<The Next Stage Ltd.>

<Kontroller>
Software Architecture Document

Version <1.0>

Revision History

Date	Version	Description	Author
<07/07/2025>	<1.0>	Write the first draft of the software architecture document	Nguyễn Gia Nghi

Table of Contents

Software Architecture Document

1. Introduction

The Software Architecture Document (SAD) for Kontroller outlines the overall structure and organization of the software system. This document defines the architectural approach, major components, interactions between them, and the design rationale behind the choices. It is intended to support the design, development, and future maintenance of the Kontroller platform.

1.1 Purpose

The purpose of this document is to provide a comprehensive and structured description of the architectural design of the Kontroller system. It ensures that all readers have a shared understanding of the system's structure and its high-level design principles.

1.2 Scope

Kontroller is a social web-based platform where users can log, review, and rate video games, create lists, interact with others, and manage their game-playing history. This architecture document covers the frontend, backend, database, and API layers of the MVP (Minimum Viable Product) version of the system.

1.3 Definitions, Acronyms, and Abbreviations

Term	Definition
MVP	Minimum Viable Product
API	Application Programming Interface
UI	User Interface
UX	User Experience
DB	Database
FE	Frontend
BE	Backend
JWT	JSON Web Token
IGDB	Internet Game Database (external API)

1.4 References

- Vision Document
- Trello scheduling
- Figma charts and graphs
- Use-case documents and models
- UI design

1.5 Overview

This document contains:

- Architectural goals and constraints
- The use-case model and logical architecture
- A component-based breakdown of the system and the relationships between components
- Descriptions of key interactions and flows within the system

2. Architectural Goals and Constraints

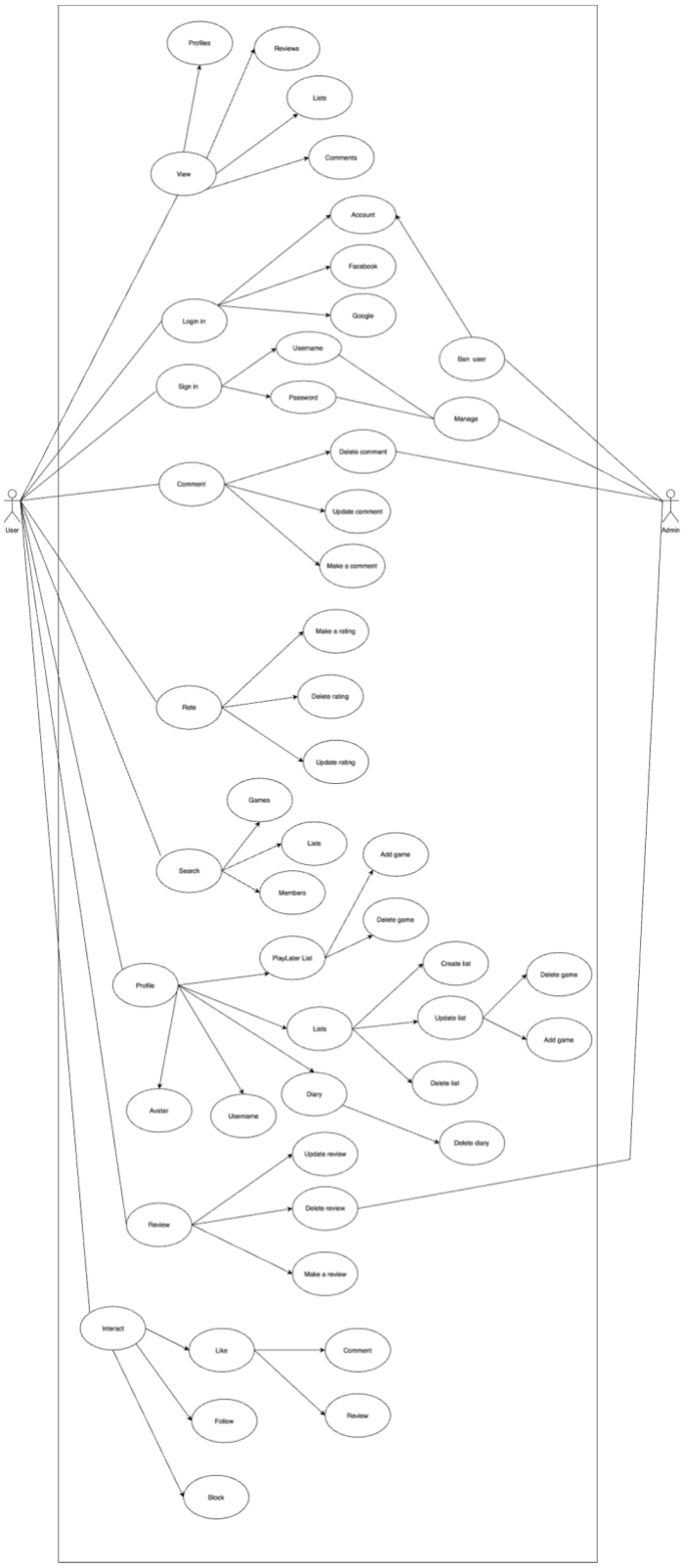
2.1 Goals:

- The application must be a responsive web-based system (works on both desktop and mobile).
- **Security:** Must support secure authentication via username/password.
- **Performance:** Pages must load within 2 seconds on average, with minimal delays on interactions.
- **Portability:** Should be deployable on most modern web servers.
- **Reusability:** Code must be modular and follow best practices to enable reuse of UI components and backend modules.
- **API-first design:** Frontend and backend communicate via RESTful APIs.
- All passwords must be securely hashed and stored using best practices

2.2 Constraints:

- Development stack constraints:
    - Frontend: ReactJS
    - Backend: C#
    - Database: MySQL
- Use of external APIs (IGDB) is required for fetching verified game data.
- CORS must be enabled and configured correctly to allow secure communication between FE and BE.

3. Use-Case Model



4. Logical View

The architecture of the Kontroller app is divided into three primary layers: Frontend (Client), Backend (Server), and Database (Data Storage). These components communicate through well-defined APIs over HTTPS, using RESTful conventions.

The diagrams below illustrate the structure of each layer in terms of classes, responsibilities, and interaction.

4.1 Component: Client (Frontend)

The Client side is responsible for rendering the UI and handling user interactions. It is developed using React.js and communicates with the backend via HTTP(S).

Main responsibilities:

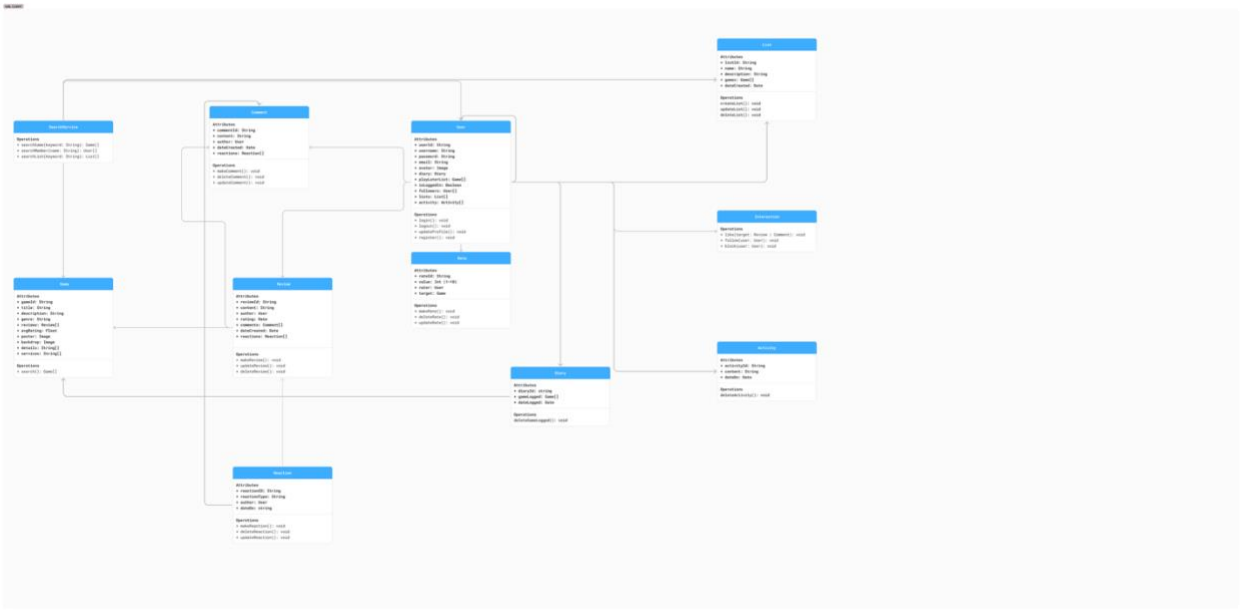
- Render user interfaces for login, profile, reviews, search, and list features.
- Interact with backend through API calls for authentication, data retrieval, and updates.
- Manage state such as currently logged-in user, cached data, and user actions like likes/follows.

