5. Implementation (Source Code & Execution)

Source Code

- **Structured & Well-Commented Code**: The code is clean, maintainable, and properly documented with sufficient comments to explain the functionality of different sections.

- **Coding Standards & Naming Conventions**: The project follows consistent coding standards, with variable names written in camelCase. ESLint is used to enforce code quality and consistency.

- **Modular Code & Reusability**: The code is organized into reusable components. For example, the footer component is designed to be reused in multiple places within the application, ensuring a modular and maintainable structure.

- **Security & Error Handling**: Input validation is implemented to ensure data integrity and prevent security vulnerabilities. Additionally, any errors received from the backend are displayed appropriately, and the code has mechanisms for exception handling to ensure smooth operation even in case of failures.

- ## Frontend structure:

```
frontend/
├── yarn/
├── node_modules/
├── public/
│   ├── svgviewer-output.svg
│   └── vite.svg
├── src/
│   ├── components/
│   │   ├── common/
│   │   │   ├── Loader.jsx
│   │   │   └── PrivateRoute.jsx
│   │   ├── dashboard/
│   │   │   ├── GoalProgressChart.jsx
│   │   │   ├── StatCard.jsx
│   │   │   ├── WorkoutChart.jsx
│   │   │   └── WorkoutTypeChart.jsx
│   │   ├── goals/
│   │   │   ├── GoalForm.css
│   │   │   ├── GoalForm.jsx
│   │   │   ├── GoalList.css
│   │   │   └── GoalList.jsx
│   │   ├── layout/
│   │   │   ├── Footer.css
│   │   │   ├── Footer.jsx
│   │   │   ├── MobileNavbar.css
│   │   │   ├── MobileNavbar.jsx
│   │   │   ├── Navbar.css
│   │   │   ├── Navbar.jsx
│   │   │   ├── Sidebar.css
│   │   │   └── Sidebar.jsx
│   │   ├── profile/
│   │   │   ├── AvatarUpload.css
│   │   │   └── AvatarUpload.jsx
│   │   ├── theme/
│   │   │   ├── ThemeProvider.jsx
│   │   │   └── ThemeToggle.jsx
│   │   └── workouts/
│   ├── pages/
│   │   ├── AboutUs.jsx
│   │   ├── ContactUs.jsx
│   │   ├── Dashboard.jsx
│   │   ├── GoalTracking.jsx
│   │   ├── LandingPage.jsx
│   │   ├── Login.jsx
│   │   ├── Profile.jsx
│   │   ├── Register.jsx
│   │   └── WorkoutLog.jsx
│   ├── redux/
│   │   ├── slices/
│   │   │   ├── authSlice.js
│   │   │   ├── goalSlice.js
│   │   │   ├── themeSlice.js
│   │   │   └── workoutSlice.js
│   │   └── store.js
│   ├── services/
│   │   └── api.js
│   ├── styles/
│   │   ├── about-contact-pages.css
│   │   ├── auth-pages.css
│   │   ├── goal-tracking.css
│   │   ├── landing-page.css
│   │   └── workout-log.css
│   ├── App.css
│   ├── App.jsx
│   ├── AppWrapper.jsx
│   ├── index.css
│   └── main.jsx
├── .dockerignore
├── .env
├── .gitattributes
├── .gitignore
├── Dockerfile.dev
├── docker-compose.yml
├── eslint.config.js
├── index.html
├── package.json
├── pnp.cjs
├── pnp.loader.mjs
├── vite.config.js
└── yarn.lock
```

Explanation of the Structure:

- `frontend/` (Root Folder): This is the main directory containing all your frontend code.

- `yarn/` and `node_modules/`: These are related to your package manager (Yarn or npm). `node_modules` contains all the installed dependencies, and `yarn/` might contain Yarn-specific files.

- `public/`: This folder holds static assets that are served directly by the web server. In your case, it contains SVG files.

- `src/` (Source Code): This is the heart of your frontend application, containing all your React code.

