Quotis System Design Document

Jisung Han, Zuhair Khan, Danish Mohammed, Melad Ebadi, Mustafa Timbawala

Contents

1 Introduction

Quotis is a mobile application designed to connect clients with reliable handymen for various trades work, including plumbing, electrical work, and moving services. Our platform aims to provide a trustworthy and efficient way for users to find qualified professionals and for service providers to find relevant jobs.

The frontend of the application is built using React Native and TypeScript, providing a consistent and responsive interface across both iOS and Android platforms. The backend is powered by Node.js and Express, also utilizing TypeScript for type safety and maintainability.

For data storage and retrieval, MongoDB is employed as the primary database, ensuring scalability and flexibility in handling diverse data types. AWS S3 is integrated for secure and efficient image storage, allowing users to upload and access images reliably.

Key frameworks and libraries used in the Quotis project include:

- React Native: For building cross-platform mobile applications.
- Node.js: As the runtime environment for the backend server.
- Express: A web application framework for managing routes and middleware.
- TypeScript: For type safety and improved code quality in both frontend and backend.
- MongoDB: For database management, offering a scalable NoSQL solution.
- AWS SDK: For interacting with AWS services, particularly S3 for image storage.
- multer: A middleware for handling multipart/form-data, primarily used for file uploads.

2 CRC Cards

2.1 Frontend Classes

Class Name: Login

Class name: Login	
Parent class: React.Component **Sub class: N/A**	
Responsibilities:	**Collaborators:**
- Render the view for the login page - Manage state variables for username, password, and login error message. - Handle changes in the username and password fields - Handle form submission for login - Make a POST request to the server to authenticate the user. - Set the login error message if authentication fails - Redirect the user to the UserDashboard or ProviderDashboard upon successful login.	- React (useState) - react-navigation (NavigationContainer, createStackNavigator) - axios (for making HTTP requests)

Class Name: Register

Class name: Register	
Parent class: React.Component **Sub class: N/A**	
Responsibilities:	**Collaborators:**
- Render the view for the registration page - Manage state variables for email, password, and role - Handle changes in the input fields - Handle form submission for registration - Make a POST request to the server to create a new user - Redirect the user to the UserDashboard or ProviderDashboard upon successful registration	- React (useState) - react-navigation (NavigationContainer, createStackNavigator) - axios (for making HTTP requests) - Picker (for selecting user role)

Class Name: CreatePost

Class name: CreatePost	
Parent class: React.Component **Sub class: N/A**	
Responsibilities:	**Collaborators:**
- Render the view for creating a post - Manage state variables for title, description, image - Handle changes in input fields - Handle form submission for creating a post - Make a POST request to the server to create a new post - Upload images to AWS S3	- React (useState) - axios (for making HTTP requests) - expo-image-picker (for image selection) - FormData (for handling form submissions)

Class Name: UserDashboard

Class name: UserDashboard

Parent class: React.Component **Sub class: N/A**	
Responsibilities:	**Collaborators:**
- Render the view for the user dashboard - Fetch and display user details and posts - Handle navigation to CreatePost component - Can modify the posts.	- React (useState, useEffect) - axios (for making HTTP requests) - react-navigation (NavigationContainer, createStackNavigator) - FlatList (for rendering lists of posts) - ProviderDashboard

Class Name: ProviderDashboard

Class name: ProviderDashboard	
Parent class: React.Component **Sub class: N/A**	
Responsibilities:	**Collaborators:**
- Render the view for the user dashboard - Fetch and display user details and posts (for all users. Based on their professional category.) - Handle navigation to CreatePost component - Can choose the posts and accept the request.	- React (useState, useEffect) - axios (for making HTTP requests) - react-navigation (NavigationContainer, createStackNavigator) - FlatList (for rendering lists of posts)

Class Name: Profile

Class name: Profile	
Parent class: React.Component **Sub class: N/A**	
Responsibilities:	**Collaborators:**
- Render the view for the user profile. - Fetch and display user details. - Allow users to change their own data.	- React (useState, useEffect) - axios (for making HTTP requests) - react-navigation (NavigationContainer, createStackNavigator)

