OverAll System

Multiplayer Platform – System Architecture & Documentation

---

1. System Overview

The multiplayer platform is a **real-time online gaming system** that enables players to:

- Authenticate securely
- Maintain live presence across multiple services
- Join matchmaking queues and get paired fairly
- Communicate with other players (chat)
- Participate in matches whose results are processed and stored reliably

The system is designed using **event-driven architecture** with **decoupled services** for scalability, fault tolerance, and maintainability.

Key Design Principles:

- **Loose coupling** via RabbitMQ events
- **Real-time updates** via WebSocket connections
- **Authoritative state ownership** per domain (Player Service, Matchmaking Service, Game Logic Service)
- **Horizontal scalability** for high concurrency
- **Ephemeral state caching** with Redis
- **JWT-based stateless authentication**

---

2. Core System Components

Component	Responsibility	Communication
Gateway Service	Real-time communication hub for WebSocket clients; relays events	RabbitMQ (events), WebSocket
Player Service	Manages authentication, profiles, stats, and presence	RabbitMQ, Redis
Matchmaking Service	Handles queues, matches, and pairing logic	RabbitMQ, Redis
Game Logic Service	Executes game rules and match resolution	RabbitMQ

Redis	Ephemeral state storage (presence, queues, match participants)	Direct connection (all services)
RabbitMQ	Event bus for inter-service communication	Pub/Sub, topic exchange

---

3. Data Flow & Presence

3.1 Player Presence

Redis stores ephemeral player data:
presence:{userId} -> { userId, username, rating, status (IDLE|QUEUED|IN_GAME), missedHeartbeats }
online_players (set) -> all currently connected users

-
- **Heartbeat mechanism**:
    - Clients send periodic heartbeat via Gateway
    - Gateway updates Player Service
    - Presence Janitor evicts stale sessions

3.2 Matchmaking Queue

- Players join queues through Matchmaking Service API
- Queue stored in Redis (sorted by rating)
- Worker matches compatible players
- Match creation triggers `match.created` event

3.3 Chat

- Real-time chat validated through match participation
- Messages relayed via Gateway → WebSocket
- Events stored/published as `chat.private`

---

4. Inter-Service Communication

4.1 Event-Driven Communication

- All services communicate via **RabbitMQ topic exchange `events`**
- Example events:

Event	Producer	Consumer	Purpose

`player.connected`	Gateway	Player Service	Register live presence
`player.disconnected`	Gateway	Player Service	Cleanup presence
`player.heartbeat`	Gateway	Player Service	Maintain alive status
`player.kick`	Gateway	Player Service	Force logout of duplicate sessions
`match.created`	Matchmaking Service	Gateway	Notify players of match
`match.started`	Game Logic Service	Gateway	Notify players game has started
`match.ended`	Game Logic Service	Player Service	Update stats, reset presence
`chat.private`	Gateway	Gateway / Recipient	Relay chat messages securely

4.2 Communication Flow Diagram

```
graph TD
  PlayerClient -->|WebSocket| Gateway
  Gateway -->|player.connected / disconnected| PlayerService
  Gateway -->|player.heartbeat| PlayerService
  Gateway -->|match.created / match.started| PlayerClient
  MatchmakingService -->|match.created| Gateway
  GameLogicService -->|match.started / match.ended| Gateway
  Gateway -->|chat.private| Gateway
  Gateway --> Redis
```

---

5. API & WebSocket Endpoints

5.1 Gateway Service WebSocket

- Path: `/ws`
- Authentication: JWT token via query param
- Real-time messages:
  - `MATCH_CREATED`
  - `CHAT_MESSAGE`

    - `ERROR` (e.g., kicked sessions)
- Heartbeat interval: 30 seconds

5.2 Player Service REST API

- `/auth/register` – Create player
- `/auth/login` – Login, issue JWT
- `/auth/refresh` – Refresh JWT
- `/auth/logout` – Stateless logout
- `/players/me` – Get/update own profile
- `/players` – Public player data

5.3 Matchmaking Service REST API

- `/join` – Join matchmaking queue
- `/leave` – Leave queue
- Events: `match.created`

---

6. System Workflows

6.1 Player Login & Connection

1. Client authenticates via Player Service → JWT
2. Client connects to Gateway WebSocket with JWT
3. Gateway validates token → publishes `player.connected`
4. Player Service updates Redis → sets presence
5. Player available for matchmaking

6.2 Matchmaking Flow

1. Player joins queue via Matchmaking API
2. Redis sorted set tracks player by rating
3. Worker polls queue → finds partner → creates match
4. Publishes `match.created` event → Gateway notifies clients
5. Match starts → Game Logic executes
6. Match ends → `match.ended` event → Player Service updates stats

6.3 Chat Flow

1. Player sends chat via WebSocket
2. Gateway validates match participation
3. Gateway publishes `chat.private` event
4. Recipient receives message in real-time

---

7. Fault Tolerance & Reliability

Mechanism	Purpose
RabbitMQ persistent queues	Prevent message loss
Heartbeat monitoring	Detect disconnected clients
Redis ephemeral storage	Fast read/write, auto-expire stale entries
Event acknowledgment (ack/nack)	Ensure at-least-once processing
Retry loops on RabbitMQ init	Gateway robust to downtime
Stateless JWT	Horizontal scaling without shared sessions

---

8. Scaling & Deployment

8.1 Horizontal Scaling

- Gateway: Multiple WS instances behind load balancer
- Matchmaking: Multiple workers for concurrent matching
- Player Service: Scales for read-heavy operations

8.2 Vertical Scaling

- Redis: Increase memory for large player base
- RabbitMQ: Clustered for high throughput

8.3 Deployment

- Dockerized services
- Environment-based configuration via `.env`
- Optional regional deployment for low latency

---

9. Security Considerations

- JWT authentication for all critical endpoints
- WebSocket authentication per connection
- Validation of match participation for chats

- Prevent duplicate logins via `player.kick` events
- Heartbeat detection to prevent ghost connections
- Input validation on REST endpoints

---

10. System Observability

- Metrics:
  - Active WebSocket connections
  - Matchmaking queue size
  - Messages per second
  - Failed event count
- Logging:
  - All critical events logged with `userId` and timestamp
  - Error logs include stack traces
- Health checks:
  - `/health` endpoint per service
  - Optional `/metrics` for Prometheus

---

11. Summary

The platform is a **loosely-coupled, event-driven multiplayer system** with:

- **Gateway Service** for real-time messaging
- **Player Service** for authoritative identity and presence
- **Matchmaking Service** for fair pairing
- **Game Logic Service** for match execution
- **Redis** for ephemeral state
- **RabbitMQ** for decoupled events

This architecture ensures **scalability, reliability, and real-time responsiveness**, making it suitable for large concurrent player bases.

frontend

A. Event Service (The Gateway)
This service manages the "Pulse."

- On WS Connect:
  - Validate JWT.
  - Publish player.connected to RabbitMQ (so Player Service knows they are online).
- On WS Heartbeat (Ping):
  - Directly update Redis: HSET presence:{userId} missedHeartbeats 0.
- On WS Disconnect:
  - Directly delete Redis key: DEL presence:{userId}.
  - Publish player.disconnected to RabbitMQ.
- The Janitor (Gateway Loop): Every 20s, the Gateway scans Redis. It increments strikes.
- The Eviction: If strikes >= 3, the Gateway:
  - Directly delete Redis key: DEL presence:{userId}.
  - Publish player.disconnected to RabbitMQ.

