a move beyond the speed radius throws "exceeds speed limit".
- **Action out of turn:** Asserts that sending an action for actor 2 when it is actor 1's turn throws an `IllegalStateException`.

Combat (Requirement: Combat)

- **Correct damage calculation:** Asserts that the `AttackAction.damage` equals the attacker's `weaponDamage` and that the target's HP in the DB is reduced by exactly that amount.
- **HP floor at zero:** Sets target HP to 5 (less than attacker's weapon damage of 15), then asserts that target HP is exactly 0, never negative.

Death Events (Requirement: Death and Spectator Mode)

- **DeathAction persisted on kill:** Asserts that a `DeathAction` row is written with the correct `actorId` and `killerId` when HP reaches 0.
- **Dead actor removed from turn order:** Asserts that the killed actor's ID no longer appears in `GameSession.turnOrder` after the attack.
- **Game finishes when one actor remains:** Asserts that `GameSession.status` transitions to `FINISHED` when only one actor remains in the turn order.

5.1.3 How to Run

```
# Requires Docker to be running (for Testcontainers)
./gradlew :mapforge-gameplay:test
```

Test reports are available at `mapforge-gameplay/build/reports/tests/test/index.html`.

5.2 Acceptance Testing

Manual acceptance testing was performed to validate the integrated system, covering scenarios that require a live multi-client setup and cannot be easily automated:

- **Multi-player turn synchronization:** Two browser tabs (one GM, one Player) were opened simultaneously. The GM started the game and the Player's lobby view transitioned to the active game board automatically via WebSocket. Turn advancement was verified to update both clients in real-time.
- **Movement broadcast:** The GM moved an actor on the Phaser board. The sprite was verified to animate to the new position on the Player's board without page refresh, via the `/topic/room.{id}.action` broadcast.
- **Session pause/resume:** The GM paused the session. Both clients displayed the pause banner. The Player was unable to perform actions. On GM resume, both clients returned to the active state.
- **Session recovery:** The `mapforge-gameplay` Docker container was stopped mid-game (`docker stop mapforge-gameplay`). After restarting it (`docker start mapforge-gameplay`), the Player refreshed the page and the board re-rendered all actors in their last known positions.
- **Gameplay down, Core up:** With `Gameplay` stopped, the campaign detail page was verified to load successfully, displaying a "No history yet" message in the Activity tab rather than an error, confirming the Feign fallback.
- *Rationale for manual testing:* These scenarios require coordinated multi-client WebSocket state, Docker container lifecycle management, and visual verification

of the Phaser game board—aspects that are impractical to fully automate in a unit or integration test framework without a full end-to-end testing infrastructure (e.g., Cypress with WebSocket support).

6 Deployment

6.1 Prerequisites

The following software must be installed on the host machine:

- **Docker Desktop** (version 24.0 or later) with Docker Compose v2
- **JDK 21** (for running Liquibase migrations locally; not required if only using Docker)
- **Node.js 20+** and **npm** (for building the Angular frontend)
- **Git** (for cloning the repository)

6.2 Installation from Scratch

1. **Clone the repository:**

```
git clone <repository-url>
cd MapForger
```

2. **Configure environment variables:** Each microservice reads its configuration from a `.env` file in its own directory. Create the following files:

   `mapforge-core/.env`:

```
DOCKER_DB_URL=jdbc:postgresql://db-core:5432/MapForge
CORE_DB_URL=jdbc:postgresql://localhost:5432/MapForge
POSTGRES_USER=postgres
POSTGRES_PASSWORD=postgres
JWT_SECRET=<your-secret-key>
```

   `mapforge-gameplay/.env`:

```
DOCKER_DB_URL=jdbc:postgresql://db-gameplay:5432/
   MapForgeGameplay
GAMEPLAY_DB_URL=jdbc:postgresql://localhost:5433/
   MapForgeGameplay
POSTGRES_USER=postgres
POSTGRES_PASSWORD=postgres
JWT_SECRET=<your-secret-key>
```

   The `JWT_SECRET` must be identical in both files. The same `docker-compose.yml` environment variable values must match these files.

