Kontroller

Software Architecture Document

Version 4.0

Revision History

| **Date** | **Version** | **Description** | **Author** |
| --- | --- | --- | --- |
| 07/07/2025 | 1.0 | Write the first draft of the software architecture document | Nguyen Gia Nghi |
| 23/07/2025 | 2.0 | Modify some information presented | Nguyen Gia Nghi |
| 29/07/2025 | 3.0 | Revised All Software Architecture Document | Hoang Ngoc Tung |
| 29/07/2025 | 4.0 | Change some details in the Software Architecture Document | Duong Duc Thinh |
| 29/07/2025 | 5.0 | Modify deployment and implementation view to match the latest implementation | Nguyen Gia Nghi |

Table of Contents

[**1. Introduction 4**](#_heading=h.e3k61qssjxxh)

[**2. Architectural Goals and Constraints 4**](#_heading=h.15vymeyypt1j)

[Architectural Goals: 4](#_heading=h.vc5sx5xl952d)

[Technical Constraints: 4](#_heading=h.3nzwpo8s57hu)

[Non-Functional Requirements: 5](#_heading=h.u3idv3kyu4m2)

[**3. Use-Case Model 6**](#_heading=h.7q44vkrq2l4i)

[**4. Logical View 7**](#_heading=h.itcc9lkri4fo)

[4.1. Component: Frontend Application 9](#_heading=h.x9mmjuutaoiu)

[4.2. Component: Backend API server 11](#_heading=)

[**5. Deployment 14**](#_heading=h.8b0sn5wyh2ob)

[5.1. Development Environment 14](#_heading=h.umq9fis782vr)

[5.2. Production Environment 15](#_heading=h.rmm6s2cjhjmk)

[5.3. Database Deployment 15](#_heading=h.kvwb22sq4poz)

[5.4. App Components in Production 16](#_heading=h.kb2udve216ia)

[5.5. File Storage Options 16](#_heading=h.1sjg3et56l3j)

[5.6. External Services 16](#_heading=h.sxcym1ff3ki)

[5.7. Monitoring & Logging 17](#_heading=h.3wtyyak8c8wt)

[5.8. Deployment Pipeline 17](#_heading=h.9dr9dift9ogt)

[5.9. Security & Configuration 17](#_heading=h.1hlifnhx69np)

[**6. Implementation View 17**](#_heading=h.niyixbfq1zt)

Software Architecture Document

# Introduction

This Software Architecture Document provides a comprehensive architectural overview of Kontroller, a cross-platform web application that allows users to log, rate, and review video games while sharing their gaming experiences with friends and the gaming community. The platform serves as a centralized hub similar to Letterboxd or RateYourMusic, but specifically designed for gamers.

The document presents multiple architectural views to depict different aspects of the system, including logical, deployment, and implementation perspectives. It serves as a guide for developers, stakeholders, and future maintainers to understand the system's structure and design decisions*.*

**Key Features Covered:**

* Game logging and reviewing system
* Social networking and community features
* Real-time data integration with gaming platforms
* Responsive web interface for cross-platform compatibility
* User-generated content management

# Architectural Goals and Constraints

### **Architectural Goals:**

* Scalability: Support growing user base with efficient data handling and caching strategies
* Performance: Ensure fast response times for game searches, reviews, and social interactions
* Usability: Provide intuitive, responsive interface across desktop and mobile devices
* Maintainability: Implement clean, modular architecture for easy future enhancements
* Security: Protect user data with secure authentication and authorization mechanisms
* Integration: Seamless connection with external gaming APIs (IGDB, Steam, PSN)
* Real-time Features: Support live updates for social interactions and game activity feeds