B. player Service

- On connect
  - Set Radis {

```
    userId,
      username,
      rating: rating.toString(),
      status: "IDLE",
      missedHeartbeats: "0", // Initialize counter
      lastPulse: Date.now().toString(),
    })
```

- On handleMatchEnded

```
const updateRedis = async (id: string, newRating: number) =>
{
      const key = `presence:${id}`;
      if (await redis.exists(key)) {
        await redis.hset(key, {
          rating: newRating.toString(),
          status: "IDLE",
        });
        await redis.expire(key, PRESENCE_TTL);
      }
    };
```

B. Match making Service
On disconnect

```
const handlePlayerDisconnected = async (data: any) => {

  const { userId } = data;
```

```
  console.log(`[Matchmaking] Cleaning up disconnected player
${userId}`);

  const QUEUE_KEY = "match:queue:ranked";
  const JOIN_TIMES_KEY = "match:join_times";

  await redis
    .pipeline()
    .zrem(QUEUE_KEY, userId)
    .hdel(JOIN_TIMES_KEY, userId)
    .hset(`presence:${userId}`, "status", "IDLE")
    .exec();
};
```

Publihes

```
  publishEvent("matchmaking.left", {
    userId,
    queue: "ranked",
  });

  publishEvent("matchmaking.joined", {
    userId,
    rating,
    queue: "ranked",
  });
    await redis.pipeline()
      .hset(`presence:${p1}`, "status", "IN_GAME")
      .hset(`presence:${p2}`, "status", "IN_GAME")
      .hdel(JOIN_TIMES_KEY, p1, p2)
      .exec();

    await publishEvent("match.created", {
      matchId,
      players: [p1, p2],
      mode: "ranked"
    });
```

Player Service

Player Service – Detailed Service Documentation

Architecture, Events, Runtime Behavior

---

1. Service Role in the System

The Player Service is the **identity, presence, and player-state authority** in the multiplayer system.

Responsibilities:

- Authenticate and uniquely identify players
- Manage player profiles and competitive metadata
- Track player online/offline state reliably
- Notify other services when player state changes

It acts as both a **producer and consumer** of domain events within the system.

---

2. Core Domain Entities (Conceptual Only)

> Entities are for understanding; persistence details are omitted.

- **Player** – Authenticated user identity
- **PlayerProfile** – Public info (username, avatar, bio)
- **PlayerStats** – Competitive stats (wins, losses, rating)
- **PlayerPresence** – Runtime state (online, idle, in-game, queued)

---

3. Internal Architecture

The service follows a **layered + event-driven design**:

Components:

- HTTP API
  - Controllers
  - Middleware (Authentication)
- Services (Business Logic)

- Event Consumers
- Event Publishers
- Redis (Presence Cache)
- RabbitMQ (Message Broker)

---

4. Authentication & Identity Management

Authentication Model:

- Stateless JWT-based tokens containing only `userId`
- Tokens validated via middleware on protected routes

Responsibilities:

- Secure registration with hashed passwords
- JWT login and refresh
- Token verification for internal services
- Client-managed logout (stateless)

Benefits:

- Supports horizontal scaling without shared session storage

---

5. Presence Management (Redis-Based)

Purpose: Real-time, ephemeral player state (not permanent).

Data Model (Redis):

- Key: `presence:{userId}` (Hash)
  - `userId`, `username`, `rating`
  - `status` (IDLE | QUEUED | IN_GAME)
  - `missedHeartbeats`
- `online_players` (Set) tracks active players

---

6. Event-Driven Communication

6.1 RabbitMQ Integration

- Topic exchange: `events`

- Dedicated queue: `player.events.queue`
- Routing keys:
    - `player.connected`
    - `player.disconnected`
    - `player.heartbeat`
    - `match.ended`

Subscribed Events & Purpose:

Event	Purpose
player.connected	Initialize presence
player.disconnected	Cleanup presence
player.heartbeat	Maintain connection
match.ended	Update competitive stats

6.2 Event Consumption Flow

1. RabbitMQ delivers a message
2. Message parsed into domain event
3. Dispatcher routes to appropriate handler
4. Business logic executes
5. Message is acknowledged (ACK)
6. Failed events are rejected (NACK) without requeue

Ensures **at-least-once processing** with controlled failure handling.

---

7. Presence Lifecycle

Player Connection

- Triggered by `player.connected`
- Fetch player data
- Initialize Redis presence

- Preserve existing state if already in game
- Add to `online_players`

Player Heartbeat

- Triggered periodically by client/gateway
- Resets `missedHeartbeats`
- Confirms player responsiveness

Player Disconnection

- Triggered by explicit disconnect, heartbeat timeout, or manual cleanup
- Actions: remove Redis presence, remove from `online_players`, notify other services

---

8. Presence Janitor (Self-Healing)

Purpose: Detect and remove stale/crashed sessions.

Behavior:

- Runs every 20 seconds
- Iterates over online players
- Increments `missedHeartbeats`
- Evicts players exceeding threshold

Eviction Actions:

- Remove Redis presence
- Publish `player.disconnected` event
- Enables Matchmaking/Gateway to clean up state

---

9. Match Result Handling

- Triggered by `match.ended` event
- Updates player statistics atomically
- Recalculates Elo rating
- Syncs Redis presence rating
- Sets player status back to IDLE

Decouples game resolution from player data ownership.

---

10. Event Publishing Responsibilities

Event	When Emitted
player.disconnected	On heartbeat eviction
player.updated (future)	Profile or stats changes

Allows other services to react **without tight coupling**.

---

11. Configuration & Environment Management

- Runtime dependencies injected via environment variables:
  - Database connection
  - Redis connection
  - RabbitMQ connection
  - JWT secrets and expiry

Benefits:

- Environment isolation
- Secure secret handling
- Supports containerized deployment

---

12. Fault Tolerance & Reliability

- RabbitMQ connection retry loops
- Message acknowledgments
- Redis-based ephemeral state
- Database transactions for critical updates
- Idempotent event handling

---

13. Current Player Service Capabilities

- ✅ Stateless authentication

- ✅ Event-driven presence tracking
- ✅ Redis-backed online state
- ✅ Heartbeat-based fault detection
- ✅ Match-result synchronization
- ✅ Decoupled inter-service communication

---

14. Why This Service Is Important

- Identity Provider
- Presence Authority
- Competitive Metadata Manager
- Event Gateway for player state

  Enables the rest of the system to remain **simple, scalable, and loosely coupled**.

Player Service – API Endpoints

Base URL: `/` (via API Gateway or directly in development)

---

1. Authentication API (`/auth`)

Stateless, JWT-based endpoints.

1.1 Register Player

- **POST /auth/register**
- **Auth:** ❌ Not required
- **Request Body:**

```
{

  "username": "player_one",

  "email": "player@example.com",

  "password": "StrongPassword123"

}
```

- **Success (201):**

```
{
  "id": "uuid",
  "email": "player@example.com",
  "profile": { "username": "player_one" },
  "stats": { "rating": 1000 }
}
```

- **Errors:** 400 – Email/username exists or validation error

1.2 Login Player

- **POST /auth/login**
- **Auth:** ❌ Not required
- **Request Body:** email + password
- **Success (200):** `{ "token": "jwt_token_here" }`
- **Errors:** 400 – Invalid credentials

1.3 Refresh Token

- **POST /auth/refresh**
- **Auth:** ✅ Required
- **Success (200):** New JWT
- **Errors:** 401 – Token missing/invalid

1.4 Logout Player

- **POST /auth/logout**
- **Auth:** ✅ Required
- **Success:** `{ "message": "Logged out. Client must delete token." }`

---

2. Player Self-Management API (`/players/me`)

2.1 Get My Data

- **GET /players/me**
- **Auth:** ✅ Required
- **Success:** Full player info (profile + stats)

2.2 Update Profile

- **PUT /players/me/profile**
- **Request Body:** username, avatarUrl, bio
- **Success:** Updated profile

2.3 Update Email

- **PUT /players/me/email**
- **Request Body:** `{ "email": "newemail@example.com" }`
- **Success:** `{ "message": "Email updated successfully" }`

2.4 Update Password

- **PUT /players/me/password**
- **Request Body:** oldPassword, newPassword
- **Success:** `{ "message": "Password updated and sessions cleared" }`

2.5 Delete Account

- **DELETE /players/me**
- **Success:** `{ "message": "Account successfully deleted" }`

---

3. Public Player API (`/players`)

3.1 Search Players

- **GET /players?search=<>&page=&limit=**
- **Success:** Paginated player list

3.2 Get Public Profile

- **GET /players/{id}/profile**
- **Success:** username, avatarUrl, bio

3.3 Get Public Stats

- **GET /players/{id}/stats**
- **Success:** rating, wins, losses, draws

---

4. Security & Access Control

Endpoint Group	Auth Required

/auth/register ❌

/auth/login ❌

/players/me/* ✅

/players/* (public) ❌

API Principles: RESTful, stateless, paginated, validated, private vs public data separated.