Key Classes/Entities:

- **User**: Represents a user object with attributes like username, password, email, avatar, diary, etc. It also includes methods like login(), logout(), updateProfile().
- **Game**: Used to display and handle game data such as title, poster, genre, and platforms.
- **Review**: A component that handles viewing, creating, updating, and deleting reviews.
- **Comment**: Handles comment interactions on reviews, with operations like makeComment(), deleteComment(), and updateComment().
- **Rate**: Supports game rating from users (0–10).
- **Interaction**: Includes social interactions such as like(), follow(), block() applied to users, reviews, or comments.
- **List**: Enables users to create custom game lists with genres and names.
- **Diary**: Records games logged by the user with date.
- **Activity**: Represents logs of user activities such as writing reviews, logging games.
- **SearchService**: Allow users to search for games, lists, members.
- **Reactions:** Stores reactions of a comment or review

Communication: HTTPS-based API calls to backend routes (e.g. /api/user/login, /api/review/add, etc.)

UML client diagram:



4.2 Component: Server (Backend)

The Server handles business logic, API request processing, data validation, and interfacing with the database. It is built using C#.

Main responsibilities:

- Receive, parse, and respond to client requests securely and efficiently.
- Interact with the database to perform CRUD operations.
- Handle session management, authentication, error logging, and API routing.

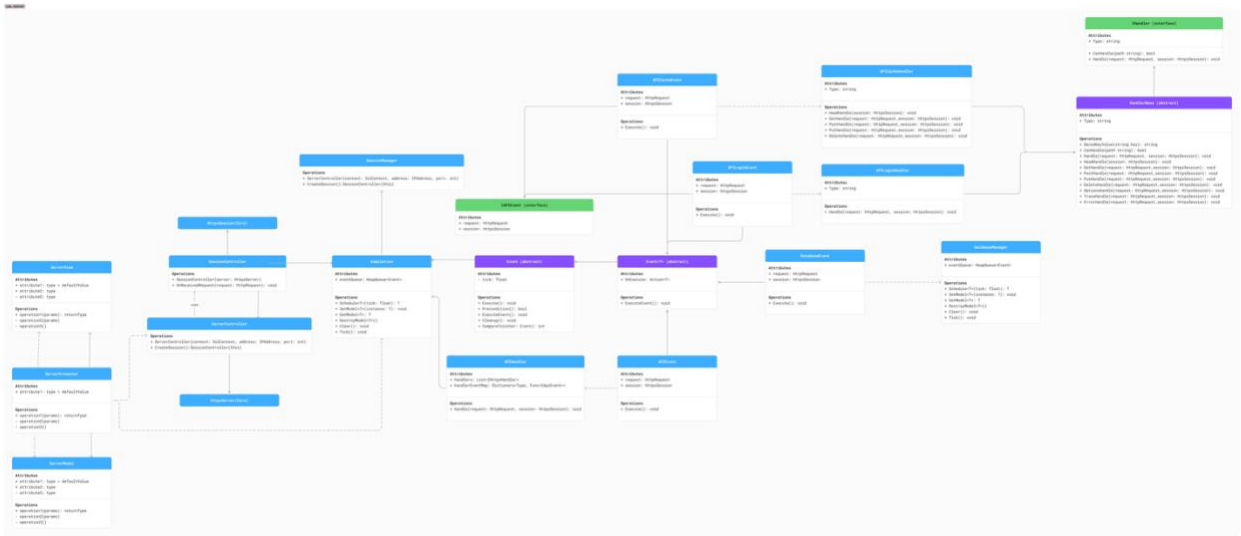
Key Classes/Modules:

- **HttpSession**: Core module maintaining session context for users.
- **SessionController**: Handles session validation and lifecycle.
- **ServerController**: The main entry point that manages socket sessions and handles requests from the HTTP server.