    - `components/`: This directory is for reusable UI components. It's further organized into subdirectories based on their purpose or feature:

        - `common/`: Contains components used across different parts of the application (e.g., `Loader` for loading states, `PrivateRoute` for protected routes).
        - `dashboard/`: Contains components specific to the dashboard section.
        - `goals/`: Contains components related to the goal-tracking feature.
        - `layout/`: Contains components that define the overall structure and layout of the application (e.g., `Footer`, `Navbar`, `Sidebar`).
        - `profile/`: Contains components related to the user profile.
        - `theme/`: Contains components for managing the application's theme.
        - `workouts/`: (Empty in the provided images, but likely for workout-related components).
    - `pages/`: This directory contains components that represent individual screens or views of your application. Each file here is typically a top-level component that might use other components from the `components/` directory.

    - `redux/`: This directory is dedicated to managing the application's state using Redux (or a similar state management library).

        - `slices/`: Contains individual "slices" of the Redux store, each responsible for managing a specific part of the application state (e.g., `authSlice` for authentication state, `goalSlice` for goals state).
        - `store.js`: Configures and exports the Redux store.
    - `services/`: This directory typically contains code for interacting with APIs or external services.

        - `api.js`: Likely contains functions for making API calls.
    - `styles/`: This directory holds your CSS files. You've organized them based on specific pages or features, which is a good practice for maintainability.

    - `App.css`, `App.jsx`, `AppWrapper.jsx`, `index.css`, `main.jsx`: These are core files for your React application:

        - `App.jsx`: The root component of your application.
        - `App.css`: Global styles or styles for the `App` component.
        - `AppWrapper.jsx`: Might be a component that wraps `App` to provide context or global functionality.
        - `index.css`: Base or global styles that are applied to the entire application.
        - `main.jsx`: The entry point of your React application, responsible for rendering the `App` component into the DOM.

- **Configuration and Utility Files:**
  - `.dockerignore`, `Dockerfile.dev`, `docker-compose.yml`: Files related to Docker for containerizing your application.
  - `.env`: Stores environment variables.
  - `.gitattributes`, `.gitignore`: Files for Git version control.
  - `eslint.config.js`: Configuration for the ESLint linter.
  - `index.html`: The main HTML file that serves as the entry point in the browser.
  - `package.json`: Contains project metadata, dependencies, and scripts.
  - `pnp.cjs`, `pnp.loader.mjs`, `yarn.lock`: Files related to Yarn's Plug'n'Play feature.
  - `vite.config.js`: Configuration for the Vite build tool.

Benefits of this Structure:

- Organization: Keeps your code well-organized and easy to navigate.
- Maintainability: Makes it easier to find, understand, and modify code.
- Scalability: Provides a solid foundation for larger applications.
- Collaboration: Makes it easier for multiple developers to work on the same project.
- Separation of Concerns: Clearly separates UI components, page logic, state management, and API interactions.

- ## Backend structure :

```
backend/

├── config/          # Configuration files (database, JWT, Cloudinary, etc.)

│   ├── db.config.js

│   ├── jwt.config.js

│   ├── cloudinary.config.js

│   └── index.js       # (To aggregate and export config files)

├── controllers/       # Logic for handling requests (data manipulation)

│   ├── auth.controller.js
│   ├── goal.controller.js
│   ├── user.controller.js
│   ├── workout.controller.js
│   └── index.js       # (To aggregate and export controllers)
├── middlewares/       # Middleware functions for request/response processing
│   ├── auth.middleware.js
│   ├── error.middleware.js
│   └── index.js       # (To aggregate and export middlewares)
├── models/            # Data model definitions (Schemas)
│   ├── auth_token.model.js
│   ├── goal.model.js
│   ├── user.model.js
│   ├── workout.model.js
│   └── index.js       # (To aggregate and export models)
├── routes/            # API route definitions
│   ├── auth.routes.js
│   ├── goal.routes.js
│   ├── user.routes.js
│   ├── workout.routes.js
│   └── index.js       # (To aggregate and export routes)
├── tests/             # Test files
│   ├── config/
│   │   ├── db.config.test.js
│   │   ├── jwt.config.test.js
│   │   └── cloudinary.config.test.js
│   ├── controllers/
│   │   ├── auth.controller.test.js
│   │   ├── goal.controller.test.js
│   │   ├── user.controller.test.js
│   │   └── workout.controller.test.js
│   ├── middlewares/
│   │   ├── auth.middleware.test.js
│   │   └── error.middleware.test.js
│   ├── models/
│   │   ├── auth_token.model.test.js
│   │   ├── goal.model.test.js
│   │   ├── user.model.test.js
│   │   └── workout.model.test.js
│   └── server.test.js
├── utils/              # General utility functions
│   ├── dateHelper.js
│   ├── hashing.js
│   ├── logger.js
│   └── validator.js
```

```
├── .dockerignore
├── .editorconfig
├── .env
├── .env.example
├── .env.test
├── .gitattributes
├── .gitignore
├── .yarnrc.yml
├── Dockerfile
├── Dockerfile.dev
├── install-state.gz
├── logs/            # (Optional folder for storing log files)
├── package.json
├── README.md
├── server.js        # Main server entry point
└── yarn.lock
```