Player Service – Event Handling

Role of Events: Enables **loose coupling** with Matchmaking, Game Logic, and Gateway.

---

1. Event Infrastructure

- RabbitMQ with topic exchange `events`
- Queue: `player.events.queue`
- Routing keys:
  - `player.connected`
  - `player.disconnected`
  - `player.heartbeat`
  - `match.ended`

---

2. Event Consumption Lifecycle

1. Connect to RabbitMQ (retry logic)
2. Declare exchange & queue, bind routing keys
3. Consume messages
4. Parse message → dispatch → handler → ACK/NACK

Central Dispatcher: `handleEvents(event)`

- Routes events to correct handlers
- Logs unrecognized events

---

3. Event Types & Handling

Event	Trigger	Handler	Actions	Importance
player.connected	Player connects	Initialize Redis presence	Fetch identity, update Redis, add to `online_players`	Consistent presence
player.heartbeat	Client heartbeat	Reset counter	Reset `missedHeartbeats`	Prevent false disconnects
player.disconnected	Disconnect/timeout/eviction	Cleanup	Remove Redis presence, notify services	Prevent ghost players
match.ended	Game finishes	Update stats	Wins/losses/draws, Elo, Redis rating, status → IDLE	Authoritative player stats

Presence Janitor:

- Periodic background worker
- Evicts unresponsive players
- Publishes `player.disconnected` events

---

4. Event Publishing Flow

- Construct event object: type, data, occurredAt
- Publish with routing key
- Mark message persistent

Error Handling:

- NACK invalid JSON or failed operations
- Avoid requeue loops
- Log errors

Benefits:

- Loose coupling
- Horizontal scalability
- Resilience to partial failures
- Clear player-state ownership

Matchmaking Service

Matchmaking Service Documentation

Overview

Service Type: Core Service
Primary Responsibility: Manages player queues, pairs compatible players, and creates game matches
Status: 🟡 Development
Version: v1.0.0
Last Updated: 2024-01-22

Architecture & Design

Service Context

Architecture Diagram:

Client/API Gateway → Matchmaking Service → Redis → Player Presence & Queue
Matchmaking Service → RabbitMQ → Game Logic Service

- **Design Patterns:**
  - Worker Pattern (Background Matchmaking Worker)
  - Publisher-Subscriber Pattern (Event-Driven)
  - Distributed Authority (Independent match creation per instance)
  - Circuit Breaker / Retry (RabbitMQ connection retries)

API Specification

Endpoints

Method	Endpoint	Description	Auth Required	Rate Limit
POST	/join	Join matchmaking queue	✅ JWT	30/min
POST	/leave	Leave matchmaking queue	✅ JWT	30/min

Join Queue Request Example:

- Method: POST /join
- Header: Authorization: Bearer <JWT_TOKEN>
- Body: empty

Success Response:

- { "message": "Joined queue", "rating": 1500 }

Error Response (Already in Queue):

- { "error": "You are already searching for a match." }

Leave Queue Request Example:

- Method: POST /leave
- Header: Authorization: Bearer <JWT_TOKEN>

Success Response:

- { "message": "Left queue" }

Error Codes:

- 400: Bad Request – Invalid input or business logic violation
- 401: Unauthorized – Missing or invalid JWT token
- 500: Internal Server Error – Redis/RabbitMQ connectivity issues

Event-Driven Communication

Published Events

Event	Channel/Topic	Payload	Description
matchmaking.joined	events	{userId, rating, queue}	Player joined queue
matchmaking.left	events	{userId, queue}	Player left queue
match.created	events	{matchId, players[2], mode}	Match successfully created

Consumed Events

Event	Channel/Topic	Handler	Action
player.disconnected	events	handlePlayerDisconnected	Remove player from queue and cleanup

Data Management

Redis Structures

- QUEUE_KEY = "match:queue:ranked" (Sorted Set: playerId → rating)
- JOIN_TIMES_KEY = "match:join_times" (Hash: playerId → join timestamp)
- PRESENCE_KEY = "presence:{userId}" (Hash: Player session data)

Presence Hash Fields

- status: "IDLE" | "QUEUED" | "IN_GAME"
- rating: numeric (string)
- additional metadata can be added

Data Flow

1. Client sends /join with JWT
2. Service fetches presence from Redis
3. Service updates Redis (status = QUEUED, adds to queue, join time)
4. Service publishes "matchmaking.joined" event to RabbitMQ
5. Matchmaking Worker polls queued players every 1.5s
6. Worker finds partners using rating ± dynamic range
7. Matches created and published as "match.created"
8. Players removed from queue

Matchmaking Algorithm

Dynamic Range Expansion

- Wait <10s: ±50 rating
- Wait 10–20s: ±100 rating
- Wait >20s: ±200 rating

Partner Selection

- Get up to 8 candidates within rating range
- Only match with players whose status = QUEUED
- If no valid partner found, keep player in queue

Worker Configuration

- Polling Interval: 1.5s
- Batch Size: up to 40 players per iteration
- Concurrent Matching: Multiple matches per loop

- Cleanup: Remove stale or disconnected players

Dependencies & Integration

External Services

- Redis: Player queue, presence, ratings
- RabbitMQ: Event bus for communication
- Player Service: JWT validation and player data

Internal Dependencies

- express, redis, amqplib, jsonwebtoken, uuid, dotenv

Configuration

Environment Variables

- PORT = 3000
- REDIS_URL = redis://localhost:6379
- RABBITMQ_URL = amqp://localhost:5672
- JWT_SECRET = your-secret-key
- JWT_EXPIRES_IN = 1d
- EVENTS_EXCHANGE = events
- MATCHMAKING_QUEUE = matchmaking.events.queue

Development & Operations

Local Development

1. cd services/matchmaking-service
2. npm install
3. docker-compose up -d redis rabbitmq
4. npm run dev

Health & Metrics (Planned)

- /health → {"status":"healthy","redis":true,"rabbitmq":true,"queueSize":42}
- /metrics → Worker stats, matches/minute

Deployment & Scaling

- Docker Compose: Multiple replicas for high availability
- Horizontal scaling: Multiple instances independently run matches
- Vertical scaling: Increase Redis connections
- Regional queues for cross-region matchmaking

Security

- JWT token authentication
- Token expiration handled
- Rate limiting and input validation planned
- Player ratings in Redis (consider encryption)

Troubleshooting

- Redis Connection: "Presence not found" → check URL/connectivity
- RabbitMQ Timeout → verify health and credentials
- Memory Leak → increase worker interval
- Stale Players → ensure `player.disconnected` events fire

Debug Commands

- redis-cli zcard "match:queue:ranked"
- redis-cli zrange "match:queue:ranked" 0 -1 WITHSCORES
- rabbitmqadmin get queue="matchmaking.events.queue"
- docker logs matchmaking-service --tail 100

Game Logic Service

Tic-Tac-Toe Game Logic Service Design

1. Data Storage Strategy

Hot Storage (Redis) – Active Matches

This data exists only for the duration of the match.

