FACULTY OF TECHNOLOGY
Information & Communication Technology
Subject: Capstone Project

Aryan Mahida – 92200133011 **Jay Mangukiya - 92200133040**

System Design and Architecture - Intermediate Review

Here, we will be discussing about entire system design flow of the project.

1. Modular design architecture

1.1 System architecture overview:

The QuizUP system follows a layered modular architecture with clear separation of concerns, enabling independent development, testing, and deployment of components. The architecture is structured into five primary layers:

- **Presentation Layer**: React Frontend, mobile application
- **Application Layer**: REST API server (expressJS, port 3000) and Match server (socket.io, port 3001)
- **Data Layer**: PostgreSQL (Port 5432)
- **Infrastructure Layer**: Docker containers, prometheus monitoring

1.2 Core modules:

1.2.1 Frontend module (React + Typescript):

- **Purpose**: User interface and experience management.
- **Structure**: Components, services (websocket clients), hooks, contexts, types, utils.
- **Features**: Quiz builder, real time gameplay, question bank, login/signup, quiz interface, leaderboard.
- **Benefits**: Component reusability due to react single page application, reduced duplication, easy role extension.
- **Command**: npm run dev -- --host

1.2.2 API/Backend server (express + sequelize):

- **Purpose**: Business logic, data management and easier data handling.
- **Endpoints**: Authentication, categories, quizzes, questions, matches, users, leaderboard etc.
- **Benefits**: Clear layering (Controller -> Service -> Model), service reuse, easy extensibility.
- **Command**: npm run dev

1.2.3 Match server (websocket + socket.io):

- **Purpose**: Real time match orchestration.
- **Core classes**: MatchEngine, PlayerManager, QuestionManager, ScoreManager, AIEngine.
- **Event flow**: authenticate -> create/join -> player_ready -> match_started -> submit_answer -> results.
- **Benefits**: Real time, supports multiple modes (friend, AI, tournament), easily extendable.
- **Command**: npm run dev:match

1.2.4 Database (PostgreSQL + sequelize)

The platform uses PostgreSQL with sequelize ORM to manage core entities like users, quizzes, questions, matches, and attempts. Sequelize provides object oriented concepts such as abstraction, reuse so it is easier to model the database. Performance is optimized through indexes on ELO ratings (for leaderboard scoring), match statuses, quiz question order, category hierarchies etc.

The entire module and file structure consists of files in the below format.

Module Structure

```
Frontend-admin/
├── src/
│   ├── components/                  # Reusable UI components
│   │   ├── ui/                      # shadcn/ui base components
│   │   ├── admin/                   # Admin dashboard components
│   │   ├── student/                 # Student interface components
│   │   └── shared/                  # Cross role shared components
│   ├── services/                    # API and websocket clients
│   ├── hooks/                       # React hooks
│   ├── contexts/                    #React contexts
│   ├── types/                       # Typescript definitions
│   └── utils/                       # Helper functions
Backend/src/
├── controllers/                     # HTTP request handlers
├── services/                        # Entire logic code
├── models/                          # Sequelize ORM models
├── routes/                          # API route definitions
├── middleware/                      # Auth, validation, logging etc.
├── utils/                           # Helper utilities
└── config/                          # Database & app configuration
```

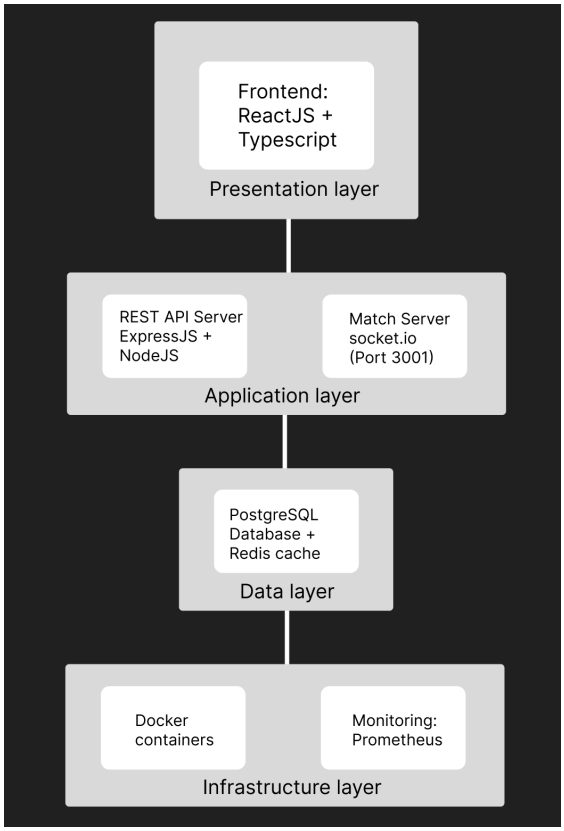
1.3 Module interaction:

The flow of the different modules will be like this

Data Flow: Frontend actions pass through the API to the database, while real time match data flows through websocket connections to the match server.