Explanation of the Proposed Structure:

- `config/`: Contains all your application's configuration files (database connection details, JWT secrets, Cloudinary API keys, etc.). An `index.js` file here helps in easily importing these configurations.
- `controllers/`: Holds the logic for handling incoming requests and interacting with the models. Each controller typically manages a specific part of your application's functionality (e.g., authentication, goals, users, workouts). An `index.js` here simplifies importing all controllers.

- `middlewares/`: Contains middleware functions that execute before or after request processing. Examples include authentication checks and error handling. An `index.js` helps in importing these functions.
- `models/`: Defines your data models (schemas) that represent the structure of your data in the database. Each file usually corresponds to a database table or collection. An `index.js` makes it easy to import all models.
- `routes/`: Defines the API routes for your application. Each file specifies the available endpoints and how the server should respond to them. An `index.js` helps in aggregating and exporting all routes.
- `tests/`: Contains all your automated test files for different parts of your backend (controllers, models, middlewares, etc.). Organizing tests in a similar structure to your main code makes it easier to find related tests.
- `utils/`: Stores general utility functions that can be used across different parts of your application (e.g., date formatting, password hashing, logging, data validation).
- Root Files:
    - Files starting with `.` are configuration files for specific tools (Docker, EditorConfig, Git, Yarn, environment variables).
    - `Dockerfile` and `Dockerfile.dev` are used for building Docker images to containerize your application.
    - `install-state.gz` is likely an internal file for Yarn.
    - `logs/` is an optional directory for storing application log files.
    - `package.json` manages your project's dependencies and scripts.
    - `README.md` provides a general description of your project.
    - `server.js` is the main entry point to start your Node.js server.
    - `yarn.lock` ensures consistent dependency installations with Yarn.

Benefits of this Structure:

- Clarity and Organization: Makes your codebase easier to understand and navigate.
- Maintainability: Separation of concerns makes it easier to make changes or fix bugs in a specific part of the application without affecting others.
- Scalability: Facilitates adding new features and organizing them logically.
- Testability: A clear structure makes it easier to write and run tests for individual units of your application.
- Better Collaboration: Makes it easier for a team of developers to work together effectively when the project structure is well-defined.

2. Version Control & Collaboration

- **Version Control Repository:**
  - The project is hosted on GitHub. You can access the repository via the following link:(https://github.com/samahali/fitness-tracker-dashboard-react.git).
- **Branching Strategy:**
  - The **main** branch is used as the primary branch for the project. All changes are merged into the **main** branch after work on features or bug fixes is completed in separate branches.
- **Commit History & Documentation:**
  - Commit messages are written in a clear and meaningful manner, with each commit providing a brief description of the changes made. For example, the following format is used:
    - [ ] `Fix: Fix bug in ...`
    - [ ] `Add: Add feature ...`
    - [ ] `Update: Update ...`
  - When creating Pull Requests, detailed descriptions of the changes are provided, explaining what has been modified and why, making it easier for reviewers to understand the updates.
- **CI/CD Integration:**
  - **CI/CD** is integrated with **Render**, where build, test, and deployment processes are automated. This ensures that every new change is tested and deployed automatically.