- **game:state:{matchId} (Hash):**
  - board: A string/JSON representation of the 3x3 grid (e.g., ["X", "", "O", ...]).
  - turn: The userId of the player whose turn it is.
  - version: A counter incremented with every move to prevent race conditions.
  - players: JSON array of the two userIds.
  - symbols: Mapping of userId to X or O.
  - lastMoveTime: Timestamp for timeout detection.
-
- **player:active_match:{userId} (String):**
  - Maps a player to their current matchId. This allows the service to quickly check if a player is already "busy" before starting a new game.
-

Cold Storage (MongoDB) – Historical Archive

Used for record-keeping and UI history.

- **Matches Collection:**
  - matchId, players[], winnerId (or null for draw).
  - moveHistory: Array of { userId, position, timestamp }.
  - finalBoard: The state of the board at the end.
  - reason: "COMPLETED", "FORFEIT", or "TIMEOUT".
  - endedAt: Timestamp.
-

---

2. Event Handling & Workflows

A. Initialization (match.created)

When the Matchmaker finds a pair:

1. **Verification:** The service checks the Redis **Presence Hash** for both players.
2. **State Setup:**
   - Generates a 3x3 empty board.
   - Randomly assigns X and O.

    - Assigns the first turn to the X player.
3.
4. **Redis Write:** Saves the game:state and sets the player:active_match pointers.
5. **Error Handling:** If any step fails (e.g., player is suddenly offline or Redis is down), it emits **match.failed** so the Matchmaker can re-queue the players.

B. Move Processing (game.move)

1. **Hydrate:** Pull the game:state from Redis using the matchId.
2. **Validate:**
    - Is the senderId one of the two players in this match?
    - Is it currently the senderId's turn?
    - Is the chosen square (0-8) currently empty?
3.
4. **Process:**
    - Update the board array with the player's symbol.
    - **Check Win Condition:** 8 possible lines (3 rows, 3 columns, 2 diagonals).
    - **Check Draw Condition:** Are all 9 squares filled with no winner?
5.
6. **Finalize:**
    - **If Ongoing:** Update Redis state, toggle the turn to the opponent, and publish **turn.completed**.
    - **If Over:** Transition to the **Settlement Phase**.
7.

C. Settlement & Archiving (Game Over)

When a game ends (Win, Draw, or Forfeit):

1. **Archive:** Build a summary object and save it to **MongoDB**.
2. **Cleanup:** Delete the match keys and player pointers from **Redis**.
3. **Release:** Publish **match.ended** to RabbitMQ.
    - *Target:* **Player Service** hears this to update ELO and wins/losses.
    - *Target:* **Gateway Service** hears this to show the "Game Over" screen.
4.

D. Resilience (Disconnects & Forfeits)

- **player.disconnected:** If a player leaves the Gateway, the Game Service hears the event. It doesn't end the game immediately; instead, it starts a **Grace Timer** (e.g., 60s).
- **game.forfeit:** If a player clicks "Surrender," the game ends immediately. The service marks the other player as the winner and proceeds to the Archive step.

---

3. API & Security

The History Route

- **Endpoint:** GET /history
- **Protection:** Requires a valid JWT.
- **Logic:** The service queries **MongoDB** for all matches where the userId (from the JWT) is in the players array. Results are paginated.

---

4. Why this Design works

1. **Stateless Execution:** The Game Logic instance doesn't keep the board in its local memory. It pulls from Redis for every move. This means you can have 20 instances of this service, and any of them can process any move for any match.
2. **Zero-Trust Validation:** Every move is checked against the rules and the current turn. A player cannot "force" a move or play twice in a row.
3. **Optimized Database usage:** MongoDB is only hit **once per game** (at the end). This prevents the database from becoming a bottleneck during high-speed play.
4. **Presence Integration:** By verifying against the **Presence Hash**, you ensure that you never waste server resources setting up a game for a player who has already closed their app.

Time management is a critical "invisible" layer in multiplayer games. In a stateless architecture, you can't rely on simple setTimeout() functions because if the server instance restarts, the timer is lost.

Instead, you must use **Distributed Time Management** using **Redis Sorted Sets**.

Game Logic – Time Management Design

1. The "Deadman's Switch" Mechanism

We use a Redis Sorted Set specifically for timers.

- **Key:** game:timers
- **Member:** matchId
- **Score:** The **Timestamp** (Unix epoch) when the current turn **must end**.

How it works:

1. **Turn Starts:** When a match starts or a player moves, calculate the deadline: Now + 30 seconds.
2. **Set Timer:** ZADD game:timers <timestamp> <matchId>.
3. **The Watchdog:** A background worker runs every 1 second and asks Redis: *"Give me all matchIds where the score is less than the current time."*
    - Command: ZRANGEBYSCORE game:timers 0 <CurrentTimestamp> LIMIT 0 10.

2. The Workflows

A. Normal Turn Completion

When a player makes a valid move within the time limit:

1. **Cancel Old Timer:** ZREM game:timers {matchId}.
2. **Process Move:** Update the board in Redis.
3. **Start New Timer:** ZADD game:timers <Now + 30s> {matchId} for the opponent.

B. Turn Timeout (The Executioner)

If the Watchdog finds an expired matchId:

1. **Atomic Claim:** Use ZREM to pull the matchId from the set. If successful, this instance "owns" the timeout logic.
2. **Identity:** Fetch game:state:{matchId} to see whose turn it was.
3. **Forfeit:**
    - Mark the current player as the **Loser by Timeout**.
    - Save the result to **MongoDB**.
    - Publish **match.ended** (Reason: TIMEOUT).
    - Clean up all Redis keys.

3. Handling Disconnection Grace Periods

Players hate losing a game because their Wi-Fi flickered for 2 seconds. You need a **Reconnection Window**.

- **Logic:**
    1. Gateway emits **player.disconnected**.
    2. Game Service receives this and checks if the player is in a match.
    3. If yes, the service sets a **game:reconnect:{userId}** timer for **60 seconds**.
    4. The main game timer continues to run.
-
- **Scenario 1 (Player Returns):** They reconnect, the game:reconnect timer is deleted, and they continue their turn.
- **Scenario 2 (Window Expires):** The player is automatically forfeited, even if they still had 10 seconds left on their turn clock.

4. Strategic Time Features

The "Tick" Synchronization

Every time the Gateway sends a match.created or turn.completed event to the client, it includes the **expiresAt** timestamp.

- *Benefit:* The client's UI can show a perfectly synchronized countdown timer (e.g., "14 seconds left") because both the server and client are looking at the same absolute timestamp.

Anti-Stalling

If a player repeatedly waits until the very last second to move, you can track "Total Bank Time" (similar to Chess).

- Add a remainingBankTime field to the Redis game:state. If it hits 0, they lose the game regardless of the individual turn timer.

5. Why this is "High-Level" Architecture

1. **Stateless Resilience:** If your Game Logic Service crashes, the timers are safe in Redis. When the service comes back online, the Watchdog immediately finds all the matches that timed out during the crash.
2. **Precision:** It doesn't matter if the network is slow. The deadline is an absolute point in time.
3. **Fairness:** The "Atomic Claim" (ZREM) ensures that if you have 10 instances of the Game Service running, only **one** instance will process the timeout for a specific match.

Summary of Redis Commands for Time:

- ZADD: Start or Reset a timer.
- ZREM: Cancel a timer (on move or game end).
- ZRANGEBYSCORE: Find players who are too slow.

To ensure a high-performance, decoupled Tic-Tac-Toe system, we will use a **Topic Exchange** (named events). This allows the Game Logic Service to "shout" updates that multiple services (Gateway, Player, Matchmaker) can listen to simultaneously.