- **REST:** Frontend - API (with JWT auth)
- **Websocket:** Frontend - Match Server
- **ORM:** API - Database

Overall system architecture:



2. Technology stack specification

2.1 Frontend techstack:

2.1.1 Core Framework:

ReactJS + Typescript + Vite

- ReactJS supports component based architecture which is easier to manage and reuse. Also it supports SPA (single page application) where unnecessary rendering is reduced.
- Typescript is a type safe version of Javascript where the runtime errors are reduced and variables are less prone to any unwanted changes.

2.1.2 UI libraries and styling:

Shadcn + Aceternity UI + TailwindCSS:

- Shadcn and Aceternity UI contains number of predefined UI components which makes the implementation easier and reusable.
- TailwindCSS is a class based CSS framrwork where there are predefined classes which also makes easier to style the components. Also it supports faster styling and maintainability.

2.1.3 State management and data fetching:

React hooks + React context + axios API

- React hooks for managing component state, lifecycle and side effects as well.
- While the context is used for sharing state or data globally across components without prop drilling.
- While Axios simplifies making HTTP requests and handling responses in the application.

2.2 Backend techstack

2.2.1 Runtime and framework:

NodeJS + ExpressJS

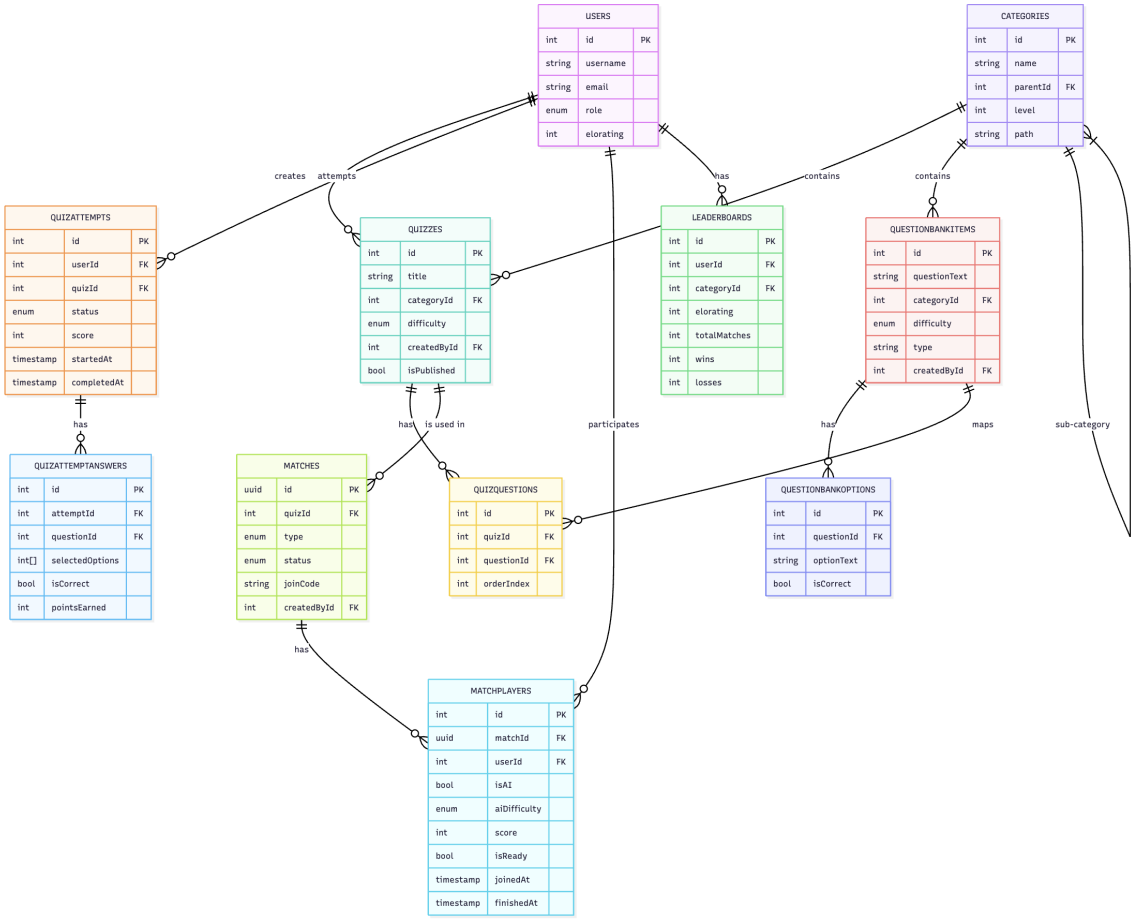
- NodeJS acts as a runtime environment to run the Javascript which is responsible to handle the backend operations
- ExpressJS is also a web framework of NodeJS which is used here to handle routes, middleware and server logic

2.2.2 Database and ORM:

PostgreSQL + Sequelize ORM

- PostgreSQL supports JSON, full text search and also applies the ACID principles while efficiently maintaining the indexes to search effectively.
- Sequelize ORM is used to map the database into classes and object which is easier to implement and manage using OOP principles.

The overall database schema should look like this:

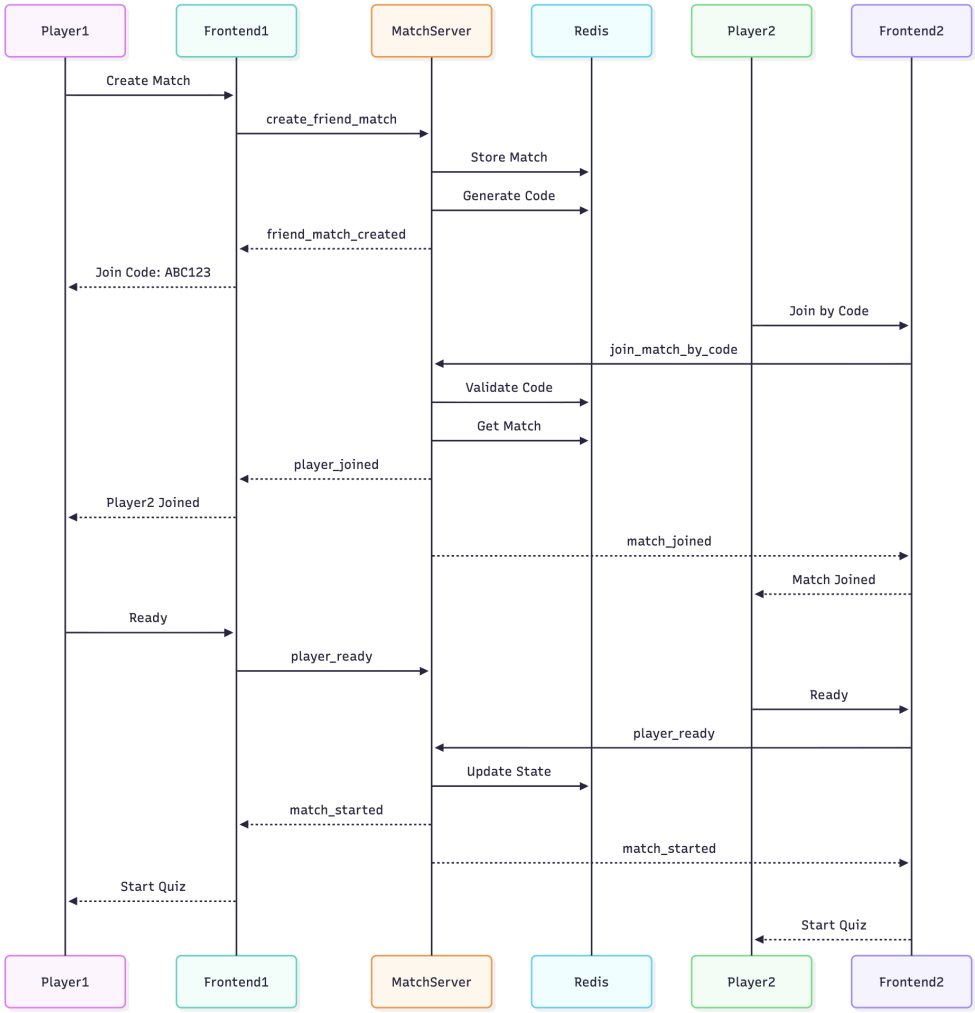


2.2.3 Real time quiz implementation:

Socket.io

- Socket.io websockets for bidirectional communication in real time to establish quiz environment.

The image displaying 1v1 match flow diagram, how the entire match between 2 players is handled.



A brief explanation about this:

- Player 1 starts the match by creating it, and the system generates a unique join code (ABC123).
- This code is shared with player 2, who uses it to join the same match. The backend (Matchserver) checks the code, validates it, and confirms player 2's entry.
- When everyone is marked ready, the Matchserver updates the match state and broadcasts a match started event.
- Finally, both players get the green light to start the quiz simultaneously which makes sure that there is synchronization.

2.3 Infrastructure and Devops:

2.3.1 Containerization:

Docker + Docker compose

- Consistent environments, multi stage builds (65% smaller images).
- Compose for local orchestration.
- Entire application runs without additional installation and requirements.

Services: Frontend (5173), Backend (3000), Matchserver (3001), Postgres, Redis.

2.3.2 Monitoring:

Prometheus + Grafana

Prometheus and Grafana collects and displays all the logs which are proven to be helpful during debugging and fixes. It collects app response time, errors, error rate etc.

3. Scalability Planning

As a part of these planning strategies, a few things can be implemented in future to increase the efficiency of the application.

3.1 Horizontal Scaling Strategy:

Application Layer Scaling:

I can use Nginx as a load balancer in future to distribute incoming traffic fairly across multiple backend instances.

Nginx used to serve static content, manage traffic, and load balance applications.

Database Scaling Strategy

Our database (PostgreSQL) is often the first bottleneck under heavy usage. To address this

- We can deploy read replicas to handle read-heavy traffic (like fetching quizzes, leaderboards, and results).
- Database writes always go to the primary to maintain consistency.