Software Requirements Specification (SRS) for Real-Time Multiplayer Quiz Game

1. Introduction

1.1 Purpose: This SRS defines the functional and non-functional requirements for a real-time multiplayer quiz game where users can select subjects, get matched with teammates, compete against another team, answer questions in real time, and view leaderboards based on score and speed.
1.2 Scope: The system will encompass user registration and authentication, subject selection, real-time matchmaking, quiz game logic, score calculation, leaderboards (global and location-based), and real-time communication between users and the server.

2. Overall Description

2.1 Product Perspective: The quiz game is envisioned as a standalone application (web and/or mobile) that leverages a backend server for real-time functionalities.
2.2 Product Features:

* User authentication (signup/login)
* Subject selection
* Real-time matchmaking for 2v2 matches
* 60-second real-time quiz game
* Score calculation based on correctness and speed
* Winning team declaration
* Global and location-based leaderboards
* Real-time updates during the game (e.g., question display, timer, scores)

2.3 User Classes and Characteristics:
Players: Users who register, participate in quizzes, and view leaderboards.
2.4 Operating Environment:

* The system will function on modern web browsers and/or mobile devices.
* Backend services will run on a server infrastructure.
* Requires stable internet connectivity.

2.5 Assumptions and Dependencies:

* Users have access to compatible devices and the internet.
* Reliance on external services for quiz questions (if applicable).
* Server infrastructure is available and reliable.

3. Specific Requirements

3.1 Functional Requirements:
1: User Authentication:

* Users shall be able to register with a unique username and password.
* Registered users shall be able to log in securely.

2: Subject Selection:

* Users shall be able to choose a subject category for the quiz.

3: Matchmaking:

* The system shall automatically match a user with a teammate based on subject selection.
* The system shall match the formed team of two against another team of two.

4: Quiz Game:

* The system shall initiate a 60-second quiz upon successful matchmaking.
* The system shall display questions and answer options in real time.
* Users shall be able to select an answer within the time limit.

5: Scoring:

* The system shall calculate individual and team scores based on correct answers and response speed(Will manage response speed based on api hit time).

6: Winning Team Declaration:

* The system shall declare the winning team based on the final scores.

7: Leaderboards:

* The system shall display global and location-based leaderboards with updated rankings.

8: Real-time Communication:

* The system shall utilize WebSockets or similar technology for real-time updates of game state, scores, and timer.

3.2 Non-Functional Requirements:

1: Performance:

* The system shall provide a low-latency experience for real-time gameplay.
* The matchmaking process should be fast and efficient.

2: Scalability:

* The system should be able to handle a growing number of concurrent users and games.

3: Reliability:

* The system should be robust and minimize downtime during quizzes.

4: Security:

* User data and quiz content should be protected.
* Secure authentication mechanisms should be implemented.

5: Usability:

* The user interface should be intuitive and easy to navigate.

4. System Architecture

4.1 High-Level Architecture:

* A typical architecture for a real-time multiplayer game like this involves a client (web/mobile app), backend services, and a real-time communication layer.

4.2 Components:

Client (User Interface): Handles user interaction, subject selection, quiz display, and real-time updates.

API Gateway: Serves as the entry point for client requests, routing them to appropriate backend services.

Authentication Service: Manages user registration and login.

Matchmaking Service: Matches players based on subject selection and creates teams.

Game Session Service: Manages the state of active quiz games.
Quiz Service: Provides quiz questions and answers.
Score Service: Calculates scores based on correctness and speed.
Leaderboard Service: Manages and updates global and location-based leaderboards.
Real-time Communication Service (e.g., WebSocket Server): Facilitates real-time communication between clients and the server for game updates, timer synchronization, and score updates.
Database: Stores user data, quiz content, scores, and leaderboard information.

4.3 Technologies:

Frontend: React, Vue.js, Flutter, etc.
Backend: Python/FastAPI
Database: PostgreSQL, MongoDB, etc.
Cloud Platform (Hosting): AWS, Google Cloud, etc.

5. APIs

5.1 API Design Principles:

* RESTful APIs for core functionalities (authentication, profile management).
* WebSocket connections for real-time game communication.
* Secure endpoints with authentication and authorization.

5.2 Key API Endpoints (Examples):

Authentication:
* POST /register: User registration
* POST /login: User login

Matchmaking:
* POST /matchmaking/join: Request to join the matchmaking queue
* DELETE /matchmaking/leave: Request to leave the matchmaking queue

Game:
* GET /quiz/subjects: Get available subjects
* POST /game/start: Start a new game
* POST /game/{game_id}/answer: Submit an answer for a question

Leaderboard:
* GET /leaderboards/global: Get global leaderboard
* GET /leaderboards/location/{location_id}: Get location-based leaderboard

5.3 Real-time Communication (WebSocket Events):

Server to Client:

* match_found: Notification when a match is found
* game_started: Notification when the quiz begins
* new_question: New quiz question and options
* timer_update: Updates on the remaining time
* score_update: Real-time score updates for all players
* game_ended: Notification when the quiz is over
* leaderboard_update: Updated leaderboard data

Client to Server:

* submit_answer: User submits an answer
* player_ready: User is ready to start the game

6. Data Model (Simplified)

User:
* user_id (unique identifier)
* username
* password (hashed)
* location
* global_score
* location_score

Quiz:
* quiz_id
* subject
* questions (array of question objects)

Question:
* question_id
* text
* options (array of answer options)
* correct_answer

Game Session:
* game_session_id
* team1_user_ids (array of user IDs)
* team2_user_ids (array of user IDs)
* quiz_id
* current_question_index
* team1_score
* team2_score
* start_time
* end_time
* status (e.g., "in_progress", "completed")

Leaderboard:
* user_id
* global_rank
* location_rank

Questions

1. How would you handle real-time communication and synchronization between players?

   Ans: To maintain real-time communication a websocket based approach would be best suitable.

   All participants in a match join the same room, enabling:

   a. Real-time question broadcast

b. Timer countdown sync
c. Instant score updates
d. Notification of match start/end

2. What strategy would you use for scaling the matchmaking service under heavy load?

   Ans: Using a sharded Queue based solution for speed, scaling, and efficiency would be suitable.
   Horizontally scale stateless matchmaking workers that pull users from queues and group them into matches.
   Add priority queues or bucketing by subject, skill level, or location to improve pairing quality and reduce latency.
   Use a circuit breaker and backoff mechanism to reject new match requests gracefully when overloaded.

3. How would you ensure the scoring service meets the performance SLA (p95 < 200ms)?

Ans: The best way to do this would be by applying an async process of storing scores, while performing the scoring in memory or using memory based DBs.

4. If the quiz involved dynamic or user-generated questions, how would that impact your design?

   Ans: Dynamic Question or UserGenerated Questions can be handled in many ways, Questions such as do we need to moderate the questions? Should the scores be added to the leaderboard? And more needs to be answered first. But taking a base assumption we can do this by implementing a question approval strat(Pending/Approved/Rejected) or for a private game. Keeping the questions temporarily in the memory or a different Table with player Id as the foreign key.

   This will bring in features like question tagging(user/system), schema changes for questions Table such as (creator_id, created_at, difficulty_status)

   Putting limitations to the new APIs that will be created to stop misuse is also an important part.