3. **Build and start all services:**

```
docker compose up --build
```

On first startup, each Spring Boot service runs its Liquibase migrations automatically, creating all required tables before accepting traffic.

Both databases are created automatically on first startup via the `POSTGRES_DB` environment variable passed to each PostgreSQL container. `db-core` creates `MapForge` and `db-gameplay` creates `MapForgeGameplay`. On subsequent starts, the named volumes (`core_data` and `gameplay_data`) persist the data, so the initialization step is skipped. No manual database creation is required.

4. **Verify deployment:**

Navigate to `http://localhost:8761` to confirm all services are registered in the Eureka dashboard. Expected registrations: `MAPFORGE-CORE`, `MAPFORGE-GAMEPLAY`, `MAPFORGE-GATEWAY`.

The Angular frontend is served at `http://localhost:4200` (development server) or can be built and served statically.

6.2.1 Docker Compose Services

The `docker-compose.yml` defines the following services:

- **db-core:** PostgreSQL 17 instance dedicated to the `Core` service, hosting the `MapForge` database. Exposed on port `5432`. Persisted via the `core_data` named volume. Health-checked via `pg_isready` before `mapforge-core` starts.
- **db-gameplay:** PostgreSQL 17 instance dedicated to the `Gameplay` service, hosting the `MapForgeGameplay` database. Exposed on port `5433` to avoid conflicts with `db-core` on the host machine. Persisted via the `gameplay_data` named volume. Health-checked via `pg_isready` before `mapforge-gameplay` starts.
- **mapforge-eureka:** The Eureka service registry. Health-checked via the Spring Boot Actuator `/actuator/health` endpoint before any other service starts.
- **mapforge-core:** The Core microservice. Depends on both `mapforge-eureka` and `db-core` being healthy.
- **mapforge-gameplay:** The Gameplay microservice. Depends on both `mapforge-eureka` and `db-gameplay` being healthy.
- **mapforge-gateway:** The API Gateway. Depends only on `mapforge-eureka` being healthy, as it holds no database state.

All services share a single Docker bridge network (`microservices-net`), enabling inter-container communication by container name (e.g., `db-core:5432`, `db-gameplay:5432`). The two database instances are intentionally separated, reflecting the microservice principle that each service owns its data exclusively. Neither microservice can directly access the other's database at the network level.

6.2.2 Startup Ordering and Health Checks

A critical aspect of the deployment is ensuring that services start in the correct order. Because `Core` and `Gameplay` depend on both Eureka and their respective databases being fully operational before registering, Docker Compose health checks are used to enforce this ordering.

Each service uses `condition: service_healthy` in its `depends_on` declaration rather than the weaker `condition: service_started`. The difference is significant: `service_started` only waits for the container process to launch, while `service_healthy` waits for the container's health check to pass—meaning the service is genuinely ready to accept connections.

- **PostgreSQL:** Health-checked via `pg_isready`, ensuring the database is accepting connections before any microservice attempts to run its Liquibase migrations.
- **Eureka:** Health-checked via the Spring Boot Actuator `/actuator/health` endpoint, ensuring the service registry is fully operational before the Gateway or microservices attempt to register or resolve service names.

Without these checks, a race condition occurs: a microservice may start, attempt to register with Eureka before it is ready, and fail silently. Subsequent client requests routed through the Gateway would return `500 Internal Server Error` because no healthy service instances are registered in the routing table.

6.3 Rebuilding Individual Services

To rebuild and restart a single service without affecting others:

```
docker compose up --build mapforge-gameplay
```

6.4 Running Database Migrations Locally

To apply Liquibase migrations directly against a local PostgreSQL instance (requires the service's `.env` to contain a valid `DB_URL`):

```
./gradlew :mapforge-core:update
./gradlew :mapforge-gameplay:update
```