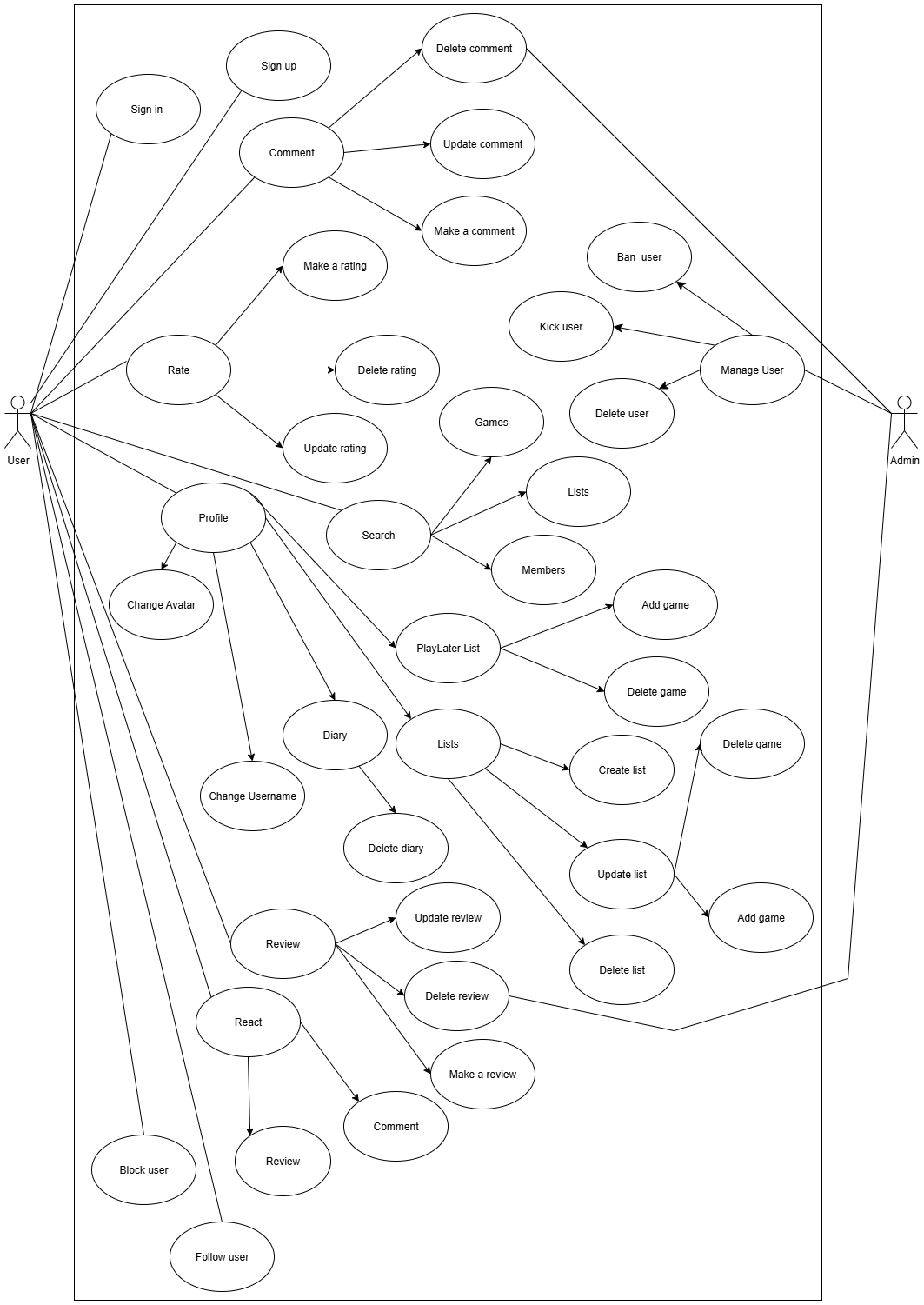
### **Technical Constraints:**

* Budget: Zero-budget deployment using free-tier platforms (Vercel, Firebase, Supabase)
* Timeline: 12-week development cycle with iterative releases
* Team Size: 5-member team with defined roles and responsibilities
* Technology Stack: Modern web technologies (React/Next.js, Node.js, PostgreSQL)
* Browser Support: Optimized for Chrome and modern browsers with mobile responsiveness
* Third-party Dependencies: Reliance on IGDB API for game data and metadata

