Real-Time Quiz challenge

Quiz Game Application Documentation

Project Overview

This is a multiplayer quiz game application built with TypeScript, featuring a layered architecture that separates concerns and maintains testability.

Core Features

Players can join/leave quiz sessions

Multiple players can participate in the same quiz

Real-time answer processing

Score tracking for participants

Leaderboard functionality (prepared for Redis implementation)

Architecture

1. Layer Structure

```
src/

├── model/       # Domain models

├── dto/         # Data transfer objects

├── repository/  # Data access layer

├── service/     # Business logic

└── constants/   # Application constants

tests/
```

```
└── service/     # Unit tests
```

2. Key Components

Models

The domain models (QuizModels.ts) define core entities:

Quiz: Represents a quiz session with words, hints, and current progress

Participant: Represents a player in a quiz session

Reference:

```
export interface Participant {
    quizId: string;
    username: string;
    score: number;
}
export interface QuizSession {
    quizId: string;
    words: string[];
    hints: string[];
    currentIndex: number;
    createdAt?: Date;
    updatedAt?: Date;
    participants: Participant[];
}
```

Data Transfer Objects (DTOs)

DTOs handle data transfer between layers and define API contracts. Examples include:

QuizJoinDto: For joining a session

QuizAnswerDto: For submitting answers

QuizLeaveDto: For leaving a session

Reference:

```
export interface QuizJoinDto {
```

```
    quizId: string;
    username: string;
}
export interface QuizLeaveDto {
    quizId: string;
    username: string;
}
export enum ActionType {
    JOIN_ACK = 'joinAck',
    LEAVE_ACK = 'leaveAck'
}
export interface QuizResponseDto {
    type: ActionType;
    quizId: string;
    username: string;
}
export interface QuizJoinResult {
    quizSession: QuizSession;
    participant: Participant;
    hints: string[];
    currentIndex: number;
}
```

Repositories

Two main repositories handle data persistence:

SessionRepository: Manages quiz sessions

SessionParticipantRepository: Manages participant data

Both currently use in-memory storage but are structured to easily switch to a real database.

Services

Services implement business logic:

QuizService: Handles core quiz operations

LeaderboardService: Manages scoring (prepared for Redis implementation)

Testing Strategy

1. Unit Tests

Each service has corresponding unit tests

Mocks are used for repository dependencies

Tests cover success and error scenarios

Example test structure:

```
describe('QuizService', () => {

    const mockQuizSession: QuizSession = {

        quizId: 'test-quiz',

        participants: [],

        words: ['test'],

        hints: ['test hint'],

        currentIndex: 0,

        createdAt: new Date('2020-01-01T00:00:00Z'),

        updatedAt: new Date('2020-01-01T00:00:00Z')

    };

    const mockParticipant: Participant = {

        quizId: 'test-quiz',

        username: 'test-user',

        score: 0

    };

    beforeEach(() => {

        jest.clearAllMocks();

    });
```

```
    describe('joinQuizSessionWithUsernameAndQuizId', () => {
        test('should create new session and add participant when session does not
exist', async () => {
            (quizRepository.getQuizSessionById as jest.Mock).mockResolvedValue(null);
            (quizRepository.createNewQuizSession as
jest.Mock).mockResolvedValue(mockQuizSession);
            (participantRepository.findParticipantInSession as
jest.Mock).mockResolvedValue(null);
            (participantRepository.addParticipantToSession as
jest.Mock).mockResolvedValue({
                session: mockQuizSession,
                participant: mockParticipant
            });
            const result = await quizService.joinQuizSessionWithUsernameAndQuizId({
                quizId: 'test-quiz',
                username: 'test-user'
            });
            expect(quizRepository.getQuizSessionById).toHaveBeenCalledWith({
                quizId: 'test-quiz'
            });
            expect(quizRepository.createNewQuizSession).toHaveBeenCalled();
            expect(participantRepository.addParticipantToSession).toHaveBeenCalled();
            expect(result).toEqual({
                quizSession: mockQuizSession,
                participant: mockParticipant,
                hints: mockQuizSession.hints,
                currentIndex: mockQuizSession.currentIndex
            });
        });
```

```
        test('should return error when participant already exists', async () => {
            (quizRepository.getQuizSessionById as
jest.Mock).mockResolvedValue(mockQuizSession);
            (participantRepository.findParticipantInSession as
jest.Mock).mockResolvedValue(mockParticipant);
            const result = await quizService.joinQuizSessionWithUsernameAndQuizId({
                quizId: 'test-quiz',
                username: 'test-user'
            });

expect(participantRepository.addParticipantToSession).not.toHaveBeenCalled();
            expect(result).toEqual({
                error: QuizServiceErrorMessages.USER_ALREADY_JOINED
            });
        });
    });
```

