Markd - Technical Design Document

Table of Contents

Introduction and System Overview

Introduction

Markd is a modern blog-sharing platform designed to enable users to create, share, and discover articles on various topics. The platform offers a seamless user experience with features like real-time updates, user authentication, article management, and social interactions such as upvoting.

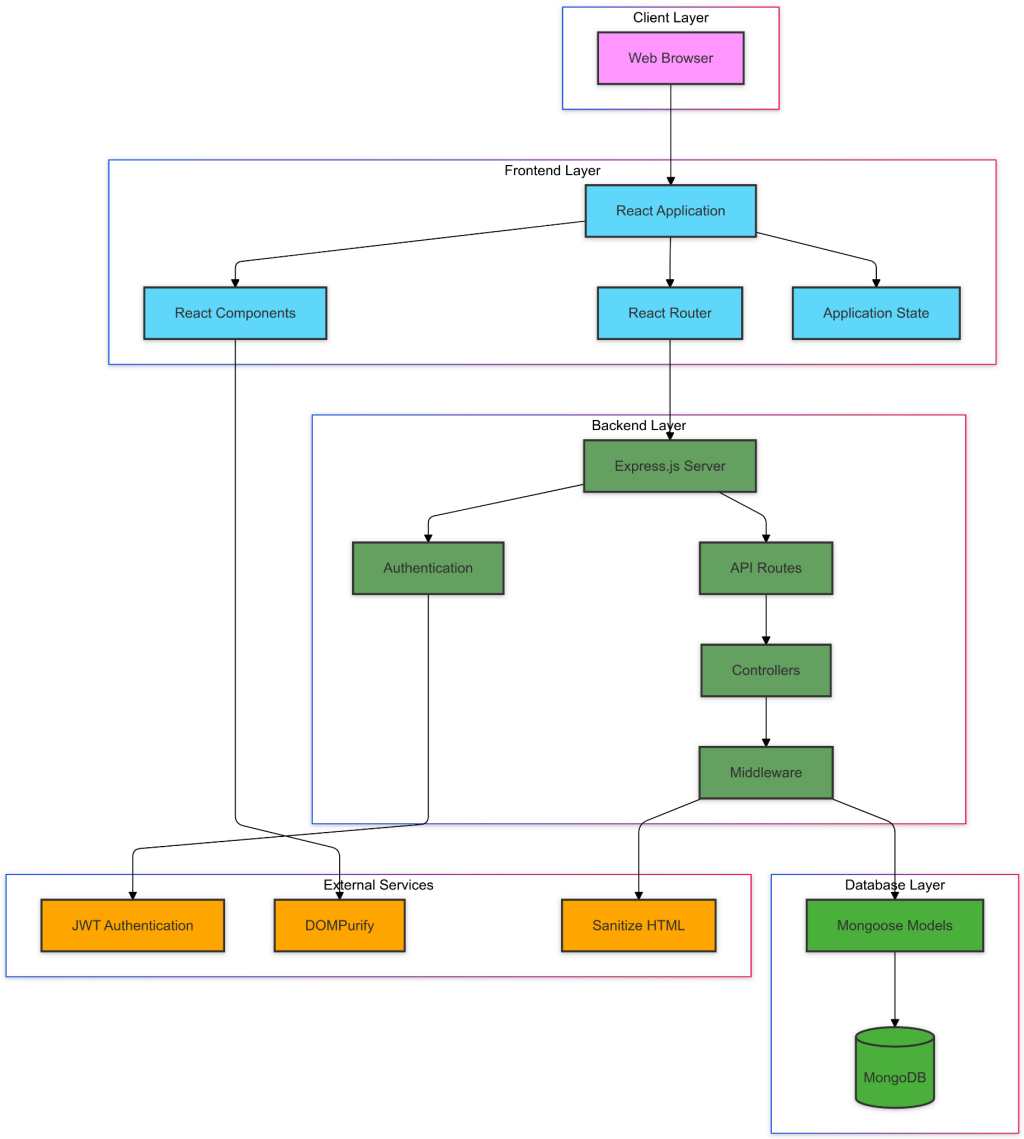
This design document provides a comprehensive overview of the project's architecture, components, technology stack, and design decisions. It aims to serve as a guide for developers, stakeholders, and contributors.