### **Non-Functional Requirements:**

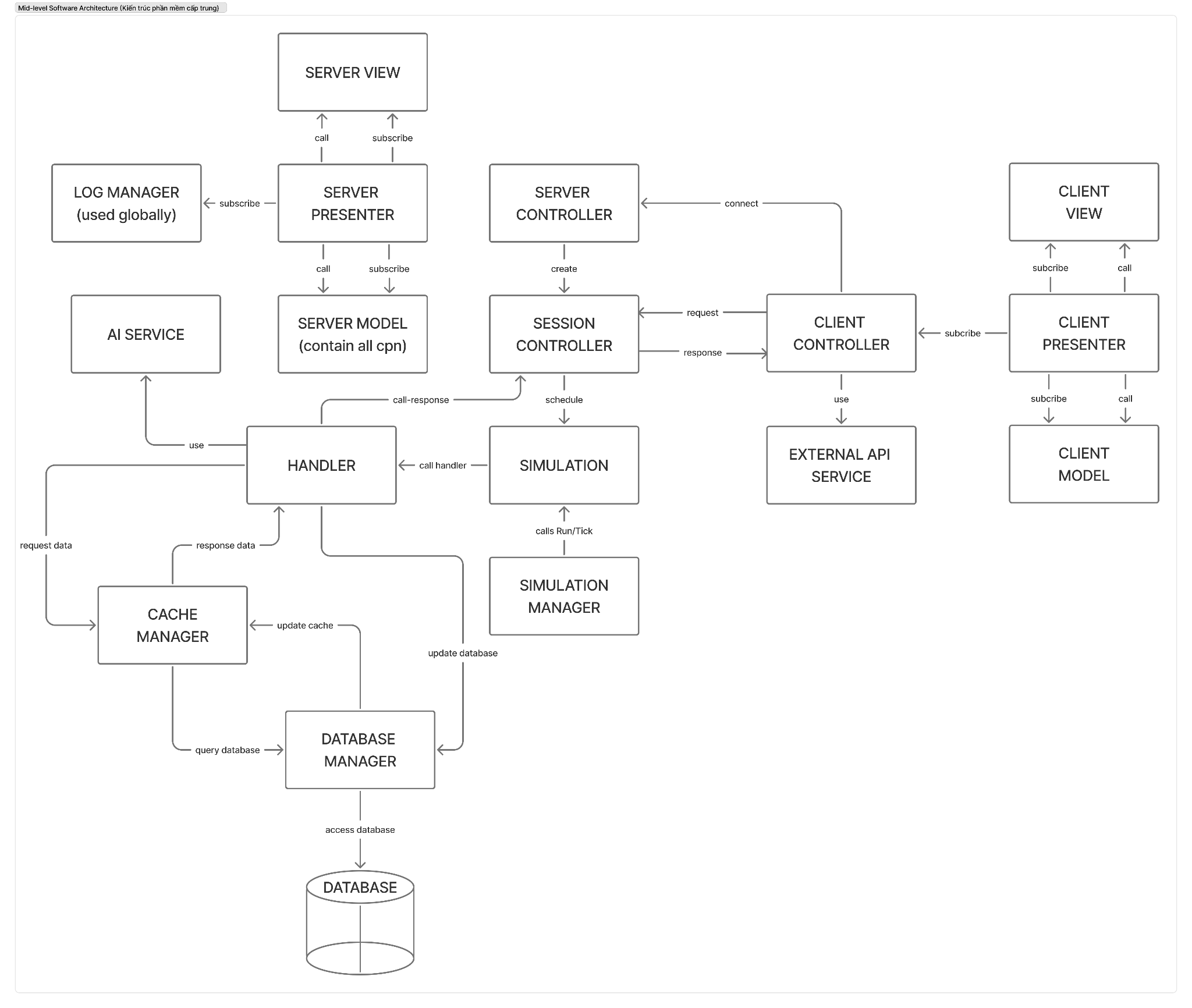
* Availability: 99% uptime during peak gaming hours
* Response Time: Page loads under 3 seconds, API responses under 500ms
* Concurrent Users: Support for 1000+ simultaneous users
* Data Storage: Efficient handling of user-generated content and game metadata
* Mobile Responsiveness: Fully functional on devices with screen sizes 320px and ab**ove**