3. Deployment & Execution

Installation Steps

- **Clone the repository**

```
https://github.com/samahali/fitness-tracker-dashboard-react.git

cd fitness-dashboard
```

- **Install dependencies**
- [**Docker**]()
- **Docker Compose**

  Run the project:

```
docker-compose up --build
```

This will:

- **Build and install frontend & backend dependencies**
- **Start MongoDB, backend API, and frontend app**

🖥️ System Requirements

Hardware:

- Minimum: 4 GB RAM, Dual-Core CPU
- Recommended: 8 GB RAM, Quad-Core CPU

Software:

- Docker v20+
- Docker Compose v2+
- Web browser (Chrome/Edge/Firefox)

⚙️ Configuration Instructions

Create environment files before running:

frontend:

https://github.com/omnyatarek/environment-/blob/main/.env

backend:

https://github.com/omnyatarek/backend-env

▶️ Execution Guide

🖥️ Run Locally (via Docker)

```
docker-compose up --build
```

- Frontend: http://localhost:3000
- Backend API: http://localhost:5173/api

💻 Alternative (Manual Run - Dev Mode)

If you want to run without Docker:

- Start MongoDB locally

Run backend:

```
cd server
yarn install
yarn start
```

- Run frontend:

```
cd client
yarn install
yarn dev
```

📡 API Documentation

🔐 Auth Endpoints

Method	Endpoint	Description
POST	`/api/auth/register`	Register a new user
POST	`/api/auth/login`	Login with credentials

🔸 Example Request – Register

```
POST /api/auth/register

Content-Type: application/json

{

  "name": "Omnya",

  "email": "omnya@example.com",

  "password": "12345678"

}
```

🏋️ Workouts Endpoints

Method	Endpoint	Description
GET	`/api/workouts`	Get all workouts
POST	`/api/workouts`	Add a new workout

PUT	`/api/workouts/:id`	Update a specific workout
DELETE	`/api/workouts/:id`	Delete a specific workout

🔸 Example Request – Add Workout

```
POST /api/workouts

Authorization: Bearer <token>

Content-Type: application/json

{

  "type": "Cardio",

  "duration": 30,

  "caloriesBurned": 250

}
```

🎯 Goals Endpoints

Method	Endpoint	Description
GET	`/api/goals`	Get all user goals
POST	`/api/goals`	Add a new fitness goal

🔸 Example Request – Add Goal

```
POST /api/goals

Authorization: Bearer <token>

Content-Type: application/json
```

```
{
  "title": "Lose 5kg in 2 months",
  "deadline": "2025-06-01"
}
```

📌 Notes:

- **All protected routes require an `Authorization` header with a JWT token:**
  `Authorization: Bearer <your_token_here>`
- **All dates should be in ISO format (`YYYY-MM-DD`)**
- **All responses are in JSON format.**

📦 Executable Files & Deployment Link

✅ Deployed App

- **backend url**

  https://api-fitpulse-dashboard.onrender.com/

- **frontend url**

  https://fitpulse-dashboard.onrender.com/

🐳 Docker Deployment (Local Executable)

If you prefer a local build:

```
docker-compose up --build
```

🧪 Testing & Quality Assurance

✅ Test Cases & Test Plan

The following test scenarios are covered to ensure the functionality of the Fitness Tracker Dashboard:

🔸 **Auth Endpoints:**

1. **POST `/api/auth/register`**
   Test Case: Register a new user with valid details.
   Expected Outcome: User is created, and a JWT token is returned.

2. **POST `/api/auth/login`**
   Test Case: Login with valid credentials.
   Expected Outcome: Successful login with a JWT token.

🔸 **Workouts Endpoints:**

3. **GET `/api/workouts`**
   Test Case: Fetch all workouts for a logged-in user.
   Expected Outcome: List of workouts is returned.

4. **POST `/api/workouts`**
   Test Case: Add a new workout.
   Expected Outcome: New workout is added and stored in the database.

5. **PUT `/api/workouts/:id`**
   Test Case: Update a specific workout by ID.
   Expected Outcome: The workout details are updated.

6. **DELETE `/api/workouts/:id`**
   Test Case: Delete a workout by ID.
   Expected Outcome: The workout is deleted from the database.

🔸 **Goals Endpoints:**

7. **GET `/api/goals`**
   Test Case: Fetch all fitness goals for a logged-in user.
   Expected Outcome: List of goals is returned.

8. **POST `/api/goals`**
   Test Case: Add a new fitness goal.
   Expected Outcome: New goal is added and stored in the database.

