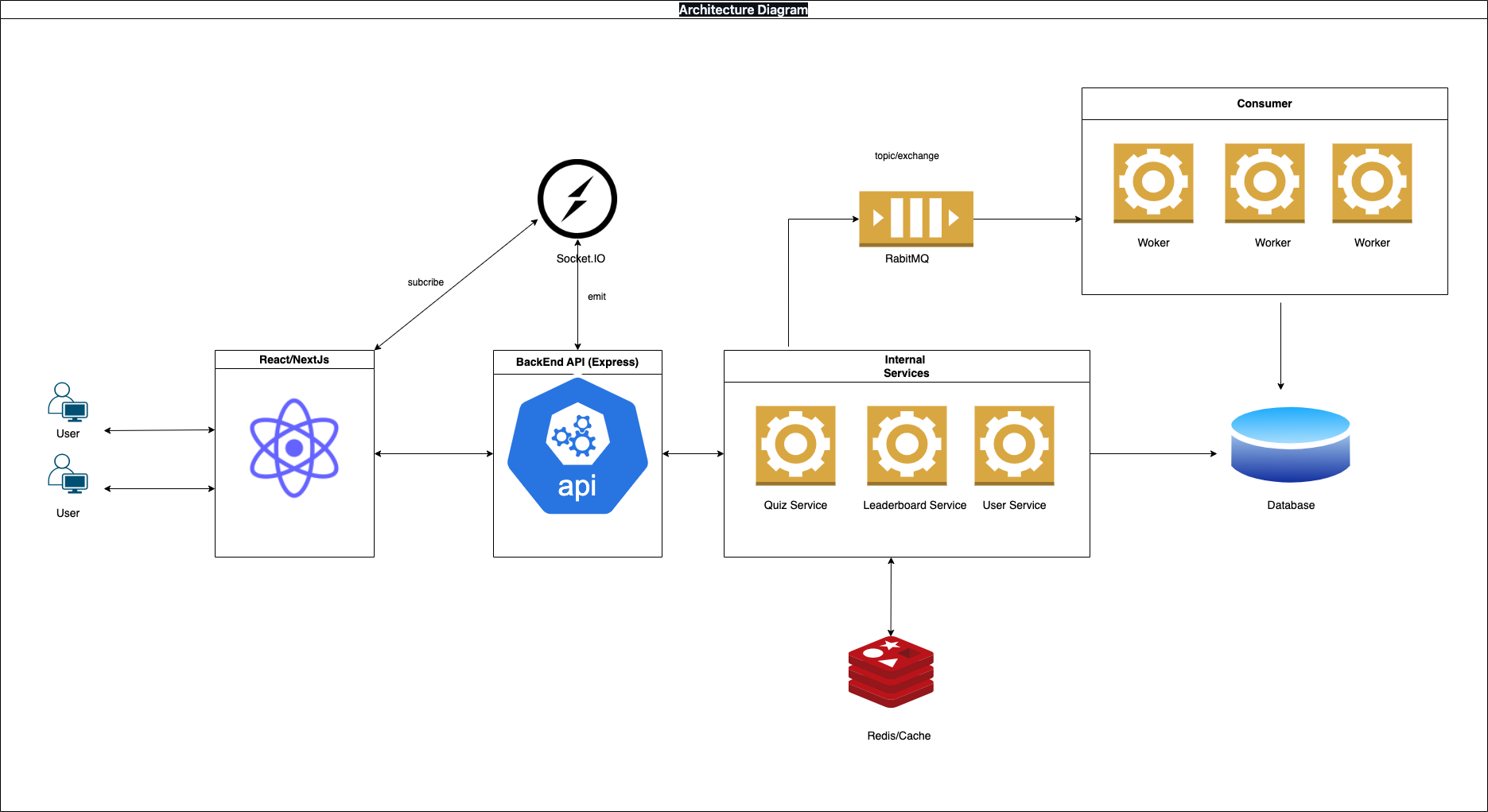
# System Design Document

## 1. Overview

This document outlines the architecture, component interactions, data flow, and technology choices for a quiz application with real-time leaderboard updates. The system enables users to participate in quizzes, submit answers, and see updated leaderboards based on their performance.

## 2. Architecture Diagram

The architecture consists of the following components:

* **Client Application**: User interface built with React/Next.js that enables users to participate in quizzes.
* **Backend Server**: Node.js server with Express handling API requests, quiz data, user authentication, and score processing.
* **Database**: PostgreSQL database using Prisma ORM to store user, quiz, question, and score data.
* **Redis**: Used as an in-memory cache for leaderboards and user scores for efficient retrieval and real-time updates.
* **WebSocket Server**: Facilitates real-time communication between the server and clients for updating scores and leaderboards.
* **RabbitMQ**: Message broker for handling asynchronous tasks like updating the global leaderboard.
* 

## 3. Component Description

### 3.1 ****Client Application****

* **Description**: The client application is responsible for user interaction, including login, joining quizzes, submitting answers, and viewing real-time leaderboard updates.
* **Technologies**: React, Next.js, WebSocket client.
* **Responsibilities**:
  + Authenticate the user and maintain session state.
  + Provide quiz and leaderboard UI.
  + Display real-time leaderboard updates.

### 3.2 ****Backend Server****

* **Description**: The backend server handles authentication, quiz logic, and leaderboard updates.
* **Technologies**: Node.js, Express, Prisma (ORM), WebSocket server.
* **Responsibilities**:
  + Expose RESTful APIs for quiz data and user interactions.
  + Calculate user scores and update quiz completion history.
  + Emit WebSocket events for real-time leaderboard updates.

### 3.3 ****Database (PostgreSQL)****

* **Description**: Stores persistent data for users, quizzes, questions, options, and scores.
* **Technologies**: PostgreSQL, Prisma ORM.
* **Responsibilities**:
  + Persist user and quiz data.
  + Track score history and answer submission history.
  + Support complex queries for calculating total user scores and leaderboard data.

### 3.4 ****Redis (Cache)****

* **Description**: Caches frequently accessed data, like leaderboards and individual user scores, to improve performance.
* **Technologies**: Redis.
* **Responsibilities**:
  + Store the global leaderboard and individual user scores for quick retrieval.
  + Update the leaderboard in real-time based on events from the backend.
  + Serve as a rate-limiting store for API calls if necessary.

### 3.5 ****WebSocket Server****

* **Description**: Handles real-time communication between clients and the server.
* **Technologies**: WebSocket (integrated with the backend server).
* **Responsibilities**:
  + Notify clients of leaderboard and user score updates in real-time.
  + Broadcast events when a user’s score changes.
  + Support multiple concurrent connections for real-time updates.

### 3.6 ****RabbitMQ (Message Broker)****

* **Description**: Message queue for handling asynchronous tasks.
* **Technologies**: RabbitMQ.
* **Responsibilities**:
  + Handle background tasks, like updating the leaderboard without blocking the main server.
  + Allow distributed processing of leaderboard updates.
  + Provide a decoupled mechanism for updating scores across multiple components.

## 4. Data Flow

### 4.1 ****User Joining a Quiz****

1. **User Login**:
   * The user logs in via the client application.
   * An authentication token is generated and stored in a cookie.
2. **Joining a Quiz**:
   * The client requests quiz data from the backend.
   * The backend fetches quiz details from the database and sends them back to the client.

### 4.2 ****Submitting Answers****

1. **Answer Submission**:
   * The user submits answers to quiz questions via an API call.
   * The backend validates answers, calculates the score, and updates UserQuizScore and UserQuizAnswerHistory in the database.
2. **Score Update**:
   * If the score changes, the backend updates the user's total score in Redis.
   * A RabbitMQ message is sent for asynchronous leaderboard processing.

### 4.3 ****Updating the Leaderboard****

1. **Leaderboard Update**:
   * Redis is queried for the user’s updated score.
   * If the score is new or higher, it’s added or updated in the global:leaderboard list in Redis.
   * A WebSocket event (userScoreUpdated) is emitted to connected clients to update the leaderboard in real-time.

### 4.4 ****Real-time Leaderboard Display****

1. **Initial Load**:
   * The client requests the current leaderboard from Redis.
2. **Real-time Updates**:
   * The WebSocket server broadcasts updates to clients whenever scores are updated in the global leaderboard.

## 5. Technologies and Tools

### 5.1 ****Frontend****

* **React & Next.js**: Chosen for their modularity, fast client-side navigation, and SEO optimization with server-side rendering.
* **WebSocket Client**: Enables real-time communication for updating leaderboards without refreshing the page.

### 5.2 ****Backend****

* **Node.js & Express**: Chosen for their non-blocking I/O model, scalability, and compatibility with WebSocket integration.
* **Prisma ORM**: Provides an abstraction over PostgreSQL, allowing for complex data relationships, type-safe database access, and easy integration with Node.js.

### 5.3 ****Database****

* **PostgreSQL**: Relational database chosen for its reliability, ACID compliance, and support for complex queries needed for calculating user scores and leaderboard data.

### 5.4 ****Cache****

* **Redis**: Used as an in-memory cache for leaderboards, providing fast retrieval and efficient storage of high-frequency data.
* **Justification**: Redis enables quick leaderboard updates and is optimized for sorted sets, which is ideal for leaderboard ranking.

### 5.5 ****Real-time Communication****

* **WebSocket Server**: Allows real-time updates of scores and leaderboards. It provides low latency for instant updates across multiple clients.

### 5.6 ****Asynchronous Processing****

* **RabbitMQ**: Message broker used for handling asynchronous leaderboard updates, improving response times for user interactions.
* **Justification**: Decouples heavy background tasks from the main request cycle, preventing delays in user-facing processes.