Design Document - Quizzie

Introduction

The design document specifies the system design - HLD/LLS for quiz application with Restful APIs - **Quizzie.**

The goal is to allow Learner to take quizzes, submit their answers and get result, and for Admins to be able to Create/Edit/Delete quizzes. Also Admins can user all features that a Learner can do.

Features

1. User Management - Authentication and Authorization.
2. Quiz Management - CRUD for quizzes.
3. Answer Management - Submissions of answers and feedback
4. Result Management for each user

High Level Design

Architecture

The system is considered to be developed with **monolith** architecture rather than a monolith.

a. Monolith architecture considers there is one service for the application. It consolidates all the components into a single codebase. So everything is in one code base. It **simplifies initial development, deployment and maintenance** which is ideal for small applications like Quizzie.
b. With all the functionalities in one place, it is **easier to test and debug** as compared to microservices architecture that spans across multiple services.
c. Monoliths **eliminates the network latency** as compared to microservices due to service to service communication. This ensures **better performance**.

Future Growth:

a. Considering scalability, if the application grows significantly in future, The current architecture is implemented such that **each module in the monolith can serve as a foundation for independent services**. This will help in scalability and flexibility taking advantage of microservices architecture.

Trade-offs considered:

a. Microservices offer flexibility to adopt different tech stacks for different services, but this is not a pressing requirement for Quizzie at this scale.

b. Microservices enables independent deployment, the applications current needs do not demand this.

The monolith architecture provides a **robust and manageable foundation** for Quizzie while **maintaining the flexibility to evolve into microservices** as the application scales and demands increase. This approach aligns with current requirements, minimizes overhead, and positions the system for efficient development and operation.

Services

a. **Auth Service**: User authentication and role-based authorization.
b. **Quiz Service**: Quiz Management with CRUD along with Questions and Answers.
c. **Result Service**: Calculates the score, stores answers, and generates user results.

Database Architecture
There are two options - Shared database among all the services or DB per service.

a. DB per service: Every service has its own database which it **manages independently**. This adds to l**oose coupling, data isolation and helps in independent scaling.** If in future we decide to have quizzes based on geographic location, this architecture will be advantageous. Each service has the opportunity to **choose DB based on their requirements**, but for now all of them will use SQL. This will also ensure that if one services DB schema has updated, it does not affect others. But here maintenance of multiple services and their DBs can cause overhead.
b. Shared Database: **Development and Maintenance will be easier** as all the data is stored in a single database,by avoiding the complexity of managing multiple databases.This approach makes **transactions easier** but introduces tight coupling, where changes to the schema can impact all services. While it can face scaling inefficiencies, it suits Quizzie's current size and requirements, with plans to adopt modular schema design and optimization strategies for future growth.

Conclusion is to use **Shared Database**.

Communication
All the services and clients will communicate with each other in a synchronous way using Restful APIs. This is simpler to implement, and as there is no requirement of bi-directional communication.
The API Gateway(Express gateway) in between client and services is to ensure proper routing of requests to appropriate service. Client just has to communicate with the API Gateway in a consistent format for request and response and the nitty gritty details of communication with each module is owned by the API Gateway.

Low Level Design

Services Design

1. Auth Service
   Responsibilities:
   a. Authenticate users: Handles the login/register process for different types of users - Admin and Learner.
   b. For Security, JWT is being used. After successful authentication a JWT will be issued to let users perform authorised operations in other services i.e. Admin to create a quiz and Learner to just take the quiz.
   c. For requests from other modules, it will verify the JWT to ensure the user is authenticated and authorized to perform the request. This is achieved using express middlewares.

   Restful API Endpoints:
   a. /auth/login
      i. METHOD - POST
      ii. Inputs - username, password
      iii. Output - JWT token
      iv. Status codes
         1. 200 if login is successful
         2. 400 if inputs are not provided
         3. 500 if internal server error

   b. /auth/register - for normal user
      i. METHOD - POST
      ii. Inputs - username, password
      iii. Output - Specifies if the user was registered or not
      iv. Status codes
         1. 201 if user is created and registered successfully
         2. 400 if inputs are not provided
         3. 500 if internal server error

   c. /auth/admin/register - for admin user
      i. METHOD - POST
      ii. Inputs - username, password
      iii. Output - Specifies if the user was registered or not
      iv. Status codes
         1. 201 if user is created and registered successfully
         2. 400 if inputs are not provided
         3. 500 if internal server error