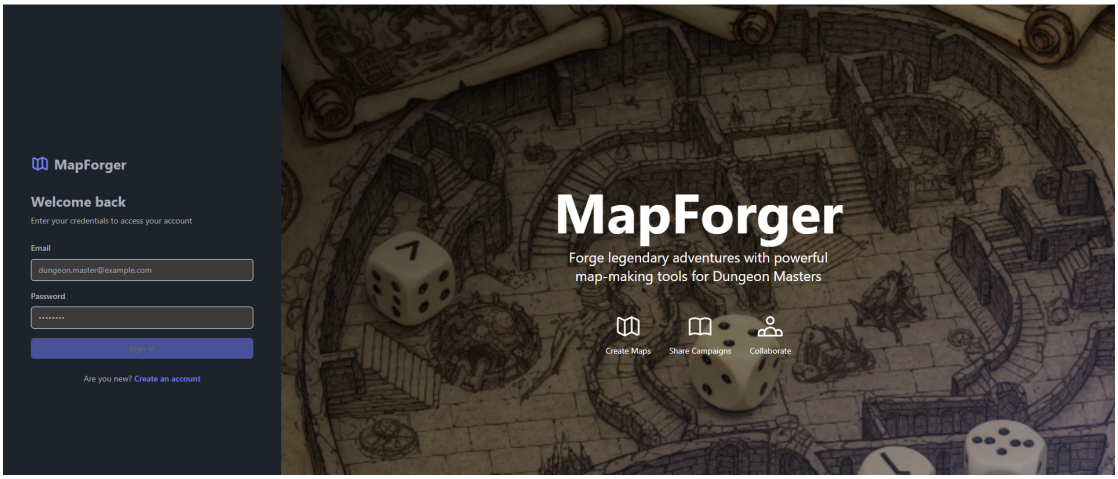

Figure 7: Login view of the application. User must insert email and password to be authenticated and authorized to the service.

Figure 8: Signup view of the application. User must insert all the necessary information in order to be registered.

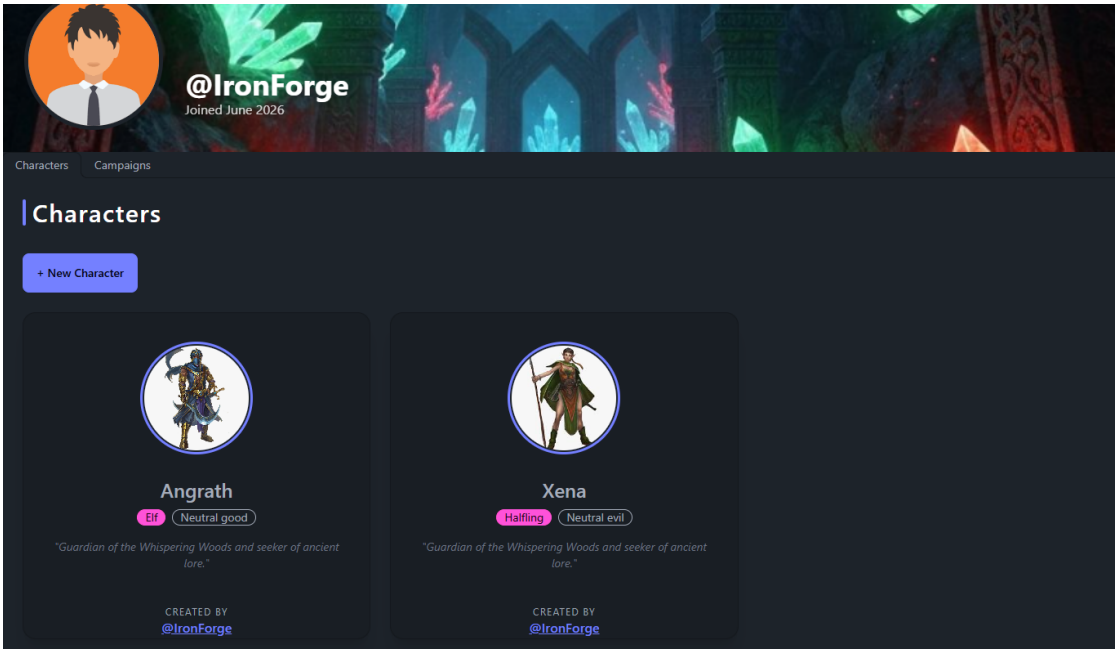

Figure 9: This is the profile section. In this page we can see all the characters and campaigns created by a user.

