**CS673 Software Engineering** 

**Team 3 - Health and Wellness Manager**

**Software Design Document**

| **Team Member** | **Role(s)** | **Signature** | **Date** |
| --- | --- | --- | --- |
| Amanda Yee | QA Leader | *Amanda Yee* | 9/22/2024 |
| Edward Lee | Config Leader | *Edward Lee* | 9/22/2024 |
| Kenny Light | Requirements Leader | *Kenny Light* | 9/22/2024 |
| Yu Luo | Design & Implementation Leader | *Yu Luo* | 9/22/2024 |
| Zihao Qian | Security Leader | *Zihao Qian* | 9/22/2024 |
| Chris Ceravolo | Team Leader | *Chris Ceravolo* | 9/22/2024 |

**Revision history**

| **Version** | **Author** | **Date** | **Change** |
| --- | --- | --- | --- |
| 1 | Team 3 | 9/23/2024 | All sections |
| 2 | Amanda Yee | 10/07/2024 | - Updated Class Diagram, Database Design, and Key Algorithms section |
| 2 | Chris Ceravolo | 10/07/2024 | - Update Business Logic  - Update APIs |
| 2 | Yu Luo | 10/07/2024 | - Update UI Design |
| 2 | Edward Lee | 10/07/2024 | - Update Security Design |

[Introduction](#_heading=h.gjdgxs)

[Software Architecture](#_heading=h.30j0zll)

[Class Diagram](#_heading=h.1fob9te)

[UI Design (if applicable)](#_heading=h.3znysh7)

[Database Design (if applicable)](#_heading=h.2et92p0)

[Security Design](#_heading=h.tyjcwt)

[Business Logic and/or Key Algorithms](#_heading=h.3dy6vkm)

[Design Patterns](#_heading=h.1t3h5sf)

[Any Additional Topics you would like to include.](#_heading=h.2s8eyo1)

[References](#_heading=h.17dp8vu)

[Glossary](#_heading=h.3rdcrjn)

# Introduction

This project is a web-based application with a technology stack that includes a React front end and a Node.js/Express back end, connected to a MongoDB cloud database. The system is currently set up for deployment on Heroku (tentative). Testing is managed with Jest, and project management is done using Jira.

# Software Architecture

### Key Frameworks and Libraries Used

* **React**: Front-end framework used for building a dynamic, single-page web application.
* **Express.js**: A minimal and flexible Node.js web application framework that provides robust features for building APIs.
* **Mongoose**: A MongoDB object modeling tool designed to work in an asynchronous environment.
* **Jest**: A JavaScript testing framework used to test React components and APIs.
* **Axios**: A promise-based HTTP client used for making requests to the back-end API.

### 

### 

### Decomposition of Software System

#### **Component 1: Front-End (React)**

* **Folder**: client
* **Role**:
  + The front end serves the user interface (UI) using React to render views.
  + Handles routing between different UI components using react-router-dom.
  + Fetches data from the backend API using Axios (axios dependency).
  + It also includes testing libraries (@testing-library/jest-dom, @testing-library/react, and @testing-library/user-event).
* **Main Dependencies**:
  + **React** (react, react-dom): For UI rendering.
  + **Axios**: For making API requests.
  + **React Router**: Handles client-side routing.
  + **Testing Libraries**: @testing-library/\* for testing front-end components.
* **Interfaces**:
  + Interacts with the backend via RESTful API calls (using Axios).
  + Communicates with user actions through event handlers.
* **Key Files**:
  + App.js: Main entry point for the React application
  + components/: Contains reusable UI components
  + services/: Handles API interactions (Axios calls)
  + utils/: Contains helper functions for application

#### **Component 2: Back-End (Node.js, Express.js)**

* **Folder**: server
* **Role**:
  + The backend serves as an API layer, built using Node.js and Express.js.
  + It handles requests from the front end (RESTful API) and interacts with MongoDB for database operations (using Mongoose).
  + Middleware, cors, is used to manage cross-origin requests.
* **Main Dependencies**:
  + **Express.js**: For handling API requests and routing.
  + **Mongoose**: For interacting with the MongoDB database.
  + **CORS**: For enabling cross-origin requests from the React front end.
* **Interfaces**:
  + Exposes RESTful API endpoints for the front-end to consume.
  + Interfaces with MongoDB through Mongoose for CRUD operations on data.
* **Key Files**:
  + server.js: Main entry point for the Express server. Currently contains routing.
  + initdb.js: Testing Only
  + db.js: Database Connectivity
  + routes: Defines the API endpoints.
  + models: Contains Mongoose models that map to MongoDB collections.
  + controllers:Handles business logic for each API endpoint.