System Overview

Markd is built using the MERN stack (MongoDB, Express.js, React, Node.js), utilizing modern web development practices. The application is structured to provide scalability, maintainability, and a responsive user experience across devices.

Architecture

High-Level Architecture



The system follows a three-tier architecture, comprising:

1. **Frontend**: Developed with React.js, responsible for the client-side user interface and interactions.
2. **Backend API**: Built with Express.js and Node.js, handling server-side logic, API endpoints, authentication, and business logic.
3. **Database**: Utilizes MongoDB for storing user data, articles, and related information.

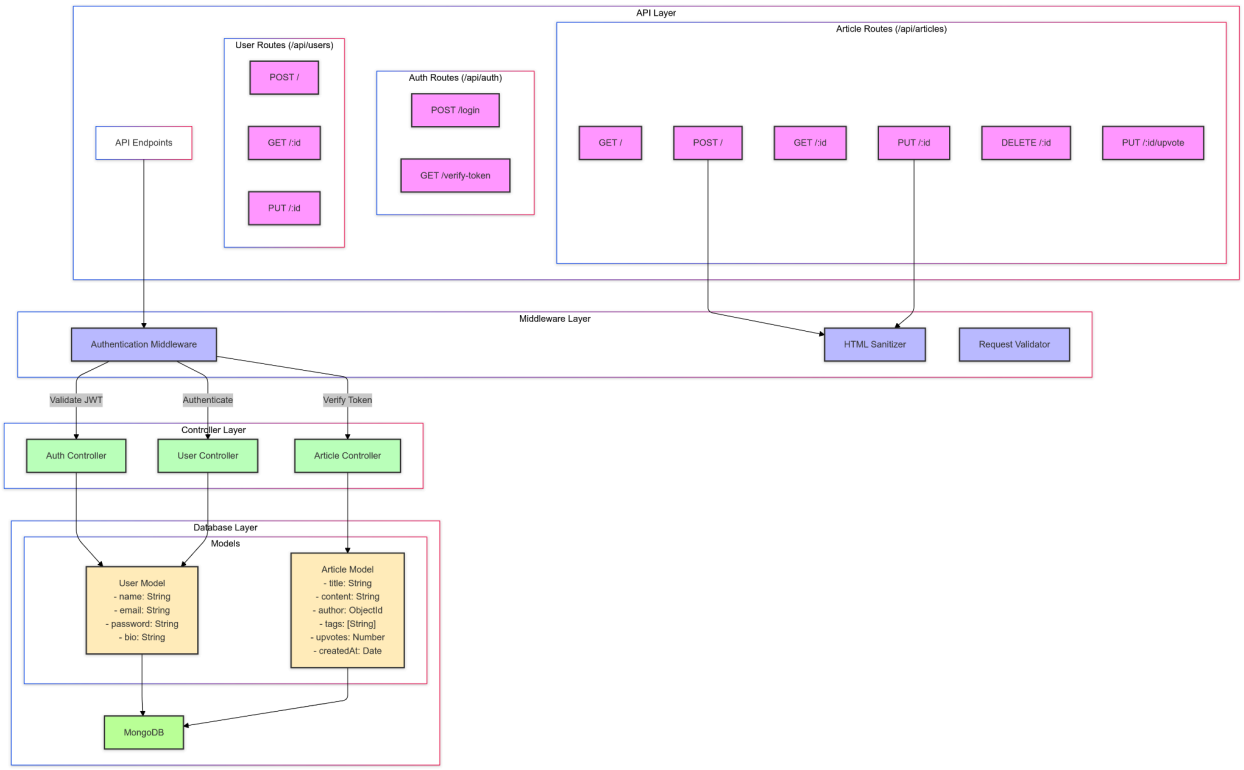
Backend Design

Technologies Used

- **Node.js**: JavaScript runtime environment.
- **Express.js**: Web application framework for building APIs.
- **MongoDB**: NoSQL database for data storage.
- **Mongoose**: ODM (Object Data Modeling) library for MongoDB.
- **JWT**: JSON Web Tokens for authentication.
- **bcrypt**: Library for hashing passwords.

