System Design Document: Real-Time Vocabulary Quiz Application

Overview

This document outlines the system design for a real-time vocabulary quiz application within an English Learning App. The application enables users to join quiz sessions, submit answers, and view real-time leaderboard updates. The design focuses on scalability, low-latency interactions, and consistent data handling across multiple users.

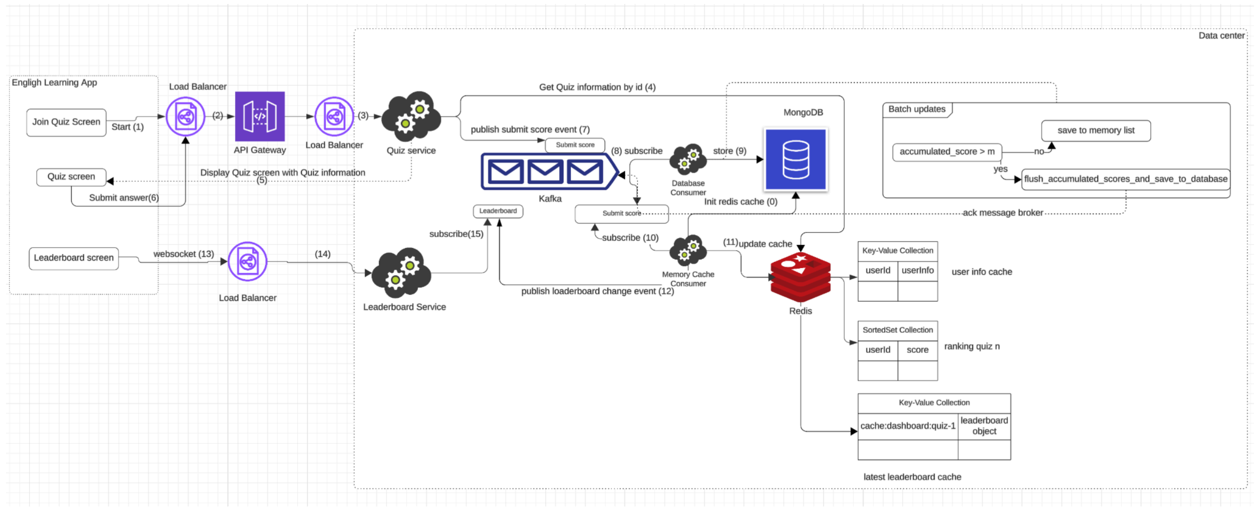
1. Requirements

- Functional Requirements
  - User Participation: Users can join a quiz session using a unique quiz ID, with support for multiple simultaneous participants.
  - Real-Time Score Updates: Scores update in real-time as users submit answers, ensuring accuracy and consistency.
  - Real-Time Leaderboard: A leaderboard displays current standings and updates promptly as scores change.
- Non-Functional Requirements.
  - Ensure the code meets criteria such as scalability, performance, reliability, maintainability, and observability.
- Need to clarify the requirements:
  - We support multiple quiz leaderboards.
  - Anonymous users are not supported.

- Users must log in and obtain an access token via the authentication service.

2. System Architecture

The architecture leverages microservices, caching, web socket and message queues to ensure scalability
and real-time performance. Key components include a Flutter-based frontend, Google Cloud infrastructure, and backend services using Express.js, Go, and databases like MongoDB and Redis. The system flow is illustrated in the architecture diagram.



3. Data flow

3.1. Real-time Quiz Participation

- **(0)** When the Redis consumer starts, it loads necessary data from MongoDB.
- **(1)** The user enters a quiz ID and clicks the “Join quick” button. The client sends a request to retrieve the quiz information.
- **(2)** The load balancer forwards the client’s request to the nearest API Gateway, which authorizes and forwards it to the Quiz Service.
- **(3)** The load balancer routes the request to the closest API Gateway and data center for optimized latency.
- **(4)** The Quiz Service attempts to retrieve the quiz information from the Redis cache. If not found, it queries the database.
- **(5)** If the quiz information is successfully retrieved, the user is navigated to the Quiz screen. Otherwise, a popup is shown indicating the failure.

3.2. Real-time Score Updates

- **(6)** When the user submits an answer, it is sent to the Quiz Service via the API Gateway and Load Balancer.
- **(7)** The Quiz Service validates the answer and publishes a score update message to Kafka.

- **(8)** The Database Consumer subscribes to the "submit score" topic in the Message Broker.
- **(9)** The Database Consumer updates the database with the new score.
- **(10)** The Memory Cache Consumer also subscribes to the "submit score" topic.
- **(11)** The Memory Cache Consumer updates the in-memory cache with the latest score.

3.3. Real-time Leaderboard

- **(12)** The Memory Cache Consumer publishes leaderboard changes to the Message Broker.
- **(13)** When the user opens the Leaderboard screen, the client establishes a WebSocket connection through the Load Balancer for real-time updates.
- **(14)** The Load Balancer routes the WebSocket connection to the nearest Leaderboard Service.
- **(15)** The Leaderboard Service subscribes to the "leaderboard" topic, applies throttling, and sends leaderboard and user score updates to the client via the WebSocket connection.