#### **Component 3: Database (MongoDB)**

* **Role**:
  + Provides persistent storage for your application's data.
  + The connection to MongoDB is handled via the Mongoose library.
* **Main Dependency**:
  + **Mongoose**: Maps MongoDB documents to JavaScript objects and provides an interface to interact with the database.
* **Interfaces**:
  + The backend connects to MongoDB through Mongoose, handling data operations such as CRUD (Create, Read, Update, Delete).

#### **Component 4: Testing (Jest)**

* **Role**:
  + Testing is done using Jest, primarily for the front-end.
  + The @testing-library suite is used to test the React components' functionality.
* **Main Dependencies**:
  + **Jest**: For running unit tests.
  + **@testing-library/**\*: For testing React components and DOM interactions.

#### **Component 5: Deployment (Heroku)**

* **Role**:
  + The deployment of the full stack (React front end and Node.js back end) is handled by Heroku.
  + The build and deployment scripts will ensure that both the front-end and back-end components are properly deployed and configured.
* **Interfaces**:
  + The Heroku platform serves the front-end and back-end, enabling end users to interact with the application.

### Relationship Between Components

* **Front End (React)** communicates with the **Back End (Node.js/Express)** via RESTful API calls using Axios.
* The **Back End** retrieves and manipulates data from the **Database (MongoDB)** using Mongoose ORM.
* **GitHub** is used for repository and workflow scripts for continuous integration/continuous deployment to **Heroku,** which will host both the front-end and back-end components.

**Front end:**

* **React**: A JavaScript library for building user interfaces, specifically single-page applications (SPA).
* **Front End Dependencies**
  + **jest-dom**: Provides additional DOM testing capabilities when working with Jest.
  + **react**: A library used for testing React components by simulating user interactions and asserting UI behavior.
  + **user-event**: Enhances tests by simulating user input like typing, clicking, and other interactions.
  + **axios**: Handles API requests from the front end to the back end.
  + **react**: The core React library for building the front-end UI.
  + **react-dom**: Used for rendering React components in the DOM.
  + **react-router-dom**: A routing library for handling client-side navigation in React applications.
  + **react-scripts**: Provides configuration and tools for building the React project.
  + **web-vitals**: A utility for collecting performance metrics like Largest Contentful Paint (LCP) and First Input Delay (FID).

**Back end:**

* **Node.js**: JavaScript runtime used to build the server-side logic.
* **Express.js**: A web application framework for Node.js that simplifies API endpoint creation and HTTP request handling.
* **MongoDB (via Mongoose)**: A NoSQL database for storing application data, with Mongoose providing object data modeling (ODM) for MongoDB.

**Web Deployment:**

* **Heroku**: A cloud platform as a service (PaaS) for deploying and managing the web application.

**Management Software:**

* **Jira**: Used for task management, issue tracking, and project coordination.

**Testing:**

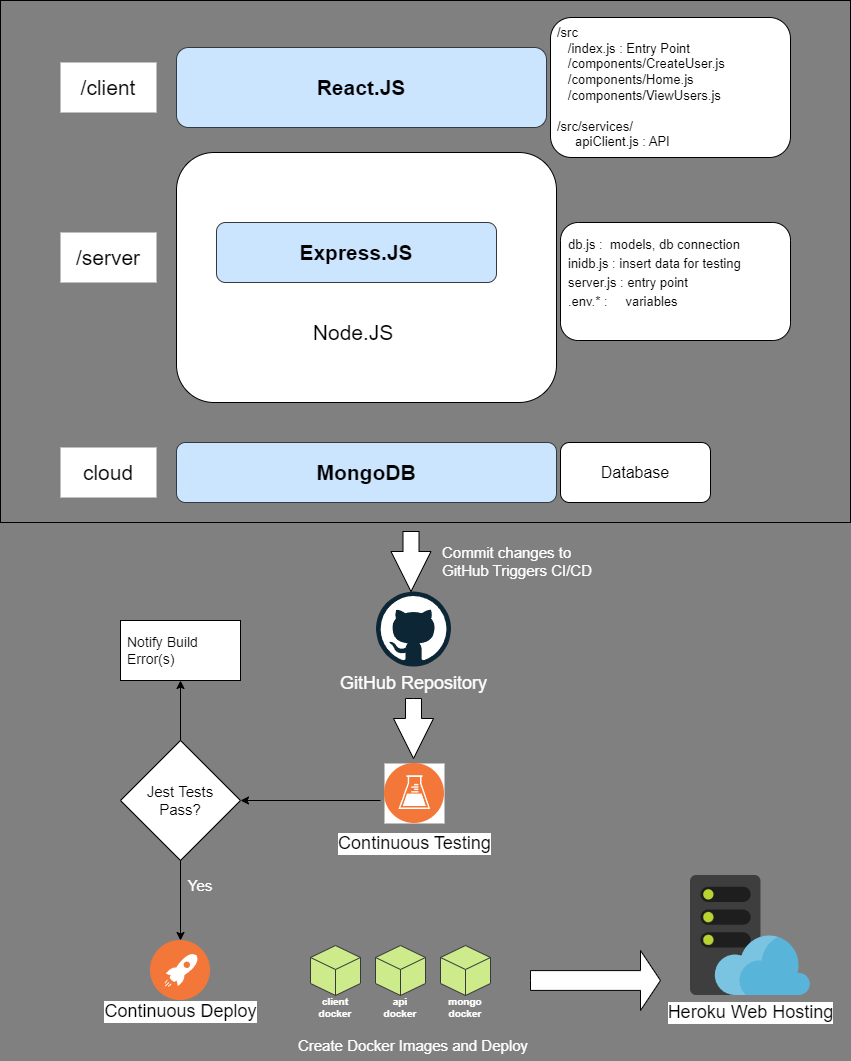
* **Jest**: A JavaScript testing framework used to write unit tests for both the front end and back end.

#### **Server-Side Dependencies**

The following dependencies are utilized on the back end:

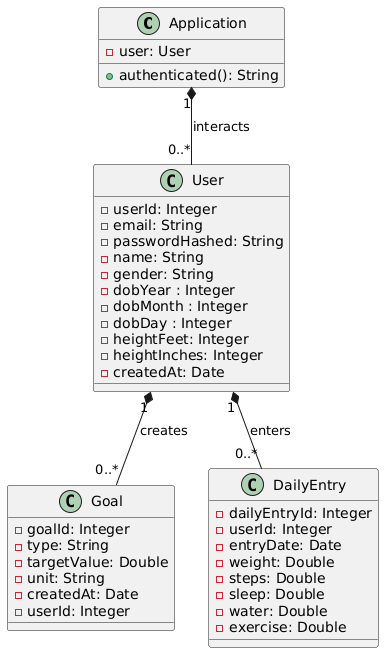
* **cors**: Enables cross-origin resource sharing, allowing the front end to communicate with the back end on a different domain or port.
* **express**: A minimal and flexible Node.js framework used for creating server routes and handling requests.
* **mongoose**: An ODM library that provides schema-based modeling and interaction with MongoDB databases.

# Architectural Diagram



# Class Diagram

*In this section, you will provide a detailed description of each component (or package) and use one or multiple class diagrams to show the main classes and their relationships in each component.*



**User Class**: Represents user-related information and their goals

* **userId**: Will be autogenerated by the application based on the maximum value of the existing user IDs in the database. Starts at value 10001.
* **email**: This acts as the username
* **passwordHashed**: The user will set their password and it will be hashed before it is saved to the database for security purposes
* **name**: The user will set their name
* **gender**: The user will specify their gender
* **dobYear:** The user will specify their date of birth year
* **dobMonth:** The user will specify their date of birth month
* **dobDay:** The user will specify their date of birth day
* **heightFeet:** The user will specify their height (feet)
* **heightInches:** The user will specify their height (inches)
* **createdAt:** The date when the user was created (defaults to the current date)

**Goal Class**: Represents the goals set by users for health-related metrics

* **goalId**: Will be autogenerated by the application based on the maximum value of the existing goal IDs in the database. Starts at value 50001.
* **type**: The type of goal, one of: sleep, weight, steps, exercise, or water
* **targetValue**: This is the target value for the given goal type that the user has set
* **unit**: The unit is determined based on the type:
  + “hours” for sleep
  + “lbs” for weight
  + “steps” for steps
  + “minutes” for exercise
  + “glasses” for water
* **createdAt:** The date the user created the goal (defaults to the current date)
* **userId:** The user ID who created the goal

**DailyEntry Class:** Represents the health data entered daily by each user

* **dailyEntryId:** Will be autogenerated by the application based on the maximum value of the existing daily entry IDs in the database. Starts at value 1.
* **userId:** The user ID who created the daily record
* **entryDate:** The date the user created the health entry for
* **weight:** The user-entered value for their weight on the given entry date
* **steps:** The user-entered value for number of steps taken on the given entry date
* **sleep:** The user-entered value for number of hours of sleep on the given entry date
* **water:** The user-entered value for number of glasses of water drank on the given entry date
* **exercise:** The user-entered value for exercise time on the given entry date

**Relationship**:

* There is a one-to-many relationship between users and goals, where a single user can create multiple goals, but each goal is associated with only one user.
* There is also a one-to-many relationship between users and goals, where a single user can create multiple daily entries for different days, but each entry is associated with one user.

# UI Design (if applicable)

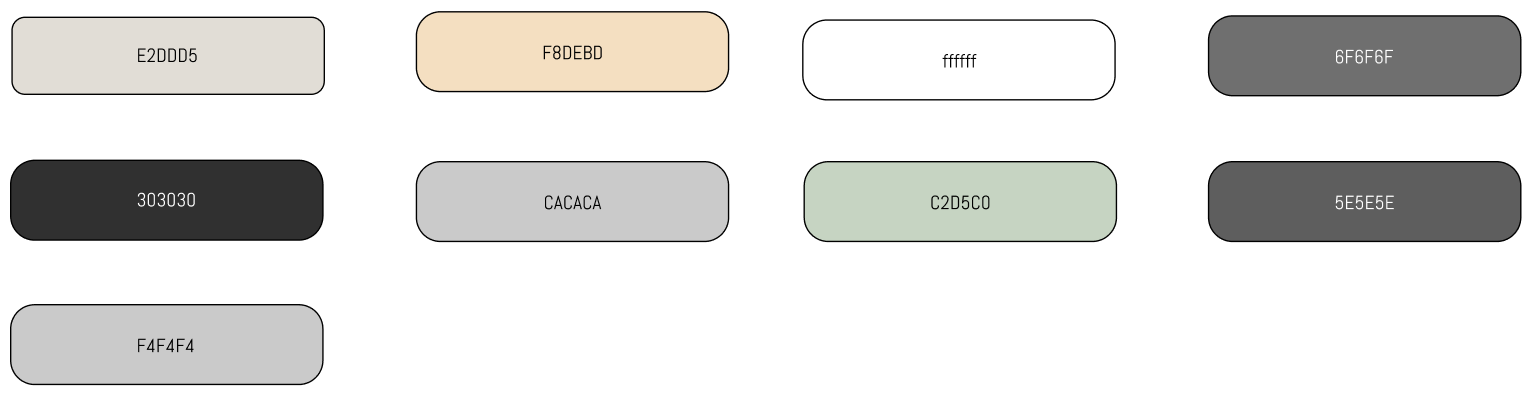
*In this section, you can describe your UI design*

We are using Figma to design the application’s user interface.

Link: [Figma Board](https://www.figma.com/design/VcXGJmgO54kiw4UjBzcu3Z/Health-and-Wellness-Manager?node-id=0-1&node-type=canvas&t=gnCoB5rELvK3hnHP-0)

**Font:** We are using the **Mulish** font throughout the application for a clean, modern look. It's a sans-serif font chosen for its readability and consistency across different devices.

**Color Scheme**

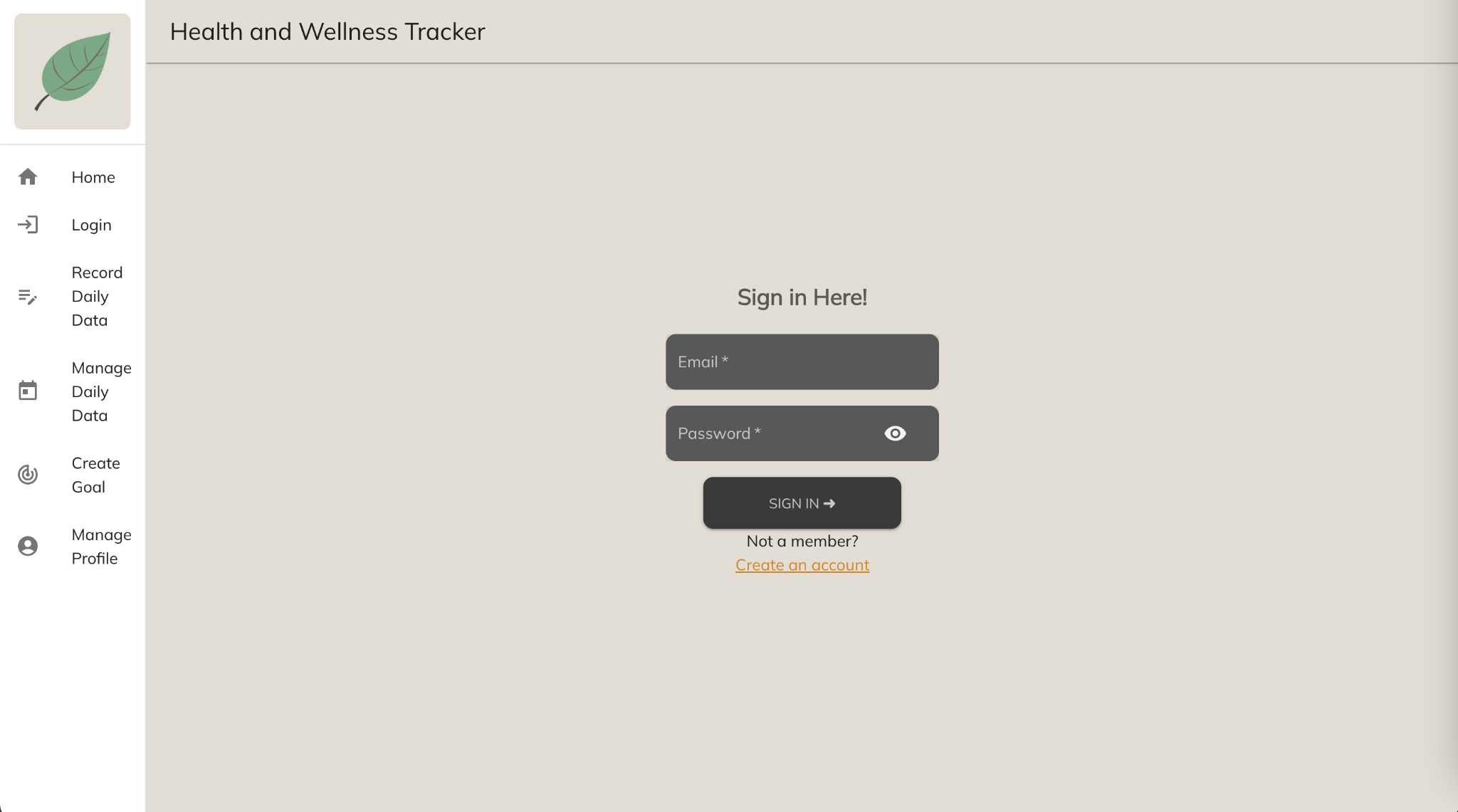


**Buttons and Forms**