Project Structure

- **index.js**: Entry point of the server application.
- **middleware/**: Contains middleware functions, including authentication.
- **models/**: Mongoose schemas for User and Article models.
- **routes/**: Defines API endpoints for authentication, users, and articles.
- **generateArticles.js**: Script for generating sample articles (used for testing or demonstration).



Backend Design - API, Database Schema and Middlewares

API Design

The backend exposes RESTful API endpoints categorized under:

- **Authentication (`/api/auth`)**
    - `POST /login`: User login and JWT token issuance.
- **Users (`/api/users`)**
    - `POST /`: User registration.
    - `GET /:id`: Retrieve user profile and their articles.
    - `PUT /:id`: Update user profile.
- **Articles (`/api/articles`)**
    - `GET /`: Fetch a list of articles with pagination and sorting options.
    - `POST /`: Create a new article.
    - `GET /:id`: Retrieve a specific article.
    - `PUT /:id`: Update an existing article.
    - `DELETE /:id`: Delete an article.
    - `PUT /:id/upvote`: Upvote an article.

Database Schema

User Model (User.js)

Fields:

- **name** (String, required): User's full name.
- **email** (String, required, unique): User's email address.
- **password** (String, required): Hashed password.
- **bio** (String, optional, default: ""): Short biography.

Indexes:

- Unique index on **email** for ensuring unique user emails.

Relations:

- A user can author multiple articles.

Notes:

- Passwords are hashed using bcrypt before being saved, as defined in userSchema.pre("save").

Article Model (Article.js)

Fields:

- **title** (String, required): Title of the article.
- **content** (String, required): Main content/body of the article.
- **author** (ObjectId, required): References the **User** model.
- **tags** (Array of Strings): Keywords associated with the article.
- **upvotes** (Number, default: 0): Count of upvotes received.
- **createdAt** (Date, default: Date.now): Timestamp of creation.

Indexes:

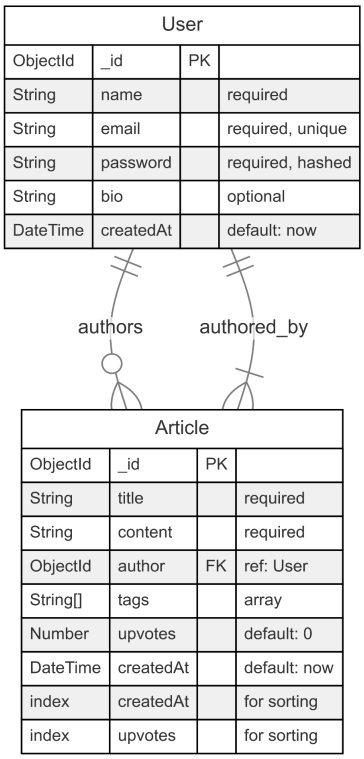
- Index on **createdAt** for chronological sorting.
- Index on **upvotes** for sorting popular articles.

Relations:

- Each article is associated with an author (user).

Notes:

- The **author** field establishes a relationship between articles and users.
- Articles can have multiple tags for better categorization.



Middlewares

Authentication Middleware (auth.js)

- Validates JWT tokens sent in the `Authorization` header.
- Attaches the authenticated user's information to the request object.
- Protects routes that require authentication.

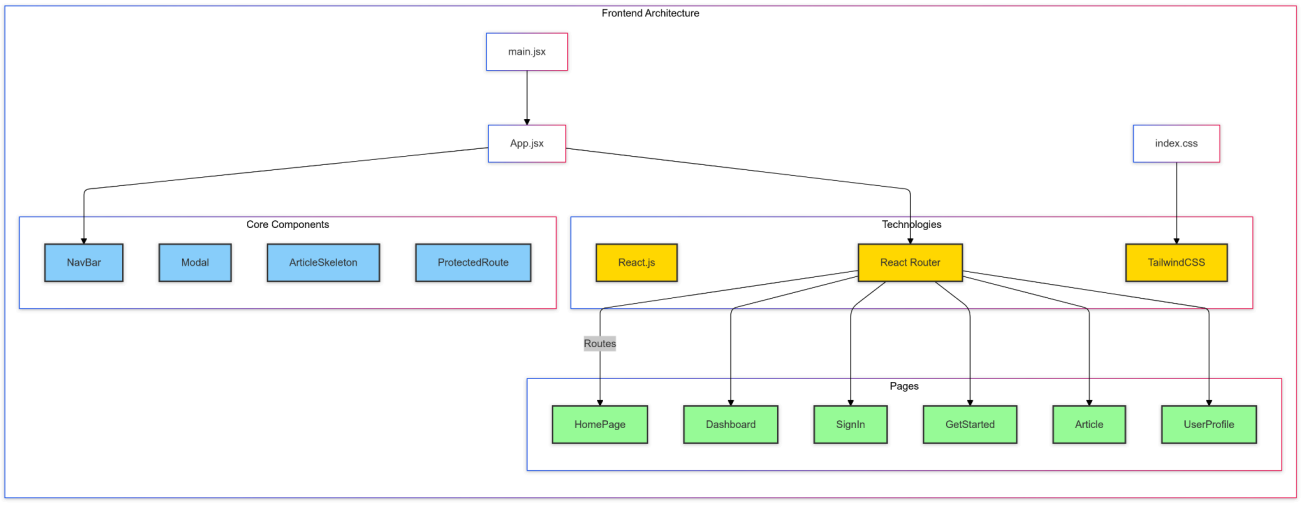
Frontend Design

Technologies Used

- **React.js**: JavaScript library for building user interfaces.
- **React Router DOM**: Handling client-side routing.
- **Tailwind CSS**: Utility-first CSS framework for styling.
- **Vite**: Build tool for faster development.
- **ESLint**: Linting utility to maintain code quality.

Project Structure

- **main.jsx**: Entry point of the React application.
- **App.jsx**: Main application component.
- **components/**: Reusable UI components.
- **pages/**: Page components corresponding to routes.
- **routes/**: Defines client-side routing.
- **lib/**: Utility functions.
- **index.css**: Global CSS and Tailwind directives.
- **public/**: Static assets.



Frontend Design - Routing, State Management and Key Components

---

Routing

Implemented using React Router:

- `/`: Home page.
- `/getstarted`: User registration page.
- `/signin`: User login page.
- `/dashboard`: User's personalized feed.
- `/articles/:id`: View a specific article.
- `/new-article`: Create a new article.
- `/articles/:id/edit`: Edit an existing article.
- `/users/:id`: View a user's profile.

State Management

- **Local State**: Managed using `useState` and `useEffect` hooks.
- **Authentication State**:
    - Stored in `localStorage`.
    - Accessed via custom hooks or utility functions.
- **Data Fetching**:
    - Utilizes `fetch` API.
    - Handles loading and error states.

Key Components

NavBar

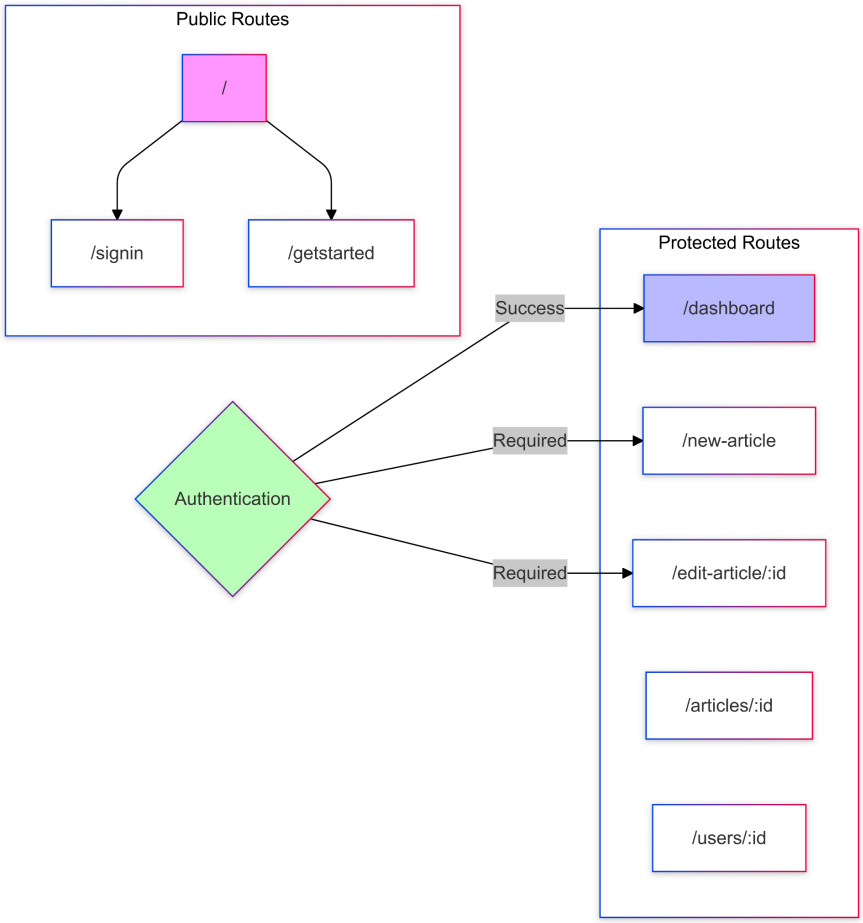
- Displays navigation links.
- Shows different options based on authentication state.
- Includes a logout mechanism.

Modal

- Generic modal component for displaying messages and actions.
- Used for confirmations, alerts, and information display.

ArticleSkeleton

- Placeholder component displayed during data loading.
- Enhances user experience by indicating content is being fetched.



Security Considerations

Authentication

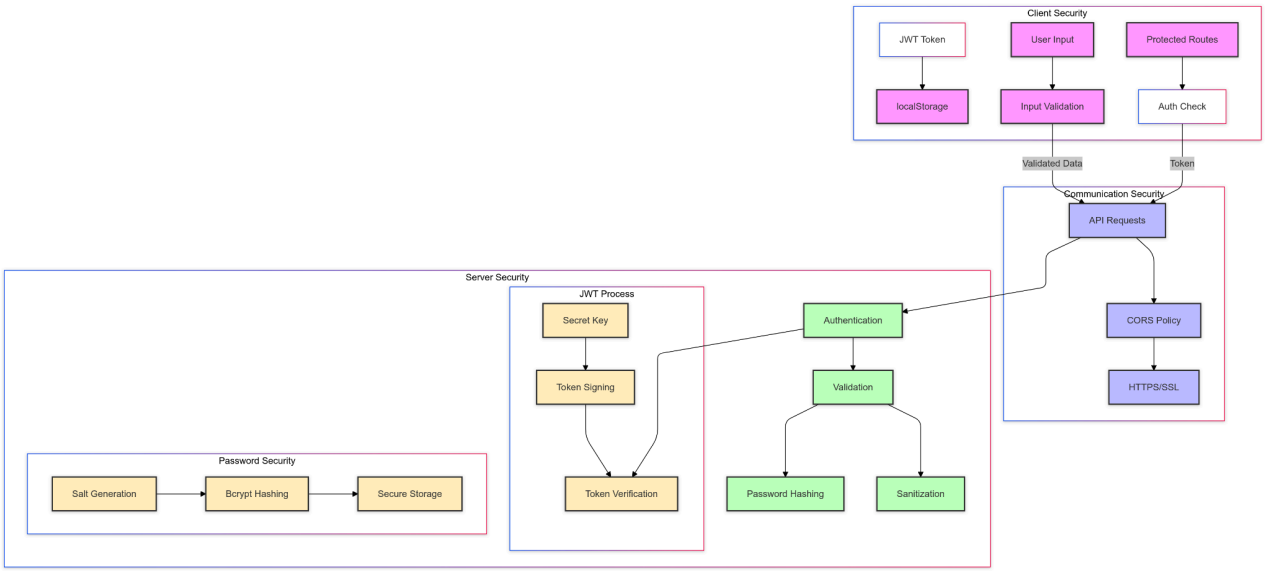
- **JWT Tokens**:
  - Securely generated and signed with a secret key.
  - Stored in the client's `localStorage`.
- **Password Security**:
  - Passwords hashed using `bcrypt` before storing in the database.
  - Plain passwords are never stored or logged.

Authorization

- **Protected Routes**:
  - Backend routes require valid JWT tokens.
  - Frontend routes use higher-order components to restrict access.
- **Input Validation**:
  - Sanitization of inputs to prevent injection attacks.
  - Validation rules applied on both client and server sides.

CORS Configuration

- **Access-Control Policies**:
  - Configured to allow requests from trusted origins.
  - Proper headers set for `Access-Control-Allow-Origin`, `Methods`, and `Headers`.



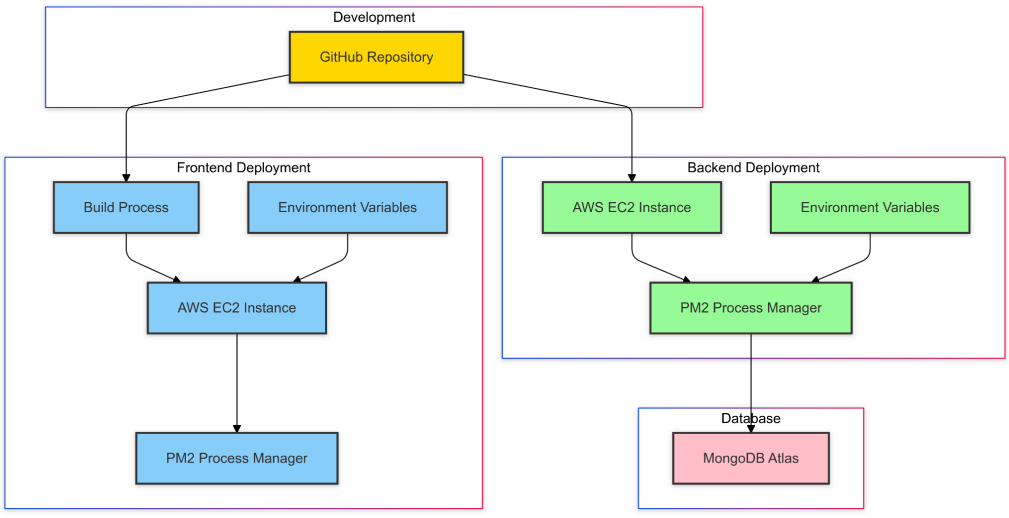
Deployment Plans

Environment Setup

- **Backend Environment Variables**:
  - `PORT`: Port number for the server.
  - `DB_CONNECTION_STRING`: MongoDB connection URI.
  - `JWT_SECRET`: Secret key for signing JWTs.
- **Frontend Environment Variables**:
  - `VITE_API_URL`: Base URL for the backend API.

Deployment Steps

1. **Backend Deployment**:
   - Host on platforms like Heroku, AWS EC2, or DigitalOcean.
   - Ensure environment variables are securely set.
   - Use process managers like PM2 for process management.
2. **Frontend Deployment**:
   - Build the React application using `npm run build`.
   - Host static files on services like Vercel or AWS.
3. **Domain and SSL**:
   - Configure a custom domain.
   - Set up SSL certificates for secure HTTPS communication.
4. **Database Hosting**:
   - Use managed MongoDB services like MongoDB Atlas.
   - Configure IP whitelisting and security measures.
5. **Continuous Deployment**:
   - Set up CD pipelines to automatically deploy on code changes.
   - Use GitHub Actions or other CI/CD tools.



Future Enhancements

Technical Improvements

- **Switch to TypeScript**:
  - Introduce TypeScript for type safety and better maintainability.
- **State Management Library**:
  - Implement Redux or Context API for more complex state needs.
- **WebSockets**:
  - Use Socket.IO for real-time features like live comments or notifications.

Feature Enhancements

- **Comment System**:
  - Allow users to comment on articles.
- **User Following**:
  - Implement a social feature where users can follow others.
- **Notifications**:
  - Real-time notifications for interactions.
- **Search Functionality**:
  - Implement search to find articles by title, content, or tags.
- **Analytics Dashboard**:
  - Provide users with insights on article views and interactions.

Conclusion

The Markd project is a full-stack web application that I've developed using modern technologies to ensure scalability, security, and a great user experience. By using React.js for the frontend, Express.js and Node.js for the backend, and MongoDB as our database, I've created a seamless platform for knowledge sharing.

During development, there was an emphasis on clear architecture with well-organized frontend and backend directories, modular components and well-defined routes.

Working on Markd has taught me a lot about web development and the importance of user-centered design. It is truly a fun experience.