2.2 Backend Classes

Class Name: User Schema

Class name: User Schema	
Parent class: N/A **Sub class: N/A**	
Responsibilities:	**Collaborators:**
- Define the schema for user data - Using for: - Handle user authentication - Handle user registration - Retrieve user details	- MongoDB (Database) - bcrypt (Password hashing) - server

Class Name: Post Schema

Class name: Post Schema	
Parent class: N/A **Sub class: N/A**	
Responsibilities:	**Collaborators:**

- Define the schema for post data - Using for: - Create new posts - Retrieve posts - Delete posts	- User (To link posts to users) - MongoDB (Database) - AWS S3 (Image Storage) - server

Class Name: ImageService

Class name: ImageService	
Parent class: server **Sub class: N/A**	
Responsibilities:	**Collaborators:**
- Handle image uploads to AWS S3 - Generate signed URLs for image access - Delete images from AWS S3	- AWS S3 (Image Storage) - multer (Middleware for handling multipart/form-data)

Class Name: server

Class name: server	
Parent class: N/A **Sub class: AuthController, PostController, S3Service, NotificationService, TextingService**	
Responsibilities:	**Collaborators:**
- Initialize and configure the Express server - Connect to MongoDB - Set up middleware (e.g., bodyParser, cors) - Define routes and use controllers for handling requests - Start the server and listen on a specified port	- Express - MongoDB - AuthController - PostController - S3Service - NotificationService - TextingService

Class Name: TextingService

Class name: TextingService	
Parent class: server **Sub class: N/A**	
Responsibilities:	**Collaborators:**
- Handle sending text messages to users - Manage texting preferences - Integrate with third-party texting services (e.g., Twilio)	- User (Model) - Express (Framework) - Twilio (or other texting services)

Class Name: AuthController

Class name: AuthController	
Parent class: server **Sub class: N/A**	
Responsibilities:	**Collaborators:**
- Handle user login - Handle user registration - Validate user credentials	- User (Model) - Express (Framework) - bcrypt (Password hashing)

Class Name: NotificationService

Class name: NotificationService	
Parent class: server **Sub class: N/A**	
Responsibilities:	**Collaborators:**
- Handle sending notifications to users - Manage notification preferences	- User (Model) - Express (Framework) - bcrypt (Password hashing)

Class Name: PostController

Class name: PostController	
Parent class: server **Sub class: N/A**	
Responsibilities:	**Collaborators:**
- Handle creating new posts - Handle retrieving posts - Handle deleting posts - Integrate with S3Service for image handling	- Post (Model) - S3Service (Service) - Express (Framework)

3 System Interaction with Environment

4 Dependencies

- "@expo/metro-runtime": " 3.2.1"
- "@react-native-picker/picker": "^{2}.7.7"
– "$@react-navigation/native$" : "^{6}.1.17"
– "$@react-navigation/stack$" : "^{6}.3.29"
– "$axios$" : "^{1}.7.2"
– "$expo$" : "^{5}1.0.11"
– "$expo-image-picker$" : " 15.0.5"
– "$expo-status-bar$" : " 1.12.1"
– "$react$" : "^{1}8.3.1"
– "$react-dom$" : "^{1}8.3.1"
– "$react-native$" : "0.74.2"
– "$react-native-gesture-handler$" : "^{2}.16.2"
– "$react-native-reanimated$" : "^{3}.12.0"
– "$react-native-safe-area-context$" : "^{4}.10.4"
– "$react-native-screens$" : "^{3}.31.1"
– "$react-native-web$" : " 0.19.10"
– "$react-native-webview$" : "13.8.6"

5 Developer Dependencies

- "@babel/core": "^{7}.20.0"
– "$@types/react$" : " 18.2.79"
– "$typescript$" : " 5.3.3"

5.1 Other details

- Operating System: Any
- Programming Languages: TypeScript, JavaScript (JSX)
- Database: MongoDB (Hosted on MongoDB Atlas) and AWS S3 (for photo storage)

- Virtual Machine: not needed
- Compiler: tsc (typescript compiler)
- Network Configuration:
  - Backend Server: Runs on port 3000
  - Frontend Server: Runs on default Expo port 8081

6 Architecture Description

Our system architecture follows a monolithic MVC (Model-View-Controller) pattern using the MERN (MongoDB, Express, React Native, Node.js) stack. The architecture is organized into three main components:

6.1 Model

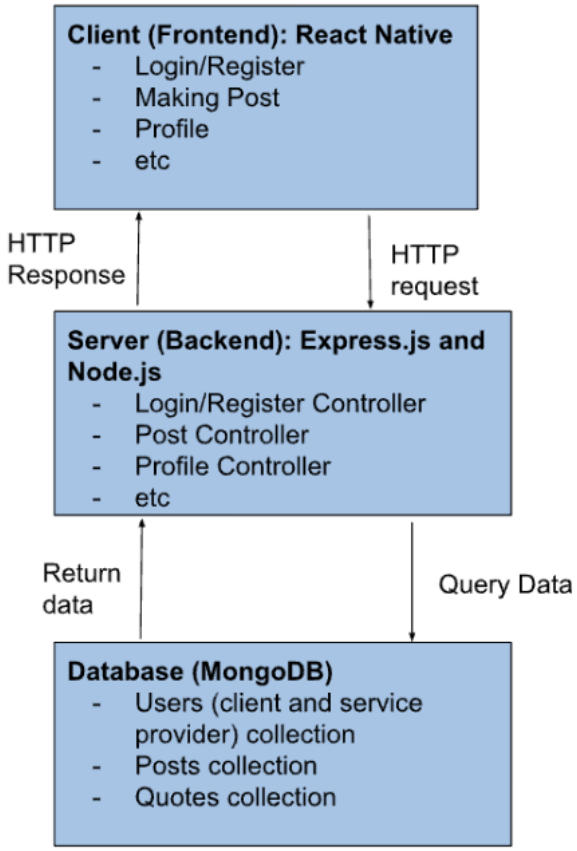
The model component is responsible for the data layer of the application. It interacts with MongoDB to perform CRUD (Create, Read, Update, Delete) operations. This component includes schemas and models for various entities like users, posts, and other relevant data structures.

6.2 View

The view component is built using React Native, providing the user interface for the application. It handles the presentation logic and user interactions. This component includes screens and components for different views such as login, registration, user dashboard, and provider dashboard.

6.3 Controller

The controller component acts as an intermediary between the model and the view. It is built using Express.js and Node.js and handles the business logic and application flow. Controllers manage API endpoints, process requests, interact with the model, and return responses to the view for appropriate output.



7 System Decomposition

Our system architecture is divided into three main components: the database, the front end, and the back end. The front end, developed with React Native, provides the user interface and handles user interactions. We utilize consistent styling across the application using reusable stylesheets. The front end communicates with the back end via API calls, allowing for the execution of CRUD operations like adding a new user, reading username and password to compare, updating profile info, deleting a quote or post, etc.

The back end is built with Node.js and Express.js, leveraging Express to manage HTTP request handling at various routes. These routes interact with our MongoDB database to perform necessary operations such as creating, reading, updating, and deleting records. Our database is structured with distinct collections for user profiles, posts, and other related data.

The following is our strategy for dealing with errors (both accidental and anticipated) as well as exceptional cases:

- Invalid User Input: Inputs are validated at both the front end and back end. The front end uses form validation to prevent incorrect data entry, while the back end performs additional checks before processing requests. If invalid data is detected, the system provides clear error messages to guide users in correcting their input.
- Network Failures: The front end includes retry mechanisms for failed API requests due to network issues. Users are informed of connectivity problems and prompted to retry actions once the network is available.
- External System Failures: The back end is designed to handle failures in external services such as the database. If a database operation fails, the system logs the error, returns a user-friendly message, and retries the operation if applicable. For persistent issues, users are informed of the problem and advised to try again later.

- General Error Handling: All unexpected errors are logged with detailed information for debugging purposes. Users receive generic error messages to maintain security and usability.