Technical Design Document

Overview

Project Name

Real-Time Vocabulary Quiz Coding Challenge

Purpose

This project aims to implement a real-time quiz feature within an English learning application, allowing users to participate in quizzes, answer questions, and see live score updates on a leaderboard.

Scope

1. In scope

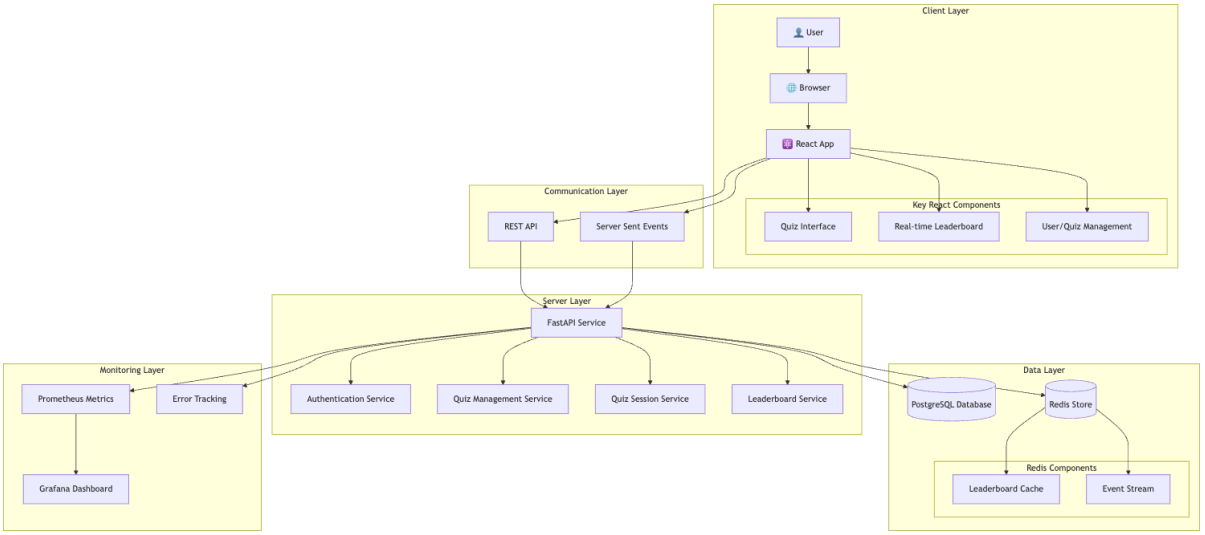
- Admin users should be able to create a quiz and add questions and answers.
- Users should be able to join a quiz using its identifier.
- Users should see the latest leaderboard of all the quiz updated live as other participants are going through the quiz.

2. Out of scope

- Quiz session resume functionality is not considered in this round.
- Currently only multiple choice questions with 1 correct answer is supported.
- Questions and answers only support text-based at the moment.

System Design

1. Architecture Diagram



2. Component Description

Client Layer

- User: The participants accessing the quiz interface through a web browser.
- Browser: The environment where the React app runs.
- React App: The front-end application that allows users to interact with the quiz and see live updates.

Communication Layer

- SSE (Server-Sent Events) : Enables real-time communication between the client and server to update scores live.
- REST API: Allows clients to fetch quiz data, submit answers, and retrieve leaderboard information.

Server Layer

- FastAPI: The core backend service that handles requests and orchestrates the functionality of various services.
- Authentication Service: Manages user authentication and authorization for quiz participation.

- Quiz Management Service: Handles quiz creation, session management, and user participation.
- Quiz Session Service: Manages specific quiz sessions and user states within those sessions.
- Leaderboard Service: Responsible for maintaining and updating leaderboard data based on scoring.

Data Layer

- PostgreSQL Database: Stores persistent data such as user information, quizzes, and historical scores.
- Redis Store: Provides caching and real-time data handling with sessions and leaderboard features.
    - Leaderboard Cache: Caches the leaderboard for quick access and updates.
    - Event Stream: Manages real-time events emitted during quiz participation.

Monitoring Layer

- Prometheus: Collects metrics regarding the application's performance and status.
- Grafana: Provides a visual dashboard for monitoring metrics and understanding system health.
- Error Tracking: Monitors and logs errors for feedback and troubleshooting.

---

3. Data Flow Explanation

- User Participation:
    - Users connect to the React app via a web browser and join a quiz session by providing a unique quiz ID.
    - The React app sends this ID to the FastAPI server through a REST API.
- Real-Time Score Updates:
    - As users answer questions, their submissions are sent to the Quiz Session Service via the REST API.
    - The server validates and updates user scores in the PostgreSQL database and Redis cache.
    - An event is emitted to the SSE, which alerts connected clients of the score update.

- Real-Time Leaderboard:
    - The Leaderboard Service fetches current scores from Redis and constructs/upgrades the leaderboard.
    - Using SSE, the updated leaderboard is sent to all connected clients, allowing for real-time updates in the React app.
    - Players see their rankings change dynamically as data flows from the quiz submissions to the leaderboard updates.

---

4. Technologies and Tools

- Frontend:
    - React: For building the user interface components, providing a dynamic and interactive experience.
    - Chakra UI: For styled components, enhancing the visual elements of the quiz.
- Backend:
    - FastAPI: A modern, fast (high-performance) web framework for building APIs with Python.
    - PostgreSQL: A powerful, open-source object-relational database for storing and managing quiz and user data.
    - Redis: Used for caching, managing event streams, and providing real-time data handling.
- Monitoring:
    - Prometheus: For monitoring metrics and querying performance data.
    - Grafana: To visualize metrics and create dashboard views.
    - Error Monitoring Tools: To track exceptions and failures in the system.

Implementation

Component: Real-time Leaderboard

Functional Requirements

- Users can view a live leaderboard that updates automatically as scores change.
- The leaderboard maintains accurate rankings based on real-time submissions.

Scalability

- Utilizing Redis for caching allows the system to handle large volumes of concurrent users and dynamic updates without performance degradation.

Performance

- Frequent leaderboard updates via SSE keep user interfaces responsive and provide immediate feedback.

Reliability

- Error handling in the backend ensures that faults do not impede user experience; for example, using try-catch structures to handle potential failures gracefully.

Maintainability

- The code is modular, following best practices for organization and readability, facilitating easy updates and collaborative work among developers.

Monitoring and Metrics Collection

Metrics to Collect

- API Performance Metrics:
    - Response Time: Measure the time taken to respond to API requests, grouped by endpoint.
    - Request Rate: Track the number of requests per second for each API endpoint.
    - Error Rate: Monitor the percentage of failed requests (e.g., HTTP 4xx/5xx errors) over a period.
- User Engagement Metrics:
    - Active Users: Count of concurrent connected users participating in quiz sessions.
    - Quizzes Started: Track how many quizzes are initiated during a certain period.
    - Submissions per Quiz: Monitor the average number of answers submitted per quiz session.
- Leaderboard Metrics:
    - Leaderboard Update Frequency: Number of real-time updates sent to connected clients per time unit.
    - Scores Updated: Frequency of score changes, indicating user participation and engagement.
- Database Performance Metrics:
    - Query Response Times: Measure the execution time for key database queries (e.g., fetching leaderboard data).
    - Connection Pool Usage: Monitor the number of active database connections and their utilization.
- Redis Performance Metrics:
    - Cache Hit Rate: Ratio of cache hits to total requests accessing Redis, useful for evaluating cache effectiveness.
    - Event Stream Latency: Time taken to process messages from the Redis event stream before they are sent to clients.

How to Collect Metrics

- Prometheus:
    - Use Prometheus client libraries (e.g., prometheus_fastapi_instrumentator for FastAPI) to instrument your application:
        - Add metrics to track request counts, latency, error rates, and custom business logic metrics.
        - Set up histograms for response times and counters for request counts.
- Database Monitoring:
    - Use PostgreSQL extensions like pg_stat_statements to collect statistics about query performance.
    - Configure database logging to log slow queries for analysis.

- Redis Monitoring:
    - Use built-in Redis monitoring tools such as INFO commands to collect usage metrics such as memory, command statistics, and keyspace hits.
    - Alternatively, consider Redis exporter for Prometheus to scrape these metrics periodically.
- Logging:
    - Implement structured logging across your application to capture detailed logs of interactions, API calls, errors, and exceptions.
    - Use logs to track user engagement and actions, which can later be analyzed for performance insights.
- Visualization with Grafana:
    - Set up dashboards in Grafana to visualize the metrics collected by Prometheus.
    - Create queries to display key metrics over time, allowing for easy monitoring of performance and health.

Strategies for Scaling the Application

For Medium traffic (1,000-10,000 rps)

- **Horizontal Scaling**:
    - Backend Service Instances: Deploy multiple instances of the FastAPI service behind a load balancer (like Traefik, for example) to distribute incoming requests across multiple containers efficiently.
    - Redis Clustering: Use Redis in cluster mode to enhance performance and ensure availability across a wider range of traffic.
- **Database Optimization**:
    - Read Replicas: Set up read replicas for PostgreSQL to offload read-heavy operations from the primary database. This helps improve performance when handling more read requests, such as leaderboard queries.
    - Connection Pooling: Utilize connection pooling in your FastAPI service to optimize the number of concurrent connections to PostgreSQL.
- **Asynchronous Processing**:
    - Introduce background processing to handle long-running tasks (e.g., scoring updates, leaderboard maintenance) using Celery or another task queue library with Redis or RabbitMQ.
- **Monitoring and Auto-scaling**:
    - Using the monitoring metrics described above to set up alerts for when a component of the application needs scaling.

For Very High traffic (> 10,000 rps)

Consider moving the whole stack to Kubernetes for better scalability. However, managing kubernetes cluster can be either sophisticated (if doing on-premise) or expensive (cloud-managed cluster)