# Use-Case Model

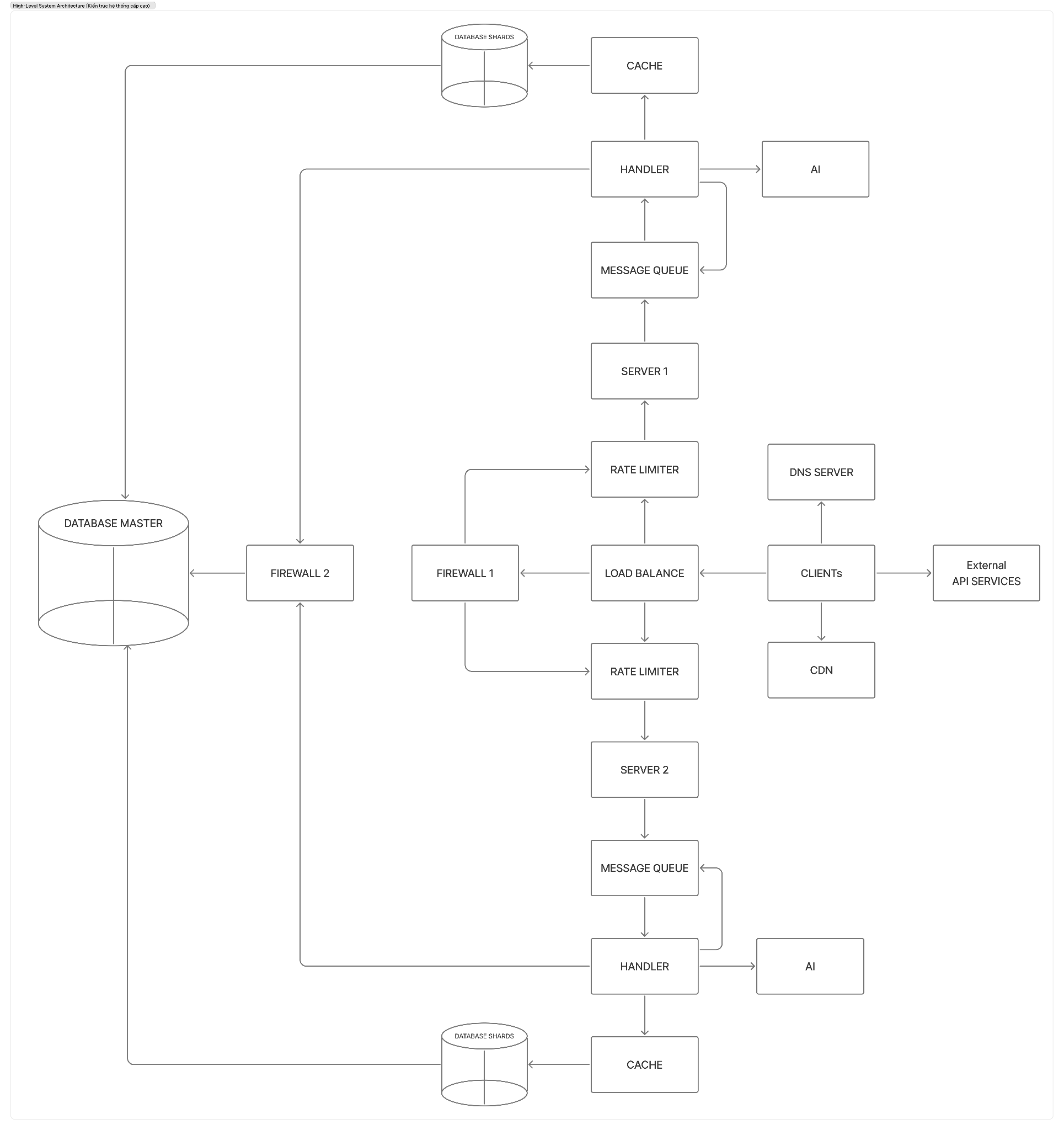


# Logical View

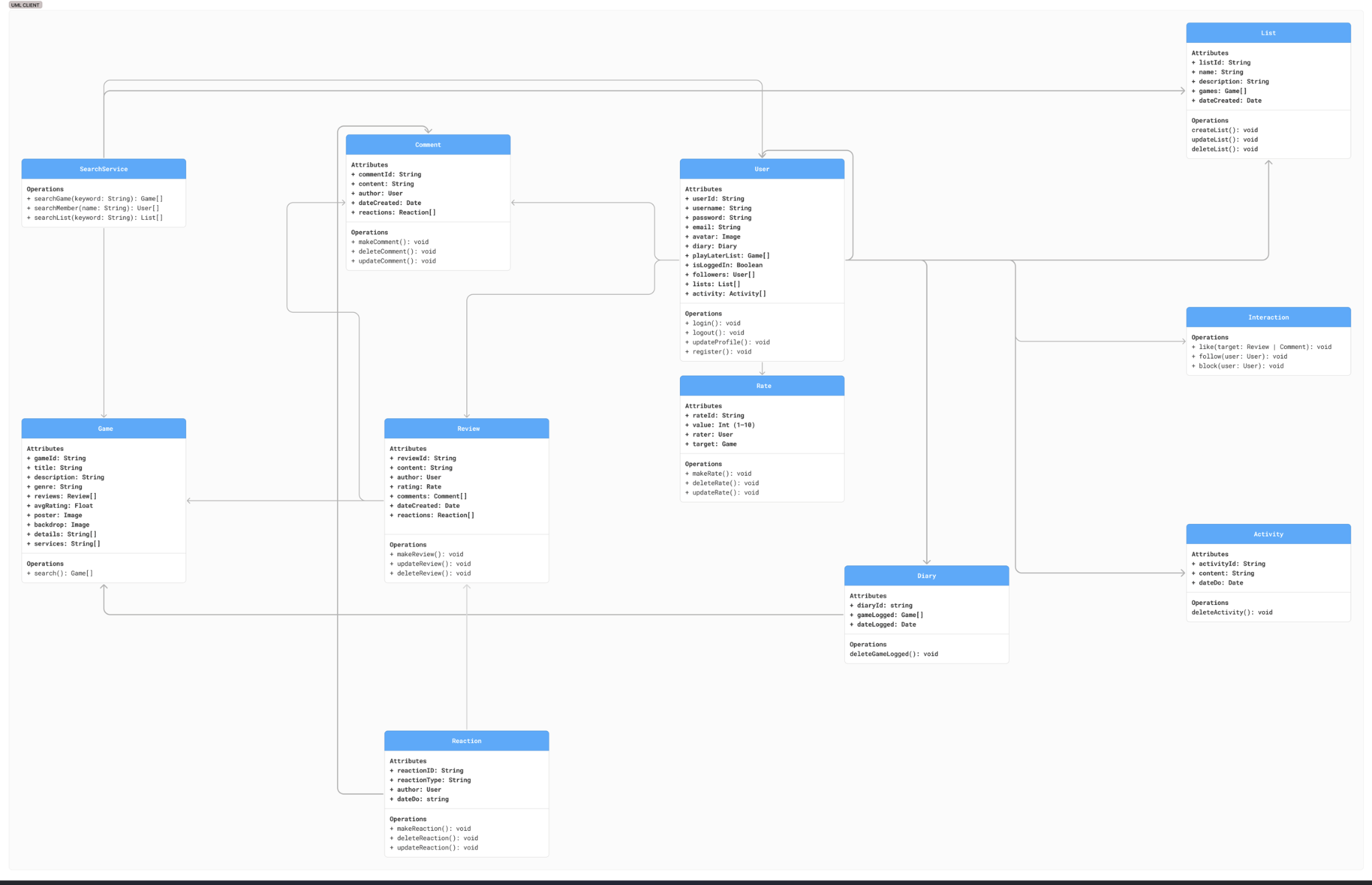
**Mid-level software architecture:**

****

**High-level system architecture:**

****

## Component: Frontend Application



**Responsibilities:**

* **User Interface (UI) Rendering:** The primary responsibility is to render the graphical user interface that users interact with. This includes displaying pages for games, user profiles, lists, reviews, and activity feeds.
* **State Management:** It manages the application's state on the client side. This includes keeping track of the current logged-in user, the data being displayed, and UI state like whether a form is open or a loading spinner is active.
* **User Input Handling:** It captures and processes all user inputs, such as text typed into search bars, clicks on buttons (like "Login", "Follow", "Post Review"), and form submissions.
* **Data Transformation:** It transforms data received from the back-end into a format suitable for display and translates user inputs into structured data to be sent to the back-end.

#### **Services Provided:** While a front-end component primarily *consumes* services, it provides the following essential functions:

* **To the User:** It provides the complete interactive experience, allowing users to browse, search, create content, and interact with other users on the platform.
* **To the Back-end:** It acts as the client that consumes the back-end's API. It translates user actions into structured API requests to fetch, create, update, or delete data on the server.

**Key classes:**

* User, Game, Review, Comment, Rate, Reaction, List, Diary, Activity: These are primarily data models. They define the structure of objects that hold data fetched from the back-end. For instance, when a user navigates to a game page, the front-end would make an API call like GET /api/games/{gameId}, and the JSON response would be parsed into a Game object, which would then be used to render the UI.
* SearchService: This is a dedicated service class. Its responsibility is to encapsulate the logic for making API calls related to searching. For example, the searchGame(keyword) method would construct and execute an HTTP request to the back-end's search endpoint (e.g., GET /api/search/games?q=keyword) and return the results as an array of Game objects.
* Interaction: This is another service class that centralizes the logic for social interactions. When a user clicks a "Follow" button, the UI would call the Interaction.follow(userToFollow) method. This method would then make the corresponding API call (e.g., POST /api/users/{userId}/follow) to the back-end to execute the action. This separation of concerns keeps the API logic out of the UI components.