Here are the defined **RabbitMQ Routing Keys** for the Game Logic Service:

---

1. Inbound Routing Keys (What the Game Logic listens to)

The Game Logic Service binds its own private queue to these keys to receive instructions and system updates.

Routing Key	Source Service	Purpose
match.created	Matchmaking	Trigger initialization of a new 3x3 board in Redis.
game.cmd.move	Gateway	A player attempted to click a square (0-8).
game.cmd.forfeit	Gateway	A player clicked the "Surrender" button.
player.disconnected	Gateway	Start the **60s Reconnection Grace Timer** in Redis.
player.connected	Gateway	Cancel the Reconnection Grace Timer (Player returned).

---

2. Outbound Routing Keys (What the Game Logic publishes)

These events are broadcast to the exchange to trigger UI updates or data persistence in other services.

Routing Key	Target Service	Purpose
game.event.started	Gateway	Notify both players of their symbols (X/O) and whose turn it is first.

game.event.turn	Gateway	Push the updated board array and the nextTurn userId to the players.
game.event.invalid	Gateway	Tell a specific player: "Invalid Move" or "Not your turn."
match.failed	Matchmaker	Initialization failed. Tell Matchmaker to re-queue innocent players.
match.ended	Player / Gateway	Game is over (Win/Draw/Timeout). Triggers ELO updates and "Victory" screens.

---

3. Targeted Routing (Unicast Integration)

Since we implemented **Instance-Affinity** for the Gateway, the Game Logic Service should be "Affinity Aware." When it sends a message meant for a specific player, it should use the target's instanceId (stored in the Redis Presence Hash).

Example Workflow:

1. **Game Logic** processes a move for Player B.

Game Logic looks at Redis presence:PlayerB
→\rightarrow→

2. finds instanceId: gateway-01.
3. **Game Logic** publishes with routing key: game.event.turn.gateway-01.
4. **Result:** Only gateway-01 processes the message, saving CPU cycles on all other Gateway instances.

---

4. Anatomy of a "Move" Payload

To keep the system stateless and easy to debug, every outbound event should follow a consistent "Envelope":

Event: game.event.turn

code JSON

downloadcontent_copy

expand_less

```
{
  "type": "game.event.turn",
  "data": {
    "matchId": "uuid-123",
    "board": ["X", "O", "", "", "X", "", "", "", ""],
    "lastMoveBy": "user-A",
    "nextTurn": "user-B",
    "expiresAt": 1715000500, // Unix timestamp for the 30s limit
    "version": 5
  },
  "occurredAt": "2024-05-20T12:00:00Z"
}
```

---

5. Why these keys create a "Self-Healing" Game

- **Decoupling:** The Game Logic Service doesn't need to know *how* to send a WebSocket message. It just publishes to game.event.turn, and the Gateway handles the rest.
- **Integrity:** By consuming player.disconnected, the Game Service can distinguish between a player **stalling** (still connected but not moving) and a player **crashing** (needs a grace period).
- **Replay Ready:** Because every move is published with a unique key, you could attach a "Replay Service" that records every single game.event.turn to a separate database for future playback.

We are now defining the **Hot State (Redis)** and the **Cold State (MongoDB)**.

In this architecture, Redis handles the high-pressure, millisecond-latency gameplay, while MongoDB provides the rich, queryable history for the players and the business.

---

1. Redis Schema: The "Live Match" Store

We use three specific data structures in Redis to manage an active game of Tic-Tac-Toe.

A. The Match Hash

Key: game:match:{matchId}

Field	Type	Description
players	JSON String	["user_A_id", "user_B_id"]
board	JSON String	["", "X", "", "", "O", "", "", "", ""] (Index 0-8)
turn	String	userId of the player who moves next.
symbols	JSON String	{"user_A_id": "X", "user_B_id": "O"}
version	Integer	Incremented every move to prevent race conditions.
status	String	ACTIVE, PENDING_RECONNECT.
expiresAt	Timestamp	When the current turn expires.

B. The Player Lookup (Index)

Key: player:match_map:{userId}

- **Value:** {matchId}
- **Purpose:** Allows the Game Logic Service to instantly verify if a player is already in a game before allowing a move or a new match.

C. The Distributed Timer (Sorted Set)

Key: game:timers

- **Score:** Unix Timestamp of the deadline.
- **Member:** {matchId}
- **Purpose:** A background worker scans this to find matches that have timed out.

---

2. MongoDB Schema: The "Match History" Archive

When the game ends, the state is "Settled" and moved to MongoDB. We use a flat document structure for fast player-based querying.

Collection: matches

code JSON

downloadcontent_copy

expand_less

```
{
  "_id": "ObjectId",
  "matchId": "uuid-string",
  "gameType": "tic-tac-toe",
  "participants": [
    { "userId": "uuid-A", "symbol": "X", "username": "DragonSlayer" },
    { "userId": "uuid-B", "symbol": "O", "username": "ShadowNinja" }
  ],
  "result": {
    "winnerId": "uuid-A", // null if Draw
    "reason": "COMPLETED", // COMPLETED, FORFEIT, TIMEOUT
    "winningLine": [0, 1, 2] // The indices that formed the win
  },
  "history": [
    { "userId": "uuid-A", "position": 0, "time": "2024-05-20T12:00:01Z" },
    { "userId": "uuid-B", "position": 4, "time": "2024-05-20T12:00:05Z" }
  ],
  "finalBoard": ["X", "X", "X", "", "O", "", "", "O", ""],
  "durationSeconds": 45,
  "startedAt": "2024-05-20T12:00:00Z",
  "endedAt": "2024-05-20T12:00:45Z"
}
```

---

3. The "State Handover" Workflow (Redis

→\rightarrow→

MongoDB)

This is the most critical logic in the service. It ensures no data is lost during the transition.

1. **Detection:** A move is made that completes a line, or a timer expires.
2. **Locking:** The Game Service calls HSET game:match:{matchId} status FINISHED. No more moves are accepted for this ID.
3. **Bundle Building:** The service gathers all data from the Redis Match Hash and the move history list.
4. **Database Write:** The service performs an **insert** into MongoDB.
5. **Event Emission:**
    - Publishes match.ended to RabbitMQ (Payload includes the winner).
    - **Player Service** picks this up to update Postgres ELO.
6.
7. **Cleanup:**
    - DEL game:match:{matchId}
    - DEL player:match_map:{playerA}
    - DEL player:match_map:{playerB}
    - ZREM game:timers {matchId}
8.

---

4. Key Performance Features

- **Indexing MongoDB:** We will index the participants.userId and endedAt fields. This makes the GET /history route lightning fast:
    - *Query:* db.matches.find({"participants.userId": myId}).sort({endedAt: -1}).
-
- **TTL in Redis:** Even if a game crashes and the cleanup fails, we set a **2-hour TTL** on all Redis game keys. This prevents memory leaks.
- **Atomic Updates:** Move validation uses a Redis **Lua Script** to ensure that "Check Turn" and "Write Move" happen in a single atomic step, preventing two moves from being written at the same time.

✅ Summary of Data Flow

- **Redis:** Authoritative state during play. Optimized for **Write Speed**.
- **MongoDB:** Authoritative state after play. Optimized for **Querying/History**.

Gateway Service

Gateway Service – Detailed Service Documentation

Architecture, Events, WebSocket & Runtime Behavior

---

1. Service Role in the System

The **Gateway Service** acts as the **real-time communication hub** for the multiplayer platform.

Responsibilities:

- Manage **WebSocket connections** with players
- Authenticate users via JWT tokens
- Relay events between **players**, **matchmaking**, and **other services**
- Track **online/offline presence** through heartbeat monitoring
- Publish/consume **domain events** via RabbitMQ
- Ensure session consistency across multiple devices

It does **not** handle game logic or player stats directly; those are delegated to other services.

---

2. Architecture Overview

The Gateway Service is structured as a **hybrid HTTP + WebSocket + Event-driven service**.

Core Components:

- **Express HTTP API**
  - Serves routes like `/health`
- **WebSocket Server**
  - Handles live player connections
  - Maintains `userSockets` map for active sessions
- **RabbitMQ Event Bus**
  - Publishes and consumes events for inter-service communication
- **Redis**
  - Tracks ephemeral state like match participants and heartbeat info
- **Gateway Service Layer**
  - Handles connection, disconnection, heartbeat, and private chat logic

3. Configuration & Environment

Environment Variables:

```
PORT=4000
REDIS_URL=redis://localhost:6379
RABBITMQ_URL=amqp://localhost:5672
JWT_SECRET=your-secret-key
JWT_EXPIRES_IN=1d
WS_PATH=/ws
```

Service Config (runtime):

```
export const config = {
  eventsExchange: "events",
  gatewayQueue: `gateway.queue.${instanceId}`,
  gatewayRoutingKeys: ["match.started","player.kick","chat.private"],
  port: Number(process.env.PORT) || 4000,
  redisUrl: process.env.REDIS_URL!,
  rabbitmqUrl: process.env.RABBITMQ_URL!,
  jwtSecret: process.env.JWT_SECRET!,
  jwtExpiry: process.env.JWT_EXPIRES_IN ?? "1d",
  wsPath: process.env.WS_PATH!,
};
```

> Each instance has a **unique queue** using `hostname + randomId` to avoid conflicts.

---

4. WebSocket Management

4.1 Connection Lifecycle

- On connection:
  - JWT token is extracted from URL query
  - `authenticateWS(token)` verifies token
  - Player `userId` is assigned to socket
  - Socket is marked `isAlive = true`
  - Added to `userSockets` map
- On heartbeat (`pong`):
  - Resets `isAlive` and triggers `handleHeartbeat(userId)`
- On message:
  - Only supports structured JSON messages (e.g., `CHAT`)

    - Validates match participation before sending private messages
- On close/error:
    - Removes socket from `userSockets`
    - Triggers `handleDisconnect(userId)`

4.2 Heartbeat & Watchdog

- Interval: 30 seconds
- Pings all connected clients
- Terminates unresponsive sockets (`isAlive = false`)
- Logs terminated connections for observability

4.3 Sending Messages

sendToUser(userId: string, payload: any)

- Sends JSON payload if socket exists and is OPEN
- Returns `true` if successful, `false` otherwise

---

5. RabbitMQ Event Handling

5.1 Initialization

- Connects to RabbitMQ with retry logic
- Declares exchange `events` (type `topic`)
- Declares durable queue for this gateway instance
- Binds relevant routing keys:
    - `match.started`, `player.kick`, `chat.private`

5.2 Event Consumption

- Each event is parsed and dispatched to `handleEvents(event)`
- Reliable processing with **ack/nack semantics**
- On failure, message is rejected but **not requeued**

5.3 Event Publishing

- `publishEvent(routingKey, payload)` constructs event with timestamp
- Publishes to exchange as **persistent** message
- Logs event emission for monitoring

---

6. Event Types & Handling

Event	Purpose	Handler Logic
player.kick	Kick a player logged in elsewhere	Terminate local WS session and delete from `userSockets`
chat.private	Direct chat between players	Validate match participation, then forward message to recipient via WS
match.created	Notify players of a new match	Send opponentId, matchId, and mode to each player

Ensures **real-time updates** and **security enforcement**.

---

7. Gateway Service Layer

Responsibilities:

- `handleConnect(userId)` – triggers `player.connected` and initial kick
- `handleHeartbeat(userId)` – triggers `player.heartbeat`
- `handleDisconnect(userId)` – triggers `player.disconnected`
- `handlePrivateChat(senderId, recipientId, matchId, text)` – validates participants, publishes `chat.private`

Security Checks:

- Only match participants can send messages
- Logs unauthorized attempts

---

8. HTTP API

Health Endpoint

- `GET /health`
- Returns:

```
{
  "status": "healthy",
  "service": "gateway-service",
  "timestamp": "2026-01-22T...",
  "uptime": 123.45
}
```

Used for Kubernetes / container monitoring

---

9. Type Definitions

```
interface JwtPayload { userId: string; iat?: number; exp?: number }
interface AuthenticatedSocket extends WebSocket {
  userId?: string;
  isAlive: boolean;
  wasKicked: boolean;
}
type Event<T = any> = { type: string; data: T; occurredAt: string }
```

Provides type safety across WebSocket, RabbitMQ, and HTTP handling.

---

10. Reliability & Fault Tolerance

- Retry loop for RabbitMQ connections
- Watchdog terminates unresponsive WS sessions
- NACK failed events without requeue
- Unique queues per instance prevent race conditions
- Logging for debugging event flow and session issues

---

11. Summary

The **Gateway Service** provides:

- Real-time player connections via **WebSocket**
- JWT-based **stateless authentication**
- **Event-driven communication** with other services
- Heartbeat-based **presence monitoring**
- Secure **chat and match notifications**
- Fault-tolerant RabbitMQ integration

Acts as the **primary entry point** for live multiplayer traffic and a bridge between players and backend services.

Gateway Service – Inter-Service Communication

---

1. Overview of Service Communication

The **Gateway Service** is the central **real-time communication hub**. It does not store authoritative game or player data but coordinates with other services to relay state and events.

Primary communication types:

Method	Purpose	Direction	Reliability
RabbitMQ Events	State changes, match notifications, player kicks, chat messages	Bi-directional	Persistent, at-least-once delivery
WebSocket (WS)	Real-time messages to clients	Outbound only	Ephemeral, heartbeat monitored
HTTP API	Health checks, optional admin commands	Inbound	Standard HTTP reliability

---

2. Event-Driven Communication via RabbitMQ

2.1 Event Flow

The Gateway subscribes to a **topic exchange** `events` and maintains a **dedicated queue per instance**.

Subscribed Routing Keys:

- `player.kick` – From Player Service when a new login invalidates an old session
- `chat.private` – From Gateway or other services for relaying private messages
- `match.created` – From Matchmaking Service when a new match is formed
- `match.started` – From Game Logic Service when a match begins

Published Routing Keys:

- `player.connected` – Notifies Player Service of a new WS connection
- `player.heartbeat` – Sends periodic heartbeat to Player Service
- `player.disconnected` – Notifies Player Service of a disconnect
- `chat.private` – Forward chat messages to recipients
- `player.kick` – Initiated by Gateway to enforce session consistency

2.2 Communication Examples

2.2.1 Player Connection

1. User connects via WebSocket and passes JWT
2. Gateway validates JWT → assigns `userId`
3. Gateway publishes `player.connected` event:

```
{
  "type": "player.connected",
  "data": { "userId": "abc123" },
  "occurredAt": "2026-01-22T12:00:00Z"
}
```

4. Player Service receives event → updates Redis presence
5. Matchmaking Service may subscribe to presence updates to add the player to queues

---

2.2.2 Match Creation Notification

1. Matchmaking Service creates a match → publishes `match.created` event:

```
{
```

```
  "type": "match.created",
  "data": {
    "matchId": "match-456",
    "players": ["abc123", "xyz789"],
    "mode": "ranked"
  },
  "occurredAt": "2026-01-22T12:01:00Z"
}
```

2. Gateway receives event → forwards via WebSocket to all participants:

```
{
  "type": "MATCH_CREATED",
  "data": {
    "matchId": "match-456",
    "opponentId": "xyz789",
    "mode": "ranked"
  }
}
```

2.2.3 Private Chat Relay

1. Player sends chat via WebSocket: `{ type: "CHAT", to: "xyz789", matchId: "...", text: "hello" }`
2. Gateway verifies both players are in the same match (Redis: `match:participants:{matchId}`)
3. Gateway publishes `chat.private` event:

```
{
  "type": "chat.private",
```

```
  "data": { "from": "abc123", "to": "xyz789", "matchId": "...", "text": "hello" },
  "occurredAt": "2026-01-22T12:02:00Z"
}
```

4. Recipient receives chat message via WebSocket in real-time

---

2.3 Security & Validation

- Only players present in a match can send messages
- Tokens are validated before any event triggers (`authenticateWS`)
- Unauthorized actions are logged and ignored

2.4 Reliability Considerations

- RabbitMQ ensures **at-least-once delivery**
- Gateway **acks** events only after successful handling
- WS connections are **monitored with heartbeat**; unresponsive sockets trigger `player.disconnected`
- Persistent queues prevent message loss during Gateway restarts

---

3. Inter-Service Communication Diagram

```
graph TD
    PlayerClient -->|WebSocket| Gateway
    Gateway -->|player.connected / disconnected| PlayerService
    Gateway -->|player.heartbeat| PlayerService
    Gateway -->|match.created / match.started| PlayerClient
    MatchmakingService -->|match.created| Gateway
    GameLogicService -->|match.started / match.ended| Gateway
    Gateway -->|chat.private| Gateway
    Gateway -->|player.kick| PlayerService
```

Gateway --> Redis

Key Points:

- Gateway mediates **all real-time messages** to clients
- Player Service owns **authoritative presence**
- Matchmaking Service owns **match logic**
- Game Logic Service owns **match progression**
- Redis provides **ephemeral state validation**

---

4. Summary

The Gateway Service ensures **loose coupling** between clients and backend services by:

- Subscribing only to relevant RabbitMQ events
- Publishing user connection, heartbeat, disconnection, and chat events
- Forwarding critical updates to clients via WebSocket
- Maintaining **session consistency**, preventing ghost connections, and enforcing single-login per user

This design allows **horizontal scaling**, real-time responsiveness, and **secure event-driven coordination** across services.

ToDO list

ToDO list

- Make frontend
- Move deployment to k9
- Add the rest of player route to postman
- Test draw

Tab 8

WebSocket API Documentation (Frontend)

1. Connection

URL

ws://<API_HOST><WS_PATH>?token=<JWT>

- WS_PATH → config.wsPath (example: /ws)
- token → **JWT access token** (same token used for REST)

2. Connection Lifecycle

On Connect

After a successful connection & authentication, the server immediately sends:

```
{
  "type": "CONNECT_SYNC",
  "data": { ...currentState }
}
```

This contains:

- active matches
- queue state
- any recoverable game state

👉 Frontend should **hydrate UI from this event.**

Heartbeat (Automatic)

- Server sends ping every **30 seconds**

- Browser WS automatically replies with pong
- No frontend handling required unless you want logging

If heartbeat fails → connection is terminated.

3. Client → Server Messages

All client messages **must be valid JSON**.

Common Shape

```
{
  "type": "MESSAGE_TYPE",
  "payload": { ... }
}
```

3.1 Private Chat

```
{
  "type": "CHAT",
  "to": "userId",
  "matchId": "matchId",
  "text": "Hello 👋"
}
```

✅ Server response:

```
{
  "type": "ACK",
  "data": {
    "originalType": "CHAT",
    "success": true
  }
}
```

3.2 Game Move

```
{
  "type": "GAME_MOVE",
  "payload": {
    "matchId": "matchId",
    "move": 4
  }
}
```

3.3 Forfeit Game

```
{
  "type": "GAME_FORFEIT",
  "payload": {
    "matchId": "matchId"
  }
}
```

3.4 Manual Sync Request

If frontend reconnects or suspects desync:

```
{
  "type": "SYNC_REQUEST"
}
```

Server replies:

```
{
  "type": "SYNC_RESPONSE",
  "data": { ...currentState }
}
```

4. Server → Client Events

All server messages follow:

```
{
  "type": "EVENT_TYPE",
  "data": { ... },
  "timestamp": "ISO-8601"
}
```

4.1 Matchmaking

Joined Queue

```
{
  "type": "QUEUE_JOINED",
  "data": {
    "userId": "..."
  }
}
```

Left Queue

```
{
  "type": "QUEUE_LEFT",
  "data": {
    "userId": "..."
  }
}
```

4.2 Match Lifecycle

Match Created

```
{
  "type": "MATCH_CREATED",
  "data": {
    "matchId": "matchId",
    "opponentId": "userId",
    "mode": "RANKED"
  }
}
```

Game Started

```
{
  "type": "GAME_STARTED",
  "data": {
    "matchId": "matchId",
    "mySymbol": "X",
    "opponentId": "userId",
    "turn": "X",
    "expiresAt": 1700000000
  }
}
```

Game Turn Update

```
{
  "type": "GAME_TURN",
  "data": {
    "matchId": "matchId",
    "board": ["X", "", "O", "", "", "", "", "", ""],
    "nextTurn": "O",
    "isMyTurn": false,
    "expiresAt": 1700000000
  }
}
```

Invalid Move

```
{
  "type": "INVALID_MOVE",
  "data": {
    "matchId": "matchId",
    "reason": "Cell already occupied"
  }
}
```

Game Over

```
{
  "type": "GAME_OVER",
  "data": {
    "matchId": "matchId",
    "result": "WIN",
    "reason": "CHECKMATE",
    "finalBoard": ["X", "O", ...]
  }
}
```

Match Error

```
{
  "type": "MATCH_ERROR",
  "data": {
    "reason": "Opponent disconnected"
  }
}
```

4.3 Chat

Incoming Chat Message

```
{
  "type": "CHAT_MESSAGE",
  "data": {
    "from": "userId",
    "to": "userId",
    "text": "gg!",
    "matchId": "matchId",
    "sentAt": "ISO-8601"
  }
}
```

5. System Messages

Acknowledgment (ACK)

Sent after **every valid client action**:

```
{
  "type": "ACK",
  "data": {
    "originalType": "GAME_MOVE",
    "success": true
  }
}
```

Error

```
{
  "type": "ERROR",
  "data": "Invalid message format"
}
```

Forced Logout (Multi-device)

If user logs in elsewhere:

```
{
  "type": "DISCONNECTED",
  "message": "You have been logged in from another device."
}
```

The connection is then closed.

Player API Documentation

Authentication

All protected endpoints require:

Authorization: Bearer <JWT_TOKEN>

JWT payload:

```
{
  "userId": 2,
  "iat": 1765978046,
  "exp": 1766064446
}
```

1. Register Player

POST /api/auth/register

Creates a new player account.

Validation Rules (Zod)

Field	Rules
username	string, 3–20 characters
email	valid email
password	8–72 characters

Request

```
{
  "username": "Alice",
  "email": "alice@test.com",
  "password": "password123"
}
```

Success (201)

```
{
  "id": 2,
  "email": "alice@test.com",
  "profile": {
    "username": "Alice"
  },
  "stats": {
    "rating": 1000
  }
}
```

Errors

Status	Response
400	{ "error": "Username already taken" }

400	{ "error": "Email already registered" }

2. Login

POST /api/auth/login

Validation

Field	Rules
email	valid email
password	min 1 char

Success

```
{
  "token": "jwt.token.here"
}
```

Errors

Status	Meaning
401	Invalid credentials
404	User not found

3. Refresh Token

POST /api/auth/refresh

Generates a new JWT if current token is valid.

Success

```
{
  "token": "new.jwt.token"
}
```

Errors

```
{ "error": "Invalid or expired token" }
```

4. Logout

POST /api/auth/logout

Stateless logout.

Success

```
{
  "message": "Logged out. Client must delete token."
}
```

5. Get My Player State (Core Endpoint)

GET /api/player/me

This is your **most important endpoint**.