2. Quiz Service
   a. /quiz - Creates a new quiz, after checking the token and authorizing logged in user
      i. METHOD - POST
      ii. Inputs - title, questions (ids of questions)
      iii. Output - Specifies if the quiz was created or not

iv. Status codes
    1. 201 if quiz was created
    2. 400 if inputs are not provided
    3. 401 if access is denied
    4. 500 if internal server error

b. /quiz - gets list of all the quizzes
    i. METHOD - GET
    ii. Inputs - NA
    iii. Output - list of quizzes
    iv. Status codes
        1. 200 if quiz list was obtained
        2. 401 if access is denied
        3. 500 if internal server error

c. /quiz/:id - gets quiz by id.
    i. METHOD - GET
    ii. Inputs - id of the quiz in params
    iii. Output - quiz details
    iv. Status codes
        1. 200 if quiz was obtained
        2. 401 if access is denied
        3. 400 if valid id is not provided
        4. 500 if internal server error

d. /question - Creates a new question, after checking the token and authorizing logged in user
    i. METHOD - POST
    ii. Inputs - text, options, correctOption
    iii. Output - Specifies if the question was created or not
    iv. Status codes
        1. 201 if question was created
        2. 400 if inputs are not provided
        3. 401 if access is denied
        4. 500 if internal server error

e. /question/ - gets list of all the questions
    i. METHOD - GET
    ii. Inputs - NA
    iii. Output - list of questions
    iv. Status codes
        1. 200 if question list was obtained
        2. 401 if access is denied
        3. 500 if internal server error

f. /question/:id - gets quiz by id.
    i. METHOD - GET
    ii. Inputs - id of the question in params
    iii. Output - question details

iv. Status codes
    1. 200 if question was obtained
    2. 401 if access is denied
    3. 400 if valid id is not provided
    4. 500 if internal server error

3. Result Service
    a. /result/submit-answers - submits answers for questions
        i. METHOD - POST
        ii. Inputs - quizId, answers - format of answers {questionId,selectedOption}
        iii. Output - specifies if the answers were submitted successfully
        iv. Status codes
            1. 200 if answers were submitted successfully
            2. 401 if access is denied
            3. 500 if internal server error

    b. /result/quizId/:id - gets result of a quiz by quizId.
        i. METHOD - GET
        ii. Inputs - id of the quiz in params
        iii. Output - score and actual answer summary
        iv. Status codes
            1. 200 if result was obtained
            2. 401 if access is denied
            3. 400 if valid id is not provided
            4. 500 if internal server error

Technology:

a. **Node.js + Express** for server
b. **ReactJs** for client
c. **JWT** for token-based authentication.
d. Brcypt for encryption/decryption of password
e. Database: **SQLite** (for user management, roles, quiz management and result management).

API Gateway:

a. Route requests to appropriate modules - Handled by Express Router
b. Handle authentication via JWT tokens.
c. Perform logging (console), and request validation.

Database Schema

Auth Service Database Schema (SQLite):

Users Table:

```
CREATE TABLE users (id INTEGER PRIMARY KEY AUTOINCREMENT,
  username TEXT NOT NULL UNIQUE,
  password TEXT NOT NULL,
  role_id INTEGER
  );
```

Roles Table:

```
CREATE TABLE roles (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  title VARCHAR(50) UNIQUE NOT NULL
);
```

Permissions Table:

```
CREATE TABLE permissions (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  title VARCHAR(50) UNIQUE NOT NULL
);
```

Roles Table:

```
CREATE TABLE role_permissions (
  role_id INTEGER,
  permission_id INTEGER,
  FOREIGN KEY (role_id) REFERENCES roles(id),
  FOREIGN KEY (permission_id) REFERENCES permissions(id),
  PRIMARY KEY (role_id, permission_id)
);
```

Quiz Service Database Schema (SQLite):

Quizzes Table:

```
CREATE TABLE quizzes (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  title TEXT NOT NULL,
  questions TEXT
);
```

Questions Table:

```
CREATE TABLE questions (
```

```
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    text TEXT NOT NULL,
    options TEXT NOT NULL,
    correct_option INTEGER NOT NULL
  );
```

Result Service Database Schema (SQLite):

Answers Table:

```
CREATE TABLE answers (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    question_id INTEGER,
    selected_option INTEGER,
    is_correct BOOLEAN,
    FOREIGN KEY (question_id) REFERENCES questions(id)
  );
```

Results Table:

```
CREATE TABLE results (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  user_id INTEGER,
  quiz_id INTEGER,
  score INTEGER,
  answers TEXT,
  FOREIGN KEY (user_id) REFERENCES users(id),
  FOREIGN KEY (quiz_id) REFERENCES quizzes(id)
);
```

Error Handling & Validation

HTTP Status Codes:

a. `200 OK`: Success
b. `201 Created`: Resource created successfully
c. `400 Bad Request`: Invalid input
d. `401 Unauthorized`: Invalid or missing JWT token
e. `404 Not Found`: Resource not found
f. `500 Internal Server Error`: Server error

Input Validation:

a. Validate all incoming data at the API level
b. Ensure all required fields (like question text and options) are present and correct.

Security

- **Authentication**:
    - JWT-based authentication is used for securing API endpoints.
    - Tokens will be sent as Authorization headers.
- **Role-Based Access Control (RBAC)**:
    - Admins can create quiz/question and take quizzes; learners can only view, take and submit quizzes.