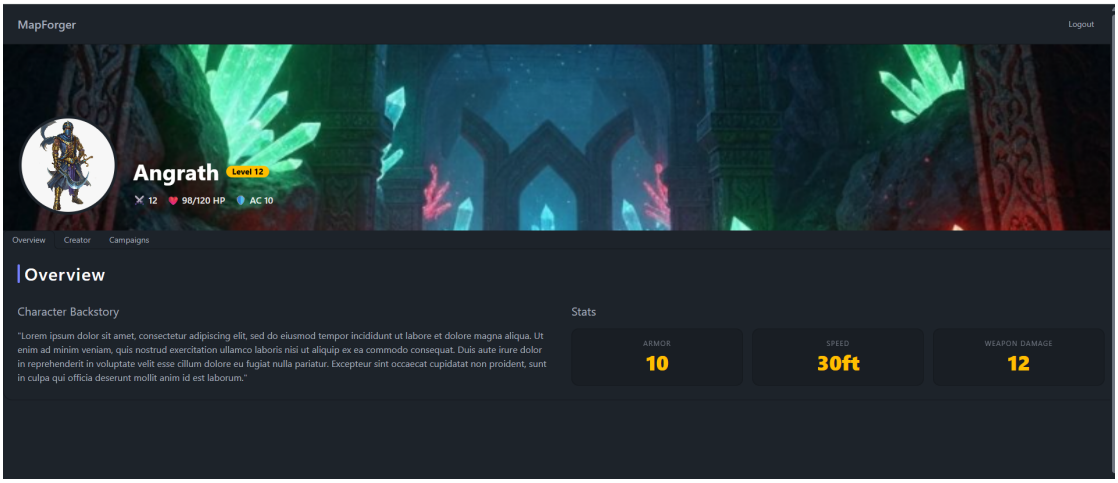

Figure 10: This is the detail section of a character. Users can check the information related to character, like stats, backstory, creator and the campaigns it has played.

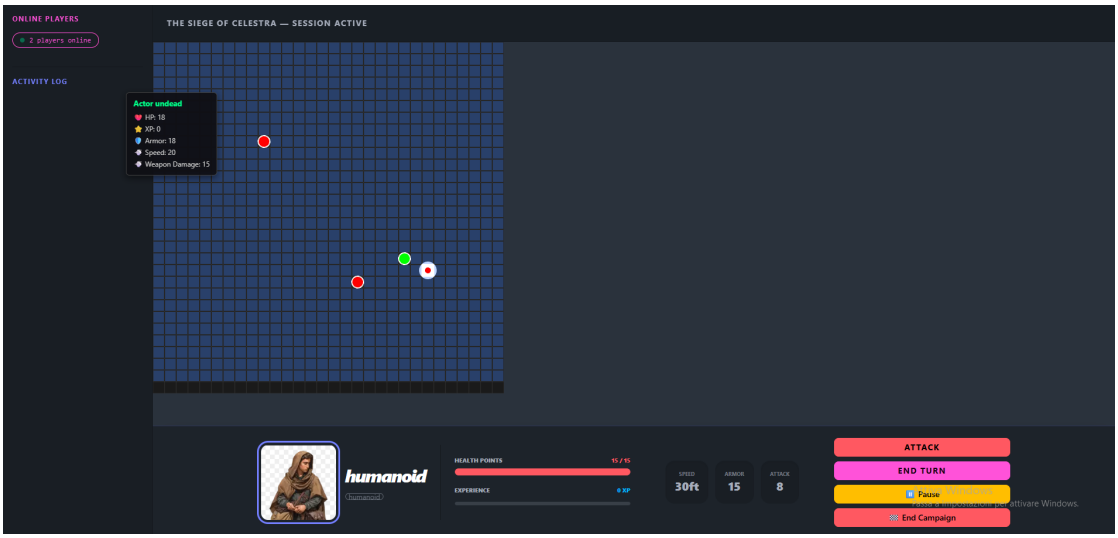

Figure 11: This is the gameplay view. In the top corner it can be seen the number of online players. In the center there is the board with all the campaign actors. By clicking on the campaign actor, its information can be seen(hp, xp, attack damage, speed, etc...). The red circles are the NPCs, whereas the white circles are actual players. The circle with the wider border is the current player. Every action gets recorded on the left panel.