**Communication:**

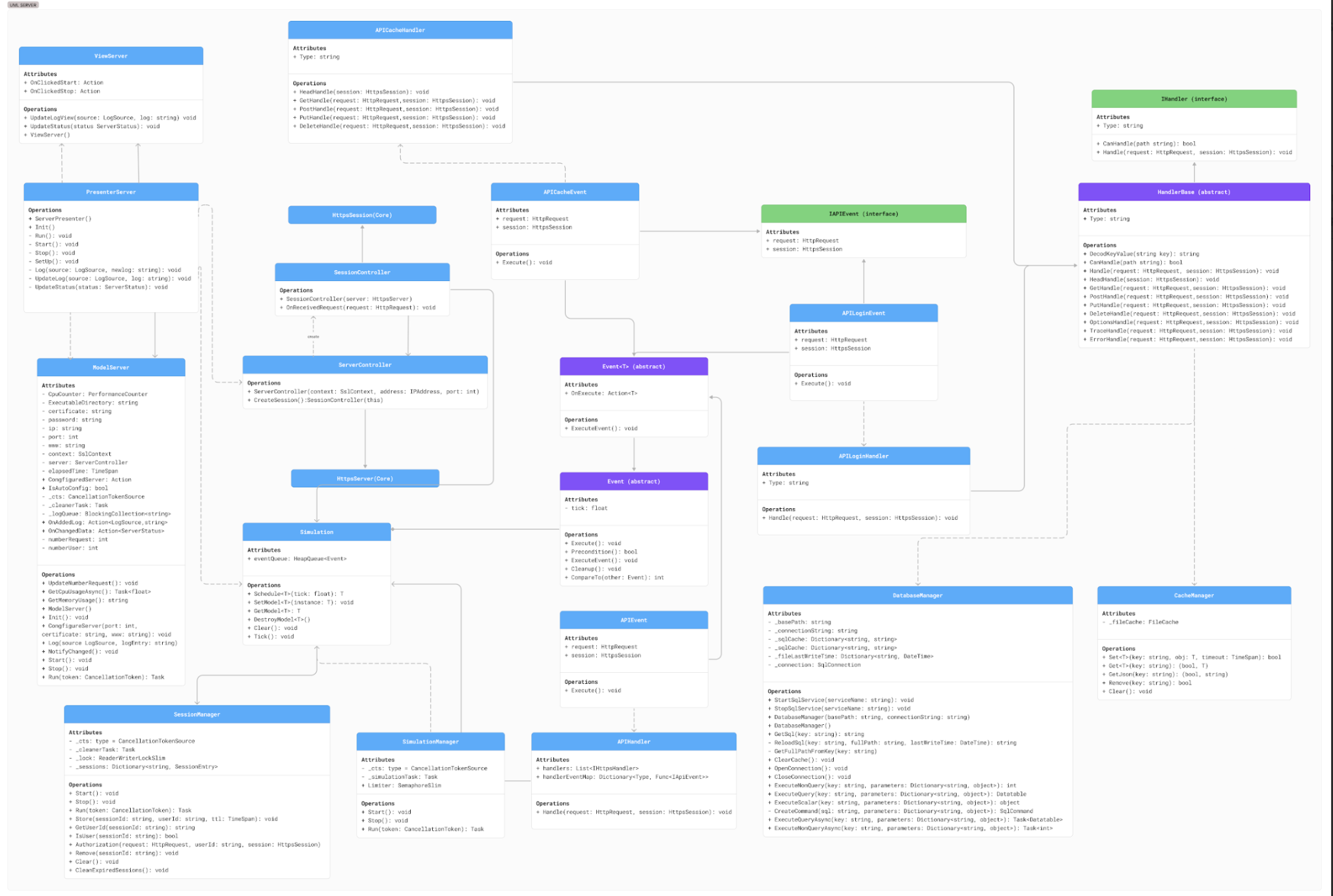
* Communication Method: Communication occurs through direct method calls and object references. For example, a UI component displaying a Game might call the game.search() method, or a User profile page would access the attributes of the User object (user.username, user.followers).

#### **External Relationships (With the Back-end)**

The front-end component communicates with a separate **Back-end Server** to persist data and retrieve information that is not stored locally. The operations listed in the classes (login, makeReview, searchGame, etc.) strongly imply the existence of a remote API.

* **Means of Communication:** **Internet** via the **HTTPS** protocol.
* **Architectural Pattern:** The communication likely follows a **RESTful API** or a **GraphQL** pattern. The front-end sends HTTP requests (e.g., GET, POST, PUT, DELETE) to specific endpoints exposed by the back-end server.  
  + **GET requests** would be used to fetch data (e.g., searchGame, loading a user's profile).
  + **POST requests** would be used to create new data (e.g., register, makeReview, makeComment).
  + **PUT or PATCH requests** would be used to update existing data (e.g., updateProfile, updateReview).
  + **DELETE requests** would be used to remove data (e.g., deleteReview, deleteActivity).

## Component: Backend API server



1. HTTP Communication Layer

This layer is the front door for all external client communication.