- **DatabaseManager**: Manages database transactions such as inserting, updating, or retrieving data.
- **APIEvent** (and its subclasses): Represent API requests (e.g. login, data fetch).
    - **Examples:** APILoginEvent, APIDataEvent, APICacheEvent, etc.
- **HandlerBase** and **Handler**: Abstract and concrete classes used to process API topics and assign appropriate logic.
- **Simulation**: Handles simulation of events and integrates server-client-side interaction.

Communication: Primarily over sockets and HTTP sessions, responding to frontend through JSON-based API responses.

UML server diagram:



4.3 Component: Database (ER Diagram)

The Database stores all persistent data for the system and is structured based on the ER diagram provided. It is built using MySQL.

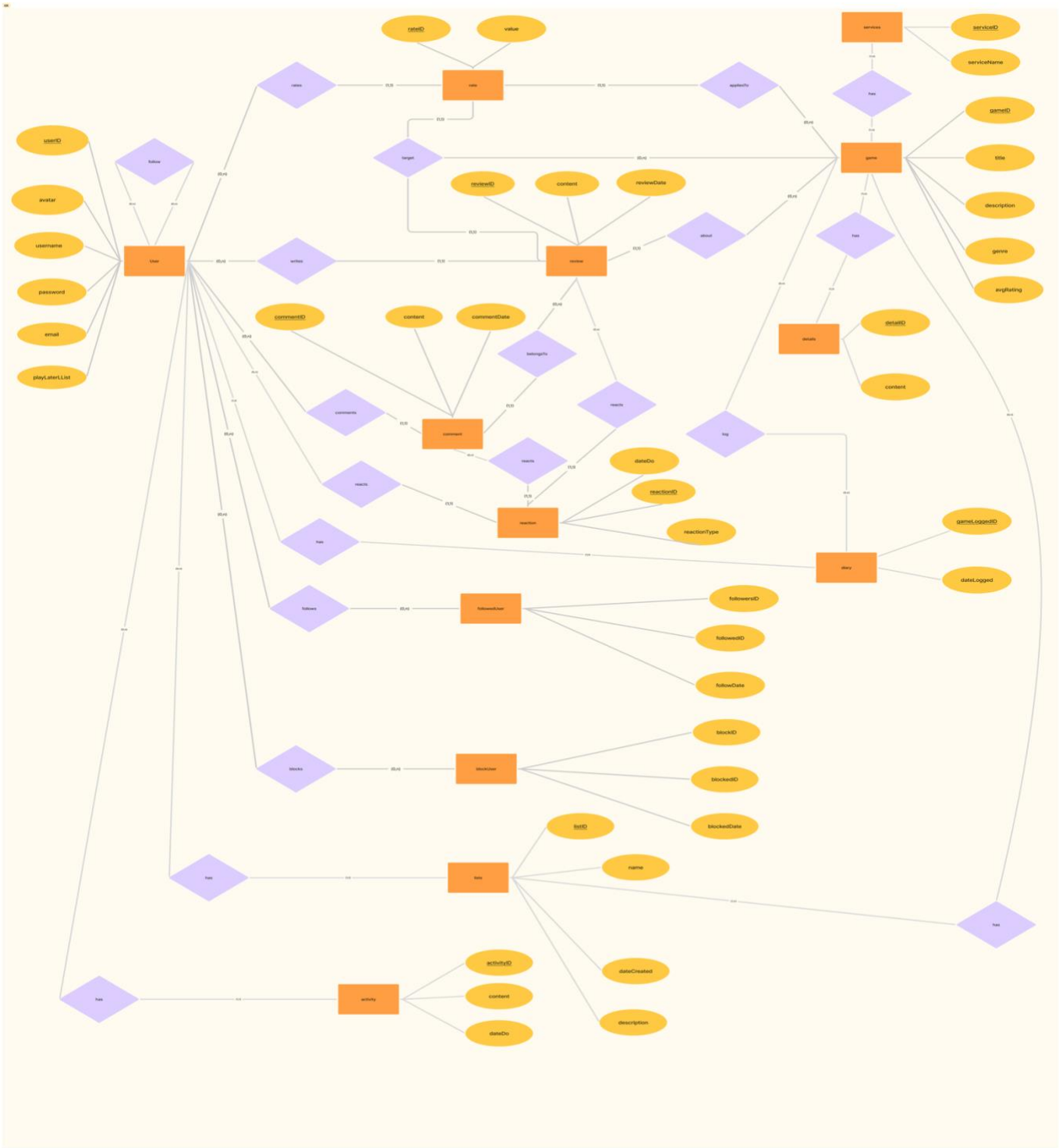
Main Entities:

- **User**: Stores user credentials, avatar, diary, and relationships (follows, blocks).
- **Game**: Stores game metadata such as title, genres, average rating, and descriptions.
- **Review**: Links to users, with review content and comments.
- **Comment**: Contains user-generated comments on reviews.

- **Rate**: Stores numeric user ratings on games.
- **Lists**: User-created lists, linked to games by genre and description.
- **Diary**: Logs game activities with timestamps.
- **Interaction**: Tracks social features, includes:
    - Followed users
    - Blocked users
- **Activity**: Records any action taken by a user (logs, reviews, etc.)
- **Services:** Stores the names of services making games, linked to games.
- **Details:** Stores details such as countries, studios and languages, linked to games.
- **Reactions:** Stores reactions of a comment or review

Communication: The server communicates with the database using the DatabaseManager module. CRUD operations are handled securely and efficiently with respect to user input.

ER model:


rateID
value
rate
rates
applies To
services
serviceID
serviceName
has
gameID
title
description
genre
avgRating
game
userID
follow
avatar
username
password
email
playLaterList
User
target
reviewID
content
reviewDate
review
writes
about
has
commentID
content
commentDate
belongsTo
details
detailID
content
comments
comment
reacts
log
reacts
dateDo
reactionID
reactionType
reaction
has
gameLoggedID
dateLogged
diary
follows
followsUser
followersID
followedID
followDate
blocks
blockUser
blockID
blockedID
blockedDate
has
listID
name
lists
has
has
activityID
content
dateDo
activity
dateCreated
description

4.4 Component: Mid-level software architecture

Responsibilities:
The **Cache Manager** is responsible for minimizing database access frequency and improving performance by caching frequently requested data. It interacts with the **Handler** to provide fast responses and updates its cache when new or modified data is received from the **Database Manager**.

Key Functions:

- **Request Handling**: When data is requested, the Cache Manager first checks if the data is already cached. If so, it returns the cached data.
- **Query Delegation**: If data is not cached, it queries the **Database Manager**, receives the response, and stores it in the cache for future requests.
- **Cache Invalidation**: When new data is inserted or existing data is updated, the **Handler** instructs the Cache Manager to refresh or invalidate the cache.
- **Performance Optimization**: Reduces latency for commonly accessed endpoints (e.g., game details, user profiles, recent activity) by avoiding repeated DB hits.

Inter-component Communication:

- Communicates with:
  - **Handler**: to receive queries and serve cached responses.
  - **Database Manager**: to retrieve original data for uncached requests and to receive notifications of changes.
- Direction:
  - From Handler → Cache Manager: **request data**
  - From Cache Manager → Database Manager: **query database**
  - From Cache Manager → Handler: **response data**
  - From Handler → Cache Manager: **update cache**

Protocols Used:

- Internal method calls within the backend application (e.g., service-to-service communication).
- Memory-based caching (e.g., in-memory JavaScript object, Redis, etc.).

Benefits:

- Improves scalability and responsiveness of the system.
- Reduces load on the database layer.
- Enables smoother simulation execution by ensuring timely access to data for real-time updates.

Mid-level software architecture: