Frontend Components (`setup/frontend/src`)

1. App (Routing Shell)

Class Name: App (Routing Shell)	
Parent Class: React functional component	
Responsibilities: • Instantiate `BrowserRouter` and define primary routes (`/home`, `/signup`, `/login`, `/content`, `/`). • Render `Navbar` persistently across pages. • Wrap protected paths with `ProtectedRoute` to enforce authentication guards.	Collaborators: • `react-router-dom` (`Link`, `useNavigate`) • Browser `localStorage` • `App.js` (host render)

2. Navbar

Class Name: Navbar	
Parent Class: React functional component	
Responsibilities: • Render top navigation links and branding. • Read JWT token from `localStorage` to toggle auth actions. • Handle logout by clearing stored auth data and redirecting via `useNavigate`.	Collaborators: • `react-router-dom` (`Link`, `useNavigate`) • Browser `localStorage` • `App.js` (host render)

3. ProtectedRoute

Class Name: ProtectedRoute	
Parent Class: React functional component	
Responsibilities: • Guard wrapped children by checking for a JWT token in `localStorage`. • Redirect unauthenticated users to `/login` using `<Navigate />`.	Collaborators: • Browser `localStorage` • `react-router-dom` (`Navigate`) • `App.js` (route wrapper)

4. Home Page

Class Name: Home

Parent Class: React functional component	
Responsibilities: • Present CommuteBuddy value proposition. • Provide CTA buttons leading to signup or login flows via programmatic navigation.	Collaborators: • `useNavigate` from `react-router-dom` • `App.js` routing

5. Signup Page

Class Name: Signup	
Parent Class: React functional component	
Responsibilities: • Manage controlled form state for `name`, `email`, and `password`. • Submit registration requests to `POST /api/users/register` using Axios. • Display feedback (alerts) on success or server-side errors.	Collaborators: • Axios HTTP client • Express backend (`userRoutes` → `registerUser`) • Browser alert/UI

6. Login Page

Class Name: Login	
Parent Class: React functional component	
Responsibilities: • Capture email/password and send `POST /api/users/login`. • Persist JWT and user profile to `localStorage` upon success. • Redirect to `/home` and surface validation errors.	Collaborators: • Axios HTTP client • Express backend (`userRoutes` → `loginUser`) • Browser `localStorage`, `useNavigate`

7. Content Page (Placeholder)

Class Name: Content	
Parent Class: React model (currently empty)	
Responsibilities: • (Planned) Fetch and display protected commute content after authentication.	Collaborators: • `ProtectedRoute` for access control • Axios (future use)

• (Planned) Consume `/api/content` responses tailored to logged-in users.	• Backend content APIs

Backend Components (`setup/backend`)

8. Express Server

Class Name: Express Server (`server.js`)	
Parent Class: Node.js module exporting an Express instance	
Responsibilities: • Load environment variables with `dotenv`. • Configure middleware (`cors`, `express.json`). • Connect to MongoDB using Mongoose. • Mount `/api/users` and `/api/content` routers and start listening on `PORT`.	Collaborators: • `routes/userRoutes.js`, `routes/contentRoutes.js` • `mongoose` connection • `.env` configuration

9. User Routes

Class Name: User Routes (`routes/userRoutes.js`)	
Parent Class: Express Router	
Responsibilities: • Define `/register`, `/login`, `/content` endpoints. • Apply `protect` middleware to guard `/content`.	Collaborators: • `controllers/userController.js` • `middleware/authMiddleware.js` • Express app (`server.js`)

10. Content Routes

Class Name: Content Routes (`routes/contentRoutes.js`)	
Parent Class: Express Router	
Responsibilities: • Expose `/api/content` endpoint for protected commute data. • Enforce authentication via `protect` middleware.	Collaborators: •`controllers/contentController.js` • `middleware/authMiddleware.js` • Express app (`server.js`)

11. User Controller

Class Name: User Controller (`controllers/userController.js`)

Parent Class: Node.js module (collection of handlers)	
Responsibilities: • `registerUser`: validate uniqueness, hash passwords, persist user, return JWT. • `loginUser`: verify credentials, issue JWT, and respond with user profile. • `getContent`: return protected content for authenticated users.	Collaborators: • `models/User.js` • `bcryptjs`, `jsonwebtoken` • `utils/generateToken.js` • Express routers

12. Content Controller

Class Name: Content Controller (`controllers/contentController.js`)	
Parent Class: Node.js module	
Responsibilities: • Return placeholder content referencing `req.user.name`. • Serve as template for future commute content logic.	Collaborators: • `middleware/authMiddleware.js` • `routes/contentRoutes.js` • Authenticated request context (`req.user`)

13. Auth Middleware

Class Name: Auth Middleware (`middleware/authMiddleware.js`)	
Parent Class: Express middleware function	
Responsibilities: • Parse bearer token from `Authorization` header. • Verify JWT signatures using `JWT_SECRET`. • Attach authenticated user to `req.user`. • Reject missing/invalid tokens with `401`.	Collaborators: • `jsonwebtoken` • `models/User.js` • Express routers consuming `protect`

14. User Model

Class Name: User Model (`models/User.js`)	
Parent Class: Mongoose model	
Responsibilities: • Define schema with required `name`, unique `email`, hashed `password` fields. • Provide persistence helpers used by controllers and middleware.	Collaborators: • `controllers/userController.js` • `middleware/authMiddleware.js` • Mongoose connection (`server.js`)

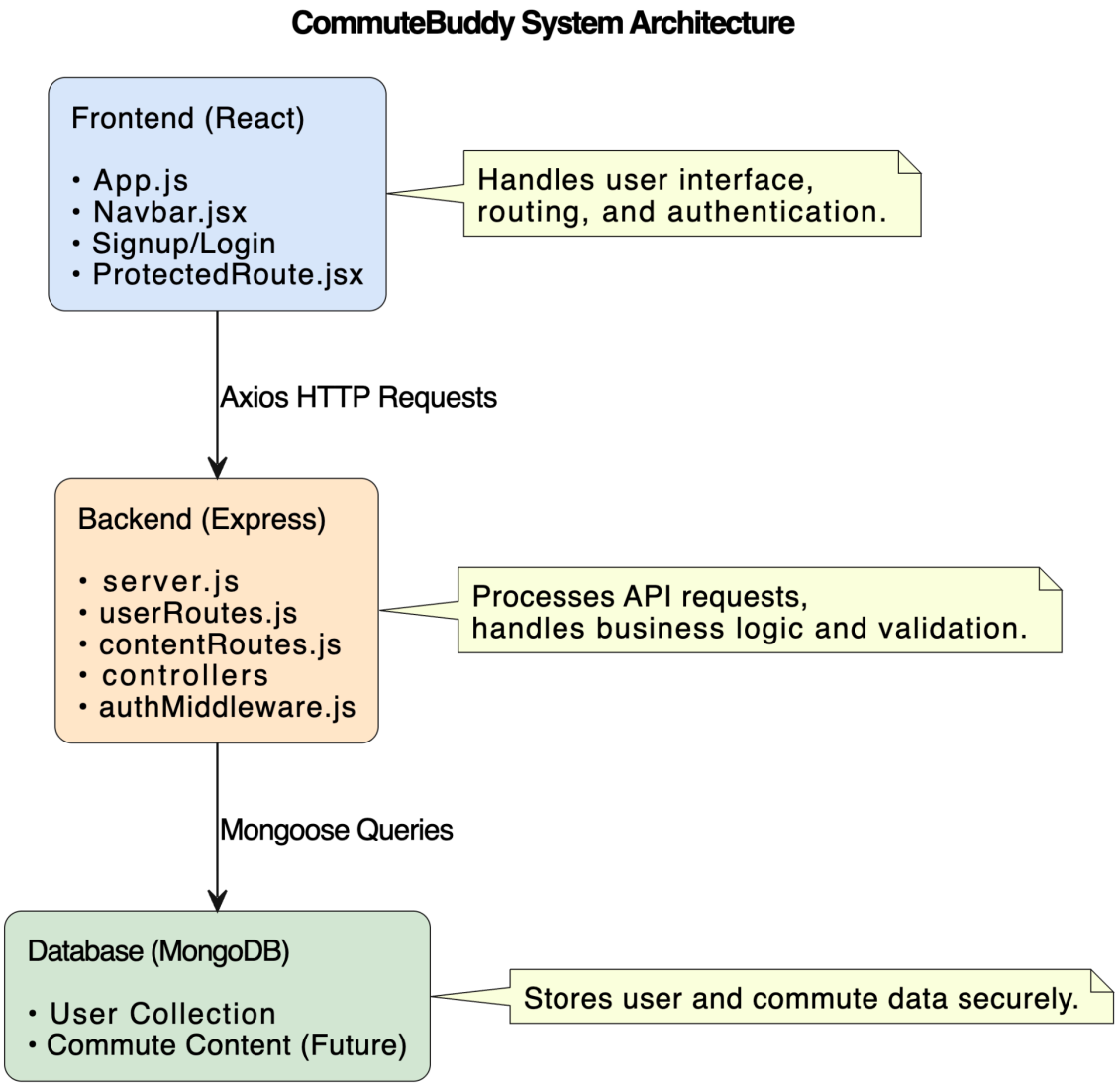
15. JWT Token Utility

Class Name: JWT Utility (`utils/generateToken.js`)	
Parent Class: Node.js helper function	
Responsibilities: • Generate signed JWT tokens with 24-hour expiry for a given user ID. • Centralize token creation to keep controllers concise.	Collaborators: • `jsonwebtoken` • `controllers/userController.js` • Environment `JWT_SECRET`

16. Environment Config

Class Name: Environment Config (`.env` / `.env.example`)	
Parent Class: Configuration file	
Responsibilities: Supply runtime values for `PORT`, `MONGO_URI`, and `JWT_SECRET`. • Enable consistent setup across local environments.	Collaborators: • `server.js` via `dotenv` • MongoDB deployment • JWT signing (`authMiddleware`, `generateToken`)

System Architecture Design



System Decomposition

The **CommuteBuddy system** follows a **three-tier client–server architecture** that clearly separates the presentation, application, and data management layers to ensure modularity, scalability, and maintainability. At the highest level, the **frontend** is implemented using React and serves as the presentation layer responsible for user interaction, navigation, and authentication workflows. It communicates with the **backend** through RESTful APIs using Axios. The **backend**, built with Node.js and Express, acts as the core application layer for processing requests, handling business logic such as user registration and login, validating input, and managing authentication through middleware. The backend connects to the **MongoDB database**, which serves as the data layer responsible for storing user credentials, session tokens, and commute-related content. This layered structure allows the system to remain loosely coupled, with each component interacting only through well-defined interfaces.

Error and exception handling are managed systematically across layers. On the frontend, invalid or missing user inputs trigger validation messages and alert prompts before data is sent to the server. In the backend, middleware ensures robustness by intercepting unauthorized requests, responding with a `401 Unauthorized` error when JWT tokens are invalid or expired, and returning descriptive `400 Bad Request` or `500 Internal Server Error` responses when input validation or server-side failures occur. Additionally, `403 Forbidden` when unauthorized users attempt to access protected or admin pages by redirecting them to a designated error or login page. In case of database or network failures, the system gracefully handles exceptions by logging errors on the server and showing friendly feedback messages on the client. Overall, this architecture provides a clean separation of concerns, predictable communication flow, and resilience against common operational issues.