7. Final Presentation & Reports

User Manual :

Registration & Login:

- **Registration: New users can sign up by providing their `name`, `email`, and `password`. After successful registration, a confirmation email is sent.**
- **Login: Registered users can log in using their email and password. Upon successful login, the user receives a JWT token for authentication.**

About Page:

- **The About Page provides an overview of the Fitness Tracker Dashboard, its features, and the technologies used in the development process.**
- **It also includes information on the team behind the project and its vision.**

Contact Page:

- **The Contact Page allows users to reach out to the development team or support team for any questions, feedback, or issues.**
- **Includes a contact form to submit inquiries directly from the page.**
- **Displays alternative ways to contact, such as email or social media links.**

Dashboard:

- **Workouts: Users can log workouts by selecting workout type, entering duration, and calories burned. They can also update or delete existing workouts.**

- **Goals: Users can set fitness goals, track progress, and view all goals in a list. Goals can be edited or deleted.**

profile page:

- **The Profile Page allows users to view and manage their personal information, including their name, email, and workout history.**

- **Users can update their personal details, and change profile picture.**

Technical Documentation

1. System Architecture:

The application follows a client-server architecture:

- **Frontend: React app (using Vite) for the user interface.**
- **Backend: Node.js + Express for API handling and business logic.**
- **Database: MongoDB for data storage (workouts, goals, users).**
- **Authentication: JWT tokens for secure login and authentication**

Entities & Relationships

1. **Users** (Stores user account details)
   - One user can have many fitness records.
   - One user can set multiple fitness goals.
   - One user can have authentication credentials.
2. **Fitness_Data** (Stores user activity logs)
   - Belongs to a user.

  - Contains workout details (type, duration, calories burned, etc.).

3. **Fitness_Goals** (Stores user-defined fitness goals)
   - Belongs to a user.
   - Tracks progress based on fitness records.
4. **Auth_Tokens** (Stores user authentication tokens for session management)
   - Linked to users for login sessions.

3. Physical Schema (Database Tables & Attributes)

Users Table

Column Name	Data Type	Constraints	Description
user_id	UUID (PK)	PRIMARY KEY	Unique identifier for the user
name	VARCHAR(100)	NOT NULL	User's full name
email	VARCHAR(255)	UNIQUE, NOT NULL	User's email address
password	TEXT	NOT NULL	Hashed password for authentication
age	INT	NULLABLE	User's age
weight	FLOAT	NULLABLE	User's weight (kg/lbs)

height	FLOAT	NULLABLE	User's height (cm/in)
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Account creation timestamp

Fitness_Records Table

Column Name	**Data Type**	**Constraints**	**Description**
record_id	UUID (PK)	PRIMARY KEY	Unique identifier for record
user_id	UUID (FK)	FOREIGN KEY -> Users(user_id)	User who logged the activity
activity_type	VARCHAR(50)	NOT NULL	Type of activity (e.g., running, cycling)
duration	INT	NOT NULL	Duration in minutes
calories	FLOAT	NOT NULL	Calories burned
date	DATE	NOT NULL	Date of workout
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Timestamp of record creation

Fitness_Goals Table

Column Name	Data Type	Constraints	Description
goal_id	UUID (PK)	PRIMARY KEY	Unique identifier for goal
user_id	UUID (FK)	FOREIGN KEY -> Users(user_id)	User setting the goal
goal_type	VARCHAR(50)	NOT NULL	Type of goal (e.g., weight loss, running distance)
target_value	FLOAT	NOT NULL	Target value (e.g., 5kg weight loss, 10km run)
progress	FLOAT	DEFAULT 0	Progress towards the goal
deadline	DATE	NULLABLE	Deadline to achieve the goal
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Timestamp of goal creation

Auth_Tokens Table

Column Name	Data Type	Constraints	Description
token_id	UUID (PK)	PRIMARY KEY	Unique identifier for token
user_id	UUID (FK)	FOREIGN KEY -> Users(user_id)	User who owns the token
token	TEXT	NOT NULL, UNIQUE	Authentication token
expires_at	TIMESTAMP	NOT NULL	Expiration time for token

3. Normalization Considerations

- **1NF (First Normal Form)**: Each table has atomic values; no duplicate columns.
- **2NF (Second Normal Form)**: No partial dependencies; all non-key attributes depend on the primary key.
- **3NF (Third Normal Form)**: No transitive dependencies; attributes only depend on their table's primary key.

- **Project Presentation**

  https://docs.google.com/presentation/d/1IUiET-UUIrYFFbb26pEWGPcpfVbSzbTShYPw8AeEjws/edit#slide=id.p