**High level Architecture**

**<Project Code>:<Project Name>**

| **Student ID** | **Name** |
| --- | --- |
| **24100175** | **Abdul Muiz** |
| **24100127** | **abdur Rafae Haroon** |
| **24100277** | **Bisma Nawaz** |
| **24100282** | **Waleed Nadeem** |
| **24100308** | **Hafsa Ahmed** |

**Table of Contents**

[1.](#_gjdgxs) Introduction 3

[2.](#_30j0zll) System Architecture 4

[2.1 Architecture Diagram 4](#_1fob9te)

[2.2 Architecture Description 5](#_3znysh7)

[2.3 Justification of the Architecture 6](#_2et92p0)

[3.](#_tyjcwt) Risk Management 7

[3.1 Potential Risks and Mitigation Strategies 7](#_3dy6vkm)

[4.](#_1t3h5sf) Tools and Technologies 8

[5.](#_4d34og8) Hardware Requirements 9

[6.](#_2s8eyo1) Who Did What? 10

[7.](#_17dp8vu) Review checklist 10

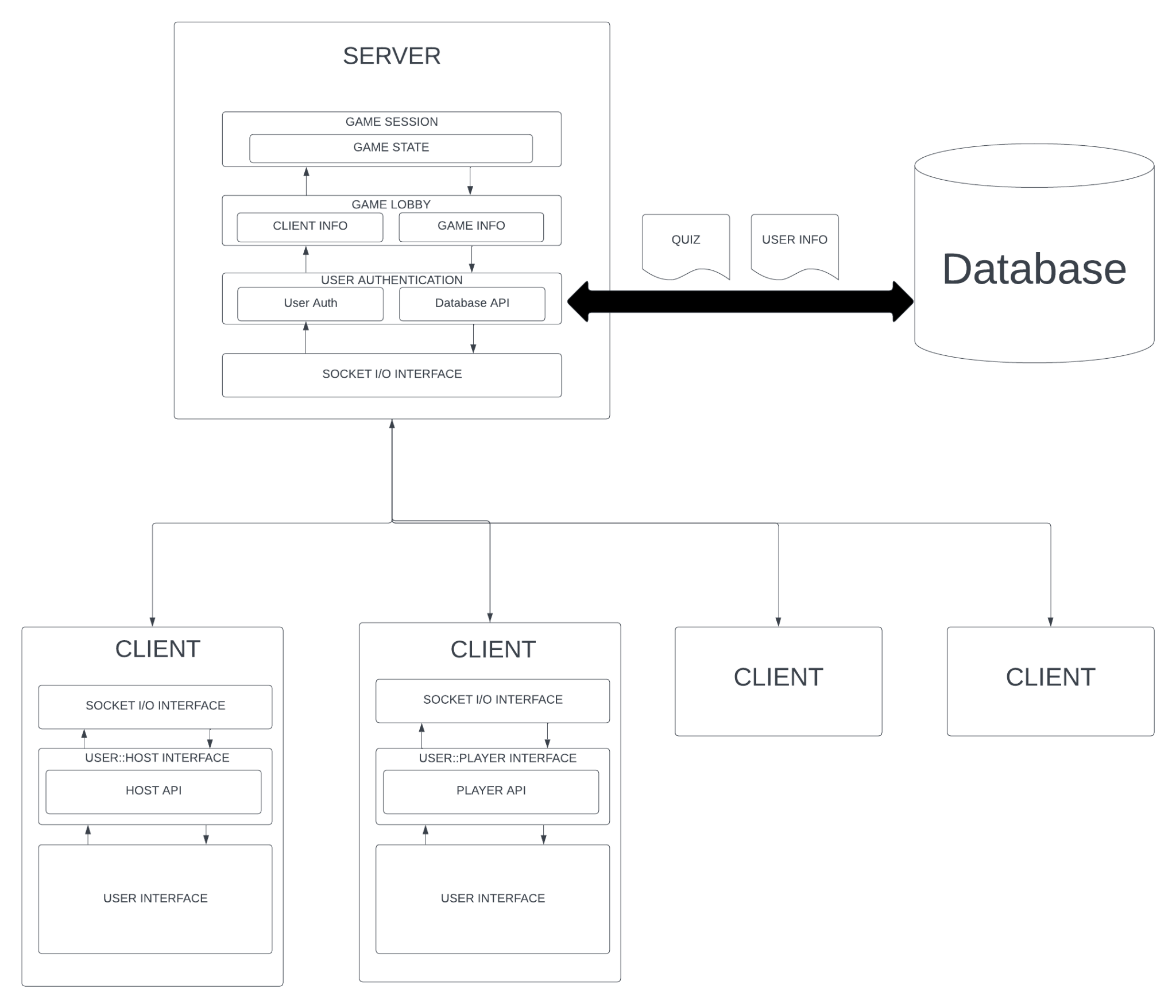
# Introduction

The proposed project is a user-hosted trivia web application. This application aims to create an interactive quiz-based learning experience, taking its inspiration from the popular game-based learning platform, ‘Kahoot!’. Designed for students, teachers, trainers or simply anyone who wants to challenge their friends to a quiz, the extent of the web application’s features allow anyone to quickly create and host a quiz-based session and let others join in and participate. Thus, within a session, there will exist a host user who creates the trivia while the participants joining voluntarily act as players. The players will use the quiz ID given by the host to join a particular session.

# System Architecture

## Architecture Diagram

<Draw a diagram of the system architecture. Remember that your system’s architecture may be defined using multiple architectural patterns.>



## Architecture Description

**Client**

The client is the user-facing application or software that interacts with the server to access services and data. It communicates with the server through various interfaces, such as Socket I/O Interface and User::Host Interface.

**Server**

The server is a centralized computer or system that responds to client requests, processes data, and manages the database. It interfaces with clients through various APIs and interfaces like Host API and User::Player Interface.

**Database**

The database stores and manages data, including user information, game state, and other relevant information. It is accessed and manipulated by the server through the Database API.

**Socket I/O Interface**

This interface enables real-time communication between the client and server by facilitating data transfer over network sockets, allowing for messages and updates to be sent and received.

**User::Host Interface**

This interface connects user clients with the server, handling authentication, game lobby management, and initial user interactions.

1. **Host API**

The Host API is used by the server to manage game lobbies, create and manage game sessions, and handle various administrative tasks related to hosting games.

**User::Player Interface**

This interface bridges the gap between individual user clients and the player functionality within the game, allowing players to join games, interact with game sessions, and manage their in-game profiles.

1. **Player API**

The Player API provides the server with the means to interact with individual players, update their game progress, and facilitate player-specific actions within the game.

**User Interface**

This is the graphical or interactive part of the client that users interact with, providing a user-friendly way to access and navigate the game's features.

**User Authentication**

This process verifies the identity of users connecting to the server, ensuring that only authorized users can access the game and its services.

1. **User Auth**

An abbreviation for "User Authentication," it may refer to the authentication mechanism used to verify the identity of users during login.

1. **Database API**

This API allows the server to interact with the database, enabling data retrieval, storage, and management.

**Game Lobby**

A virtual space where users can gather, create, and join game sessions, enabling interaction and game setup before entering actual gameplay.

1. **Client Info**

Information about the client, including user preferences, settings, and device specifications.

1. **Game Info**

Data about available games, their descriptions, and the characteristics of individual game sessions.

**Game Session**

A specific instance of a game being played, including the players involved, the game rules, and other session-specific data.

1. **Game State**

The current status and progress of a game session, storing information such as player scores, game board configurations, and other relevant in-game data.

## Justification of the Architecture

**Pros of the Architecture:**

1. **Scalability:** The architecture is designed to support multiple clients, making it suitable for a trivia game with potentially many simultaneous players. It can scale horizontally by adding more server instances to handle increased user loads.

2. **Modularity:** The system is divided into various subsystems (Client, Server, Database) with clear interfaces and APIs. This modularity simplifies maintenance and allows for easier upgrades and enhancements to individual components without affecting the entire system.

3. **Real-time Interaction:** The Socket I/O Interface enables real-time communication between clients and the server, which is crucial for a trivia game where players need instant feedback and updates.

4. **Security:** The User Authentication component ensures secure access to the system, protecting user data and maintaining the integrity of the game. This is essential for maintaining user trust and complying with security requirements.

5. **Flexibility:** The architecture allows for different roles within the game (Host and Player), and the User::Host and User::Player Interfaces facilitate these distinctions. This flexibility caters to the diverse needs of users.

6. **High Availability:** By centralizing game state and user information in the Database, the architecture can ensure high availability and data consistency. This is critical for a system where users may expect continuous access.

7. **Ease of Management:** The use of APIs (Host API, Player API) to manage game lobbies, game sessions, and player interactions simplifies administrative tasks, making it easier for game administrators to oversee the system.

8. **User-Friendly Interfaces:** The User Interface component provides a user-friendly way for clients to access game features, enhancing the overall user experience.

**Cons of the Architecture:**

1. **Complexity:** The architecture is relatively complex due to its various components and interactions. This complexity can make development and debugging more challenging.

2. **Latency:** While the Socket I/O Interface enables real-time communication, it might introduce some latency, especially when handling a large number of simultaneous connections.

**Justification for the Architecture:**

This architecture is appropriate for the trivia web application for several reasons:

1. **Scalability:** The ability to scale horizontally supports the potential growth of the user base. This is vital for a trivia game, where user numbers can vary significantly over time.

2. **Real-time Interaction:** Real-time communication is a must for interactive games. The Socket I/O Interface ensures that game sessions are engaging and responsive.

3. **Security:** The architecture places a strong emphasis on user authentication and data security, which is essential for protecting user privacy and maintaining trust.

4. **Modularity:** Modularity facilitates development and maintenance, making it easier to add new features, fix issues, and adapt to changing requirements.

5. **User-Friendly Interfaces:** The User and Player interfaces ensure that users can interact with the system with ease, improving the overall user experience.

6. **High Availability:** Centralizing data in the database enhances availability and ensures that players can access their game history and profiles reliably.

7. **Flexibility:** The ability to cater to different user roles, such as Host and Player, accommodates various user needs, making the system more inclusive.

This architecture also helps in the implementation of non-functional requirements:

1. **Performance:** The architecture's modularity and scalability support performance improvements by allowing the addition of more server instances and optimizing specific components.

2. **Reliability:** The centralized database, clear interfaces, and real-time communication help ensure system reliability, critical for uninterrupted gameplay.

3. **Security:** The user authentication and database security mechanisms ensure the safety of user data and meet security requirements.

4. **User Experience:** The user-friendly interfaces and real-time interaction enhance the overall user experience, contributing to user satisfaction.

In summary, the proposed architecture is well-suited for a trivia web application, considering its ability to handle real-time interactions, scalability, security, and user-friendliness while helping fulfill non-functional requirements like performance and reliability.

# Risk Management

## Potential Risks and Mitigation Strategies

<List down top 10 potential risks and their mitigation strategies>

| **Sr.** | **Risk Description** | **Mitigation Strategy** |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

# Tools and Technologies

**Development tools:**

* Front-end framework:
  + Svelte (latest version: 4.2.1): A lightweight and efficient JS framework for building user interfaces.
* Backend framework:
  + Node.js (version 16+): A javascript run-time for server-side development.
  + Express.js (latest version): A Node.js library for building web applications, simplifying routing and server-side development.
* Database:
  + Supabase (latest version): An open-source alternative to Firebase that offers real-time database and authentication services.
* Real-time communication:
  + Socket.io (latest: 4.7.2): A JS library for real-time bidirectional event-based communication between the client and the server.
* Version control:
  + Git (latest version): A distributed version control system for tracking changes in the project.
* Code editor:
  + VS Code (latest version): A popular source code editor with support for various programming languages and frameworks.
* Package manager:
  + npm (Node package manager, combined with Node.js): Used for managing and installing project dependencies.
* Project management:
  + Jira: A management tool to facilitate agile development, thereby allowing efficient tracking of tasks and project progress.

**Deployment tools:**

* Deployment platform:
  + Heroku: A platform-as-a-service (PaaS) to streamline and simplify the process of deploying and managing web applications.
* Scalability:
  + Amazon EC2 auto-scaling: For automatically adjusting the number of EC2 instances in response to changing traffic loads. It will be a great choice for scaling our web application based on the network traffic.

**Database tools:**

* Supabase (latest version): Will be used as a primary database for storage and retrieval.

# Hardware Requirements

## Development Machine Requirements:

* It should atleast have a multi-core processor.
* It should atleast have 8 GB RAM for a smooth development and testing experience.
* It should use solid state drives as they have faster data access than their older HDD counterparts which makes for a smoother development and testing experience.
* Any modern operating system will fulfill the requirement as long as it is compatible with development tools.
* The display should have a resolution of at least 1920x1080 to maximize productivity and provide an immersive experience for the users.
* Keyboard, mouse, and other input devices required for a smooth coding and developing experience.

## Deployment Servers Requirements:

* The web application may have a medium to high load so the deployment server should at least have 16 GB RAM for handling concurrent users and higher complexity in quizzes.
* The system should at least have a multi-core processor to handle simultaneous requests.
* It will use clustered database architecture with servers spread across multiple regions to ensure high availability and scalability.
* It will have a load balancer that analyzes incoming requests and diverts those to the nearest server or if the nearest server is near capacity, divert the request to the next nearest server that is not at capacity. This reduces latency and maximizes response time.
* It should use solid state drives (SSDs) to maximize performance, availability and scalability, by ensuring faster data access.
* The SSD should be adequate for storing user data, quizzes, and quiz reports.
* The server will also use network attached servers to balance scalability and performance.
* It will use RAID (Redundant Array of Independent Disks) storage as it provides redundancy or backups in case primary storage drives fail.
* As the web application will have a medium to high load so the deployment servers should atleast have a bandwidth of 1 Gbps(GigaBits Per Second).

# Who Did What?

| **Name of the Team Member** | **Tasks done** |
| --- | --- |
| Abdul Muiz | Architecture Diagram, Architecture Description, Review |
| Bisma Nawaz | Tools and technologies |
| Hafsa Ahmed | Hardware Requirements |
| Abdur Rafae Haroon | Hardware Requirements |
| Waleed Nadeem | Justification of the architecture |

# Review checklist

| **Section** **Title** | **Reviewer Name(s)** |
| --- | --- |
| 2.3, 4, 5 | Abdul Muiz |
| 2.3, 5 | Bisma Nawaz |
| 4, 5 | Hafsa Ahmed |
| 4,5 | Abdur Rafae Haroon |
| 2,4 | Waleed Nadeem |