4. Components

4.1. Frontend (Client-Side): English Learning App - Flutter

- **Join Quiz Screen**: Allow users to join a quiz session using a unique quiz ID.
- **Quiz Screen**: Displays quiz questions with and answers
- **Leaderboard Screen**: Shows player rankings based on quiz performance. Each quiz will have a separate leaderboard
- **Communication**: Uses HTTP and WebSocket (via **web_socket_channel** package).
- **Why Flutter?** Cross-platform support, near-native performance with Skia, and a rich ecosystem of packages.

4.2. Load Balancers - AWS Elastic Load Balancing (ELB)

- **Purpose**: Balances traffic at two levels: client requests (Application Load Balancer) and inter-service communication (Network Load Balancer).
- **Why AWS ELB?** Provides global traffic distribution, supports WebSocket for real-time updates, and offers autoscaling to handle varying loads. AWS's global network ensures low latency and high availability.

4.3. API Gateway - AWS API Gateway

- **Functionality**: Unified entry point for client API calls, handling routing, authentication (via AWS IAM or Lambda authorizers), rate limiting, and throttling.
- **Why AWS API Gateway?** Fully managed, integrates with AWS services like Lambda, CloudWatch (for logging), and AWS WAF (for security). Supports WebSocket APIs for real-time communication and scales automatically to handle traffic spikes.

4.4. Quiz Service

- **Role**: Manages quiz logic, fetches quiz data from Redis, and publishes score events to Kafka.
- **Tech Stack**: Uses ioredis for Redis access and kafkajs for Kafka integration.
- **Why Express.js**? Fast, asynchronous handling of quiz logic, seamless integration with Redis and Kafka.

4.5. Message Queue - Kafka

- **Topics**: submit_score and leaderboard.
- **Purpose**: Decouples services for real-time score updates and leaderboard changes.
- **Why Kafka?** Ensures real-time event delivery, consistency, durability, and fault tolerance through replication.

4.6. Memory cache - Redis

Redis is used as an in-memory caching layer to store and manage frequently accessed data, enabling fast retrieval and updates for real-time quiz interactions. It handles three key data structures:

Key-Value Collection: Maps userId to userInfo (User Info Cache), allowing the Quiz Service to quickly fetch user details by ID.

Sorted Set: Maps userId to score (Per Quiz Ranking), maintaining a real-time ranking of participants for each quiz, which the Leaderboard Service queries for efficient leaderboard updates.

Key-Value Collection: Stores leaderboard data (e.g., cache:dashboard:quiz-1), caching the latest leaderboard state to reduce database load and speed up client requests.
Redis is queried directly by the Quiz and Leaderboard Services and updated by the Memory Cache Consumer (Golang) upon receiving submit_score events from Kafka. Its high performance, support for sorted sets, and scalability through replication make it ideal for handling multiple quiz sessions and users concurrently.

4.7. Database - MongoDB

- **Role**: Persistent storage for quiz results.
- **Why MongoDB?** Flexible schema for varying quiz formats, horizontal scalability, and Serverless options reduce DevOps overhead.

4.8. Consumers - Golang

- **Database Consumer**: Subscribes to submit_score topic, writes results to MongoDB.
- **Memory Cache Consumer**: Subscribes to submit_score topic, updates Redis, and publishes leaderboard updates.
- **Why Go?** Fast execution, lightweight memory usage, efficient concurrency with goroutines,

and simple deployment.

4.9. Leaderboard Service - Express.js (NodeJS)

- **Role**: Handles leaderboard logic, subscribes to leaderboard topic, reads from Redis, and serves updates via WebSocket. •
- **Why Express.js?** Non-blocking I/O, WebSocket support, fast development, and horizontal scalability.

5. Discussion

- To reduce database reads and writes, we can implement a batch update mechanism at the database consumer. However, this requires additional handling—such as acknowledgments—to ensure data integrity and prevent loss. For real-time updates, we rely on a Redis consumer service.
- To avoid WebSocket connection overload, I implemented a throttling mechanism that emits events every *n* milliseconds (e.g., 200ms) per quiz ID. As a result, leaderboard updates may experience a slight delay and might not appear perfectly real-time. However, we assume this delay is negligible and does not impact the user experience.
- To display the current standings of all participants in real-time, we optimize for performance by calculating only the delta and publishing changes via WebSocket. When a large number of users join a quiz, the leaderboard can become quite large. This approach reduces the amount of data sent, but it may introduce a slight delay due to the delta computation.
- We notify all users whenever the leaderboard changes, which can result in high network load due to the volume of notifications. However, based on the business requirements, we assume that all such notifications are necessary and there are no redundant updates.
- Kafka was selected instead of RabbitMQ because it offers higher throughput and better data durability—both of which are critical requirements for our system. This choice, however, involves a more complex infrastructure setup.
- We chose MongoDB (NoSQL) over PostgreSQL (SQL) because our data model is relatively simple, without complex relationships or the need for join-heavy queries. With Redis serving as the primary data access layer, our focus is on achieving high write throughput and horizontal scalability through sharding and replication. However, this architecture requires additional mechanisms to ensure data consistency and support transactional operations.
- Load balancing distributes network traffic across multiple servers to enhance performance, scalability, and reliability. It can introduce complexity, potential latency, and increased costs due to additional infrastructure or configuration. Proper implementation ensures high availability but requires careful management to avoid single points of failure.
- Consider integrating a logging service (e.g., Grafana with Loki + Prometheus) to monitor and visualize service logs. This adds observability but may increase infrastructure complexity and resource usage.
- Expose a `/health` endpoint that returns the status of services