* Components: HttpsServer(Core), ServerController, SessionController, APIHandler, IHandler (interface), HandlerBase (abstract), APICacheHandler, APILoginHandler.
* Responsibilities:
  + HttpsServer(Core): Listens for incoming TCP connections on a specific port and performs the SSL/TLS handshake to establish a secure HTTPS session. It's the lowest-level network component.
  + ServerController: Acts as the primary factory for client sessions. When HttpsServer accepts a new connection, ServerController creates a dedicated SessionController to manage it.
  + SessionController: Manages the lifecycle of a single client's HTTPS session. Its primary role is to receive requests from the client and delegate them to the APIHandler for processing.
  + APIHandler: Functions as a central router or dispatcher. It maintains a collection of specialized handlers (those implementing IHandler). When a request is received, it determines which specific handler is responsible for the request's endpoint (e.g., /api/login/).
  + IHandler / HandlerBase / Concrete Handlers: These define the logic for specific API endpoints. HandlerBase provides common functionality for handling different HTTP methods (GET, POST, etc.). Concrete classes like APILoginHandler implement the logic for a specific feature, such as validating login credentials.
* Communication:
  + External: Communicates with external clients (like the Front-end component) over the Internet using the HTTPS protocol.
  + Internal: Communication occurs via direct method calls. ServerController creates SessionController instances. SessionController calls APIHandler.Handle(). APIHandler calls handler.Handle() on the appropriate concrete handler.

2. Application Core & Management Layer (MVP)

This layer manages the server application itself, rather than individual user requests. It's likely used for starting, stopping, and monitoring the server from a control panel or command line.

* Components: PresenterServer, ViewServer, ModelServer.
* Responsibilities:
  + ModelServer: The "Model". It holds the server's core configuration (ip, port, sslContext) and its runtime state (numberRequest, numberUser). It contains references to all major components (ServerController, SessionManager, etc.) and manages their initialization and shutdown.
  + ViewServer: The "View". It represents the server's state to an observer (e.g., a GUI or console window). It has methods like UpdateLogView and UpdateStatus to display real-time information.
  + PresenterServer: The "Presenter". It acts as the mediator between the ModelServer and ViewServer. It translates user actions from the view (e.g., a "Start Server" button click) into operations on the model (ModelServer.Start()) and updates the view with changes from the model.
* Communication: Internal communication via direct method calls and events/actions (e.g., OnClickedStart).

3. Business Logic & Event Processing Layer

This layer decouples request handling from the actual execution of business logic using an event-driven pattern.

* Components: Simulation, SimulationManager, Event (abstract), APIEvent, APILoginEvent.
* Responsibilities:
  + Event / APIEvent: These are objects that represent a unit of work to be done, such as logging in a user or fetching data. They encapsulate all the necessary data for the task (e.g., HttpRequest). This design promotes asynchronous processing.
  + Simulation: The core of the event processor. It maintains a priority queue of Event objects (eventQueue). Its Tick() method likely dequeues and executes the next event. This allows for scheduled, ordered, or delayed execution of tasks.
  + SimulationManager: Manages the lifecycle of the Simulation event loop. It starts, stops, and runs the loop, possibly throttling it with a SemaphoreSlim to control resource usage.
* Communication: Handlers in the HTTP layer (e.g., APILoginHandler) create Event objects and schedule them with the Simulation using Simulation.Schedule(). The SimulationManager then drives the execution of these events.

4. Data Persistence & Caching Layer

This layer abstracts all data storage details from the rest of the application.

* Components: DatabaseManager, CacheManager.
* Responsibilities:
  + DatabaseManager: Manages all communication with a SQL database. It handles connection management, loading SQL queries from files, and provides a set of methods (ExecuteQuery, ExecuteNonQueryAsync, etc.) to perform database operations securely, preventing direct SQL injection from business logic.
  + CacheManager: Provides a generic caching service to reduce database load and improve response times. It offers standard Set, Get, and Remove operations and may persist its cache to a file (\_fileCache).
* Communication: Internal communication via method calls from event handlers or other business logic components. The DatabaseManager communicates with the database server over a LAN connection using a database-specific protocol (e.g., TDS for SQL Server).

5. Session Management Layer

This component is dedicated to handling user sessions.

* Component: SessionManager.
* Responsibilities:
  + Manages the lifecycle of user sessions after a successful login.
  + Stores session information, mapping a session ID to a user ID.
  + Provides an Authorization method to verify that an incoming request belongs to a valid, authenticated session.
  + Includes a background task (\_cleanerTask) to periodically remove expired sessions, preventing memory leaks.
* Communication: The SessionManager is called by business logic components (likely event handlers like APILoginEvent) to create sessions and by handlers for protected endpoints to authorize requests.