It returns:

- Account data
- Profile
- Stats
- Presence
- Matchmaking status
- Live game state

Response Model (Dynamic)

Case 1 – IDLE

```
{
  "userId": 2,
  "email": "alice@test.com",
  "profile": {
    "username": "Alice",
    "avatarUrl": "",
```

```
    "bio": ""
  },
  "stats": {
    "rating": 1450,
    "wins": 10,
    "losses": 3,
    "draws": 1
  },
  "rating": 1450,
  "status": "IDLE",
  "lastOnline": "2026-01-24T10:00:00Z"
}
```


Case 2 – QUEUED


```
{
  "status": "QUEUED",
  "queue": {
    "position": 3,
    "waitTimeSeconds": 42
  }
}
```


Case 3 – IN_GAME


```
{
  "status": "IN_GAME",
  "game": {
    "matchId": "match_98",
    "players": ["2", "7"],
    "board": ["X", "", "", "", "O", "", "", "", ""],
    "turn": "7",
    "mySymbol": "X",
    "status": "ACTIVE",
    "version": 4,
    "expiresAt": 1766069000
  }
}
```


6. Update My Profile

PUT /api/player/me/profile

Validation

Field	Rules

username	optional, 3–20
avatarUrl	valid URL or empty
bio	max 200 chars

Errors

```
{ "error": "Username already taken" }
```

7. Update Email

PUT /api/player/me/email

Validation

Field	Rules
email	valid email

Error

```
{ "error": "Email already in use" }
```

8. Update Password

PUT /api/player/me/password

Validation

Field	Rules
oldPassword	min 6
newPassword	min 6

Error

```
{ "error": "Incorrect current password" }
```

Success


```
{
  "message": "Password updated and all sessions cleared. Please log in
again."
}
```


9. Delete Account

DELETE /api/player/me

Success


```
{
  "message": "Account successfully deleted"
}
```


10. Get Public Profile

GET /api/player/{id}/profile

Success


```
{
  "username": "Bob",
  "avatarUrl": "https://example.com/bob.jpg",
  "bio": "Pro gamer"
}
```


11. Get Public Stats

GET /api/player/{id}/stats


```
{
  "rating": 1600,
  "wins": 25,
  "losses": 12,
  "draws": 3
}
```

12. Search Players (Paginated)

GET /api/player?search=al&page=1&limit=10

Response


```
{
  "data": [
    {
      "id": 2,
      "email": "alice@test.com",
      "username": "Alice",
      "avatarUrl": "",
      "bio": "",
      "stats": {
        "rating": 1450,
        "wins": 10,
        "losses": 3,
        "draws": 1
      }
    }
  ],
  "meta": {
    "total": 1,
    "page": 1,
    "limit": 10,
    "totalPages": 1
  }
}
```


Matchmaking API

All endpoints require authentication:

Authorization: Bearer <JWT_TOKEN>

Base URL:

{{base_url}}/api/matchmaking

1. Join Matchmaking Queue

Endpoint

POST /join

Description

Adds the authenticated player to the ranked matchmaking queue.

The player will start searching for an opponent.

Request

No body required.

Success Response (200 OK)

```
{
  "message": "Joined queue",
  "rating": 1450
}
```

Error Responses

Status	Response
400	{ "error": "Presence not found. Please ensure your game is connected." }
400	{ "error": "You are already searching for a match." }
400	{ "error": "Cannot queue while in an active match." }
401	{ "error": "Unauthorized" }

2. Leave Matchmaking Queue

Endpoint

POST /leave

Description

Removes the authenticated player from the matchmaking queue.

Request

No body required.

Success Response

```
{
  "message": "Left queue"
}
```

Edge Case Responses

Status	Response
200	{ "message": "Player offline" }
200	{ "message": "Player not in queue" }
401	{ "error": "Unauthorized" }

Matchmaking State

The current matchmaking status is always returned via:

GET /api/player/me

Possible values:

```
{
```

```
  "status": "IDLE"
}
```

```
{
  "status": "QUEUED",
  "queue": {
    "position": 3,
    "waitTimeSeconds": 42
  }
}
```

```
{
  "status": "IN_GAME",
  "game": { ... }
}
```

Client Rules (API Contract)

From the client perspective:

Rule
You must call `/join` before matchmaking
You cannot call `/join` twice
You cannot join while in a game

You can call /leave at any time

UI state comes only from
/player/me

Minimal API Summary (for report)

Method	Endpoint	Purpose
POST	/api/matchmaking/join	Start matchmaking
POST	/api/matchmaking/leave	Stop matchmaking
GET	/api/player/me	Read matchmaking status

Game History API

Base URL:

{{base_url}}/api/history

All endpoints require:

Authorization: Bearer <JWT_TOKEN>

1. Get My Game History (List)

Endpoint

GET /me

Returns a paginated list of completed matches for the authenticated player.

Query Parameters

Parameter	Type	Default	Description
page	number	1	Page number
limit	number	20	Items per page (max 50)
sortBy	string	endedAt	endedAt, durationMs, turnCount
order	string	desc	asc or desc
result	string	-	WIN, LOSS, DRAW
reason	string	-	COMPLETED, FORFEIT, TIMEOUT
search	string	-	matchId or opponentId
from	ISO date	-	Filter by start date
to	ISO date	-	Filter by end date

Example Request

```
GET
/api/history/me?page=1&limit=10&result=WIN&sortBy=durationMs&order=desc
```

Success Response (200)


```
{
  "page": 1,
  "limit": 10,
  "total": 42,
  "totalPages": 5,
  "sortBy": "durationMs",
  "order": "desc",
  "data": [
    {
      "matchId": "match_98",
      "opponentId": "7",
      "result": "WIN",
      "yourSymbol": "X",
      "finalBoard": ["X", "O", "X", "", "O", "", "", "", "X"],
      "reason": "COMPLETED",
      "turnCount": 7,
      "durationMs": 54213,
      "endedAt": "2026-01-24T10:40:12Z"
    }
  ]
}
```


Error Responses

Status	Response
401	{ "error": "Unauthorized" }
500	{ "error": "Failed to fetch game history" }

2. Get Match By ID (Detailed View)

Endpoint

```
GET /:matchId
```

Returns full information about a specific match.

Example Request

```
GET /api/history/match_98
```

Success Response

```
{
  "matchId": "match_98",

  "players": {
    "you": "2",
    "opponent": "7"
  },

  "symbols": {
    "you": "X",
```

```
  "opponent": "O"
},

"firstTurn": "7",

"result": "WIN",
"winnerId": "2",

"reason": "COMPLETED",
"terminationBy": "BOARD_FULL",

"finalBoard": ["X", "O", "X", "", "O", "", "", "", "X"],

"moves": [
  {
    "playerId": "2",
    "position": 0,
    "symbol": "X",
    "at": "2026-01-24T10:38:01Z"
  },
  {
    "playerId": "7",
    "position": 1,
    "symbol": "O",
    "at": "2026-01-24T10:38:04Z"
  }
],
```

```
  "turnCount": 7,

  "startedAt": "2026-01-24T10:37:40Z",
  "endedAt": "2026-01-24T10:40:12Z",
  "durationMs": 54213
}
```

Error Responses

Status	Response
400	{ "error": "Match ID is required" }
401	{ "error": "Unauthorized" }
403	{ "error": "Access denied" }
404	{ "error": "Match not found" }
500	{ "error": "Failed to fetch match" }

History API Summary (for frontend)

Method	Endpoint	Purpose

GET	/api/history/me	List my matches
GET	/api/history/:matchId	Match details

Client Contract (Simple Rules)

- You can only see **your own matches**
- You cannot access other players’ matches
- Pagination is mandatory for large histories
- Filtering is optional but supported
- Sorting is restricted to safe fields only