7 User Guide

8 Self-evaluation

8.1 Eduard Toni Alexandru

Role As the sole developer of this project, I was responsible for all aspects of the system: requirements analysis, architectural design, backend implementation across both microservices, frontend development, DevOps configuration, and testing.

Strengths The architectural decision to give each microservice its own database, communicate via Feign at session boundaries only, and use WebSockets exclusively for real-time events proved to be sound. The resulting system is genuinely fault-tolerant: the Core service continues to function when Gameplay is unavailable, and sessions are fully recoverable from the database after a crash without any manual intervention. The CAP theorem trade-off (CP over AP) was not an abstract choice but a concrete, deliberate design decision that manifests in the turn validation logic and the atomic transaction wrapping of every gameplay action.

The use of Liquibase from early in the project paid dividends when the schema evolved significantly—moving tables between services, renaming columns, adding position fields—without ever losing data or requiring manual schema repair.

The Phaser.js integration with Angular's SSR architecture required careful handling (dynamic import, `isPlatformBrowser` guards, `app.routes.server.ts` client-only rendering) but resulted in a clean, performant game board with smooth animations and real-time updates.

Weaknesses and Limitations The integration test coverage, while covering all core gameplay invariants, does not include end-to-end tests of the WebSocket broadcast path itself. Testing this layer would require a test STOMP client, which was outside the scope of the course timeline.

References

AtomicJar, Inc. Testcontainers for java documentation. `https://java.testcontainers.org`, 2025. Accessed: 2025.

Eric A. Brewer. Towards robust distributed systems. *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2000. URL `https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf`.

Docker, Inc. Docker documentation. `https://docs.docker.com`, 2025. Accessed: 2025.

Ian Fette and Alexey Melnikov. The websocket protocol — rfc 6455. `https://datatracker.ietf.org/doc/html/rfc6455`, 2011. Accessed: 2025.

Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002. URL https://dl.acm.org/doi/10.1145/564585.564601.

Google LLC. Angular documentation. https://angular.dev/docs, 2025. Accessed: 2025.

Michael Jones, John Bradley, and Nat Sakimura. Json web token (jwt) — rfc 7519. https://datatracker.ietf.org/doc/html/rfc7519, 2015. Accessed: 2025.

Liquibase Inc. Liquibase documentation. https://docs.liquibase.com, 2025. Accessed: 2025.

Netflix, Inc. Netflix eureka — service discovery in the cloud. https://github.com/Netflix/eureka/wiki, 2024. Accessed: 2025.

Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015. ISBN 978-1491950357.

Photon Storm Ltd. Phaser 3 — desktop and mobile html5 game framework. https://phaser.io/phaser3, 2025. Accessed: 2025.

Chris Richardson. *Microservices Patterns: With Examples in Java*. Manning Publications, 2018. ISBN 978-1617294549.

STOMP Working Group. Stomp protocol specification, version 1.2. https://stomp.github.io/stomp-specification-1.2.html, 2012. Accessed: 2025.

The PostgreSQL Global Development Group. Postgresql 17 documentation. https://www.postgresql.org/docs/17/, 2024. Accessed: 2025.

VMware, Inc. Spring boot reference documentation. https://docs.spring.io/spring-boot/docs/current/reference/html/, 2025a. Accessed: 2025.

VMware, Inc. Spring cloud reference documentation. https://docs.spring.io/spring-cloud/docs/current/reference/html/, 2025b. Accessed: 2025.

VMware, Inc. Spring cloud openfeign reference documentation. https://docs.spring.io/spring-cloud-openfeign/docs/current/reference/html/, 2025c. Accessed: 2025.

VMware, Inc. Spring cloud gateway reference documentation. https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/, 2025d. Accessed: 2025.

Wizards of the Coast. *Player's Handbook*, 5 edition, 2014. URL https://dnd.wizards.com/products/rpg_playershandbook.