# Deployment

## Development Environment

**Frontend**

* Environment: Local development server  
  Tech stack:
  + Static HTML pages styled with **Tailwind CSS**
  + Modular JavaScript (api.js, event.js, handle.js, ui.js)
* Sample Pages: activity.html, auth.html, diary.html, friends.html, game-detail.html, etc.

**Backend**

* Framework: **.NET Core (C#)**
* Running locally on localhost
* Local connection to development database
* Debugging enabled with verbose logging

**Database**

* Engine: **SQL Server** for development
* Lightweight configuration for local testing

## Production Environment

#### **Frontend Deployment**

* **Platform Options**:
  + GitHub Pages
  + Netlify (Free Tier)
* **Assets**:
  + Static HTML/CSS/JS
  + Tailwind CSS compiled and JS modules bundled via build tools
* **Other Features**:
  + Built-in CDN via hosting provider
  + HTTPS via GitHub/Netlify
  + Custom domain support (~$10/year)

#### **Backend Deployment**

**Option 1: Railway or Render (Free Tiers)**

* Deploy the .NET backend with Git integration
* Automatic HTTPS & CI/CD pipeline

**Option 2: Oracle Cloud Free Tier**

* Free forever: 2 ARM-based virtual machines
* 1GB RAM per instance — enough for small traffic
* Host backend and optionally the database

## Database Deployment

**Option 1: Azure SQL Free Tier**

* Managed DB with backups
* 32MB free — suitable for small test deployments

**Option 2: SQL Server Express on VPS**

* Install and run SQL Server on a hosted VM
* Use SSMS for backups and admin

**Option 3: SQLite (Production Lite)**

* Embedded file-based DB, no server required
* Works well for lightweight workloads with limited concurrency

## App Components in Production

* **Controllers**: SessionController.cs, ServerController.cs: Handle routing and session flow
* **Core Modules**: Fuzzy logic, simulation engine, and time tracking
* **Data Layer**: UserDatabase.cs: ORM & DB interaction
* **Event System**: Includes APICacheEvent, APIEvent, APILoginEvent, etc.
* **Handlers**: API processors like APIHandler, APIUserHandler, etc.
* **Managers**: Modules for caching, logging, sessions, and DB management

## File Storage Options

**Option 1: GitHub Repository (Raw URL serving)**

* Use for static images
* Free, but limited by GitHub's size/policy constraints

**Option 2: Google Drive API**

* 15GB free storage
* Requires Drive API integration for uploads/serving

**Option 3: Cloudinary (Free Tier)**

* 25GB/month with CDN and optimization
* Good for user-uploaded avatars/screenshots

## External Services

* **IGDB API**:
  + Game metadata provider
  + Uses secured environment variables
  + Built-in caching and throttling
* **File Storage**:
  + Azure Blob or AWS S3 integration
  + Supports CDN delivery
* **Email**:
  + Providers: SendGrid, Mailgun, or AWS SES
  + Used for password resets and system notifications
  + Includes template support and tracking

## Monitoring & Logging

* Built-in LogManager.cs handles application logging
* Performance metrics and system health monitoring
* Basic alerting for server errors and downtime

## Deployment Pipeline

| **Stage** | **Description** |
| --- | --- |
| Version Control | Code maintained in Git (frontend + backend) |
| Build | Tailwind CSS compiled, JavaScript bundled; .NET backend packaged |
| Testing | Optional unit and integration tests pre-deployment |
| Staging | Pre-release verification |
| Production | Deployment with rollback support |

## Security & Configuration

1. Environment-specific .env configurations
2. Secure handling of API keys and DB connections
3. HTTPS/TLS enforced across all endpoints
4. Authentication & session management via token
5. Rate limiting and brute-force/DDoS mitigation

# Implementation View

Controller/

├── ServerController.cs

└── SessionController.cs

Core/

├── Fuzzy.cs

├── HeapQueue.cs

├── Simulation.cs

├── Simulation.Event.cs

├── Simulation.InstanceRegister.cs

└── Time.cs

Event/

├── APICacheEvent.cs

├── APIEvent.cs

├── APILoginEvent.cs

└── APIUserEvent.cs

Handler/

├── APICacheHandler.cs

├── APIHandler.cs

├── APILoginHandler.cs

└── APIUserHandler.cs

Interface/

├── IAPIEvent.cs

├── IDatabase.cs

└── IHandler.cs

Manager/

├── CacheManager.cs

├── DatabaseManager.cs

├── LogManager.cs

├── SessionManager.cs

└── SimulationManager.cs

Model/

└── ModelServer.cs

NetCoreServer/

├── HttpsServer.cs

├── HttpsSession.cs

Presenter/

└── ServerPresenter.cs

View/

├── ViewServer.cs