2. Test Coverage

Tests cover:

Session creation/joining

Participant management

Answer processing

Error handling

Design Patterns Used

Repository Pattern: Abstracts data access

DTO Pattern: Manages data transfer between layers

Singleton Pattern: Used for service instances

Dependency Injection: Through constructor injection in services

Future Improvements

Database Integration

Current in-memory storage can be replaced with:

PostgreSQL for quiz and participant data

Redis for leaderboard and real-time features

Real-time Features

WebSocket integration for live updates

Real-time leaderboard updates

Scalability

Session management across multiple servers

Caching layer for quiz data

Message queue for answer processing

Development Workflow

Feature Development

```
git checkout -b feat/feature-name

# Make changes

git add .

git commit -m "feat: description"
```

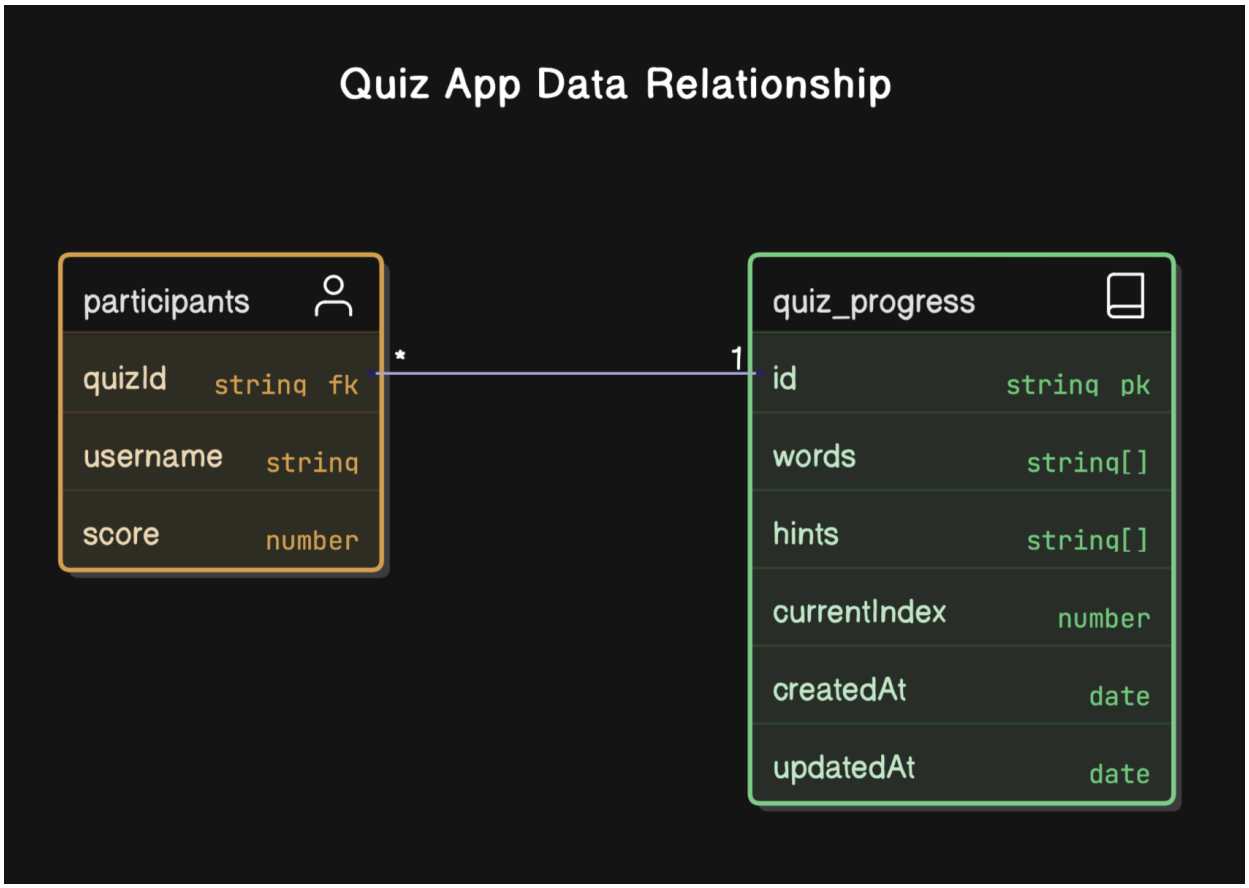
Testing

Write unit tests alongside feature development

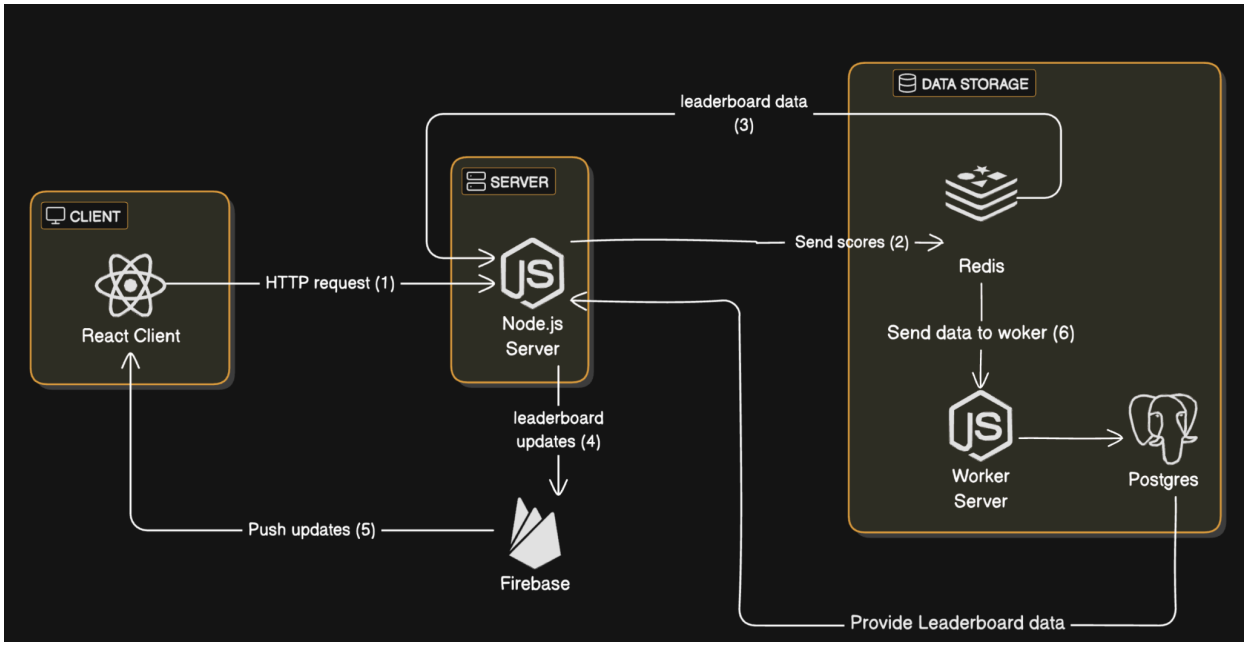
Run tests: `npm test`

I. Diagram

Database design: We have multiple participants who can join one quiz progress or quiz session as below. Participants table has a primary key on col (quizId, username) to determine where users are participating at.



This diagram outlines the interactions between various components



Our system architecture consists of an Authentication Server and Quiz

Server that are the “NodeJS server" in the diagram. Leaderboard Service, and a Client Application. We also use Redis for temporary data storage and Postgres for persistent data storage. A hand-drawn diagram illustrates how these components interact.

https://app.eraser.io/workspace/F3SaRt1kqQlk82rm7yfj?origin=share

II. Component Descriptions

Authentication Server: Parses user access tokens and forwards quiz request details such as username, quiz ID, and quiz answer to the Quiz Server.

Quiz Server: Manages quiz operations including joining or quitting sessions, processing quiz answers, calculating scores, and communicating with the Leaderboard Service for real-time updates.

Leaderboard Service: this is setted as a mock in the source code. It processes updated user scores, ranks players using Redis Zset structures and ZRange, ZAdd etc to update data, and broadcasts leaderboard updates to all connected clients.

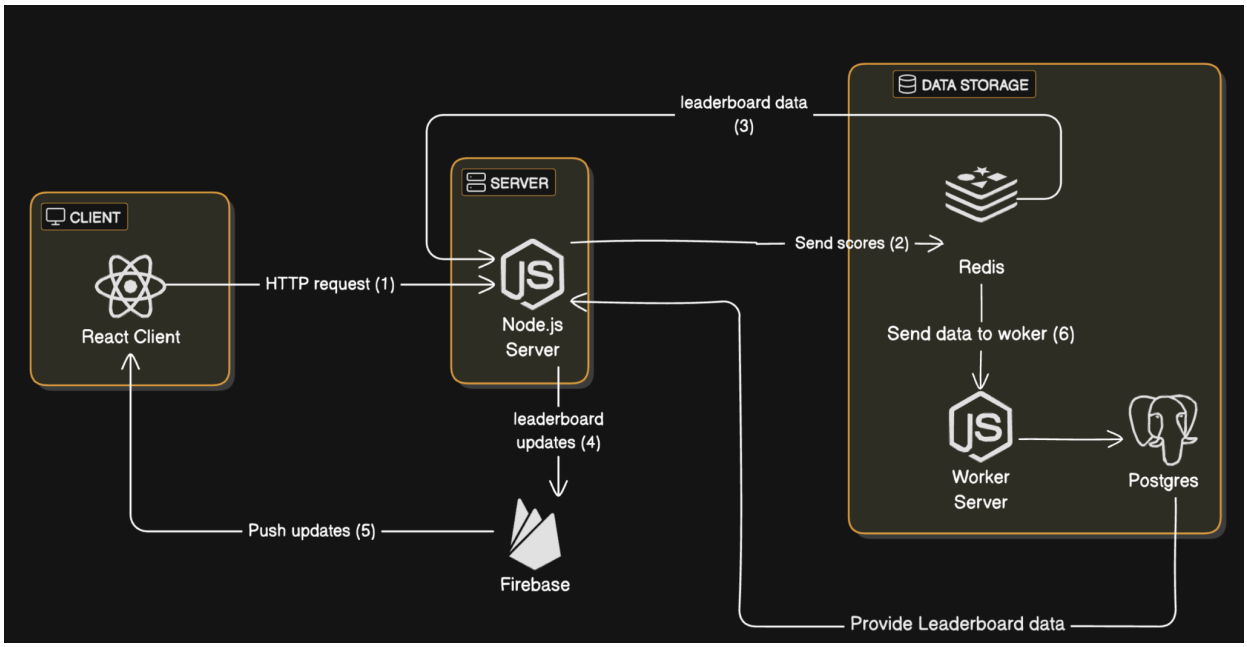
Notification Service: this is setted as a mock in the source code. It’s to send updates to players on leaderboard changes.

Redis: Provides fast, temporary storage and manages session data and real-time leaderboard calculations using Zsets.

Postgres: Stores long-term user data and historical leaderboard records. A background job transfers data from Redis to Postgres for permanent storage.

III. Data Flow

Here are explanations for numbers on the arrows on the diagram arrows:



1. **A player initiates a quiz session by sending an authentication request.**
2. **The Authentication Server validates the request and passes the necessary details to the Quiz Server.**
3. **The Quiz Server retrieves user and quiz data from Postgres and creates or joins a quiz session in Redis.**
4. **During the quiz, players submit answers, which are validated by the Quiz Server. Correct answers earn points that are updated in Redis.**
5. **Changes in points trigger a push notification that updates the leaderboard for all connected clients.**
6. **After the quiz ends, a scheduled job transfers leaderboard data from Redis to Postgres for historical storage.**

IV. Technology Justification

We use Fastify and NodeJS because their event-driven, non-blocking architecture is ideal for I/O-intensive applications like our Quiz App. Redis is chosen for its speed and capability to manage temporary data efficiently. Postgres is our choice for reliable, long-term data storage and its support for data partitioning, which is useful for handling large volumes of leaderboard records.

V Conclusion

Our Quiz Application is designed to deliver a seamless, scalable, and interactive user experience by integrating efficient authentication, real-time quiz management, and dynamic leaderboard updates. Thank you for reading this document. If you have any questions or need more details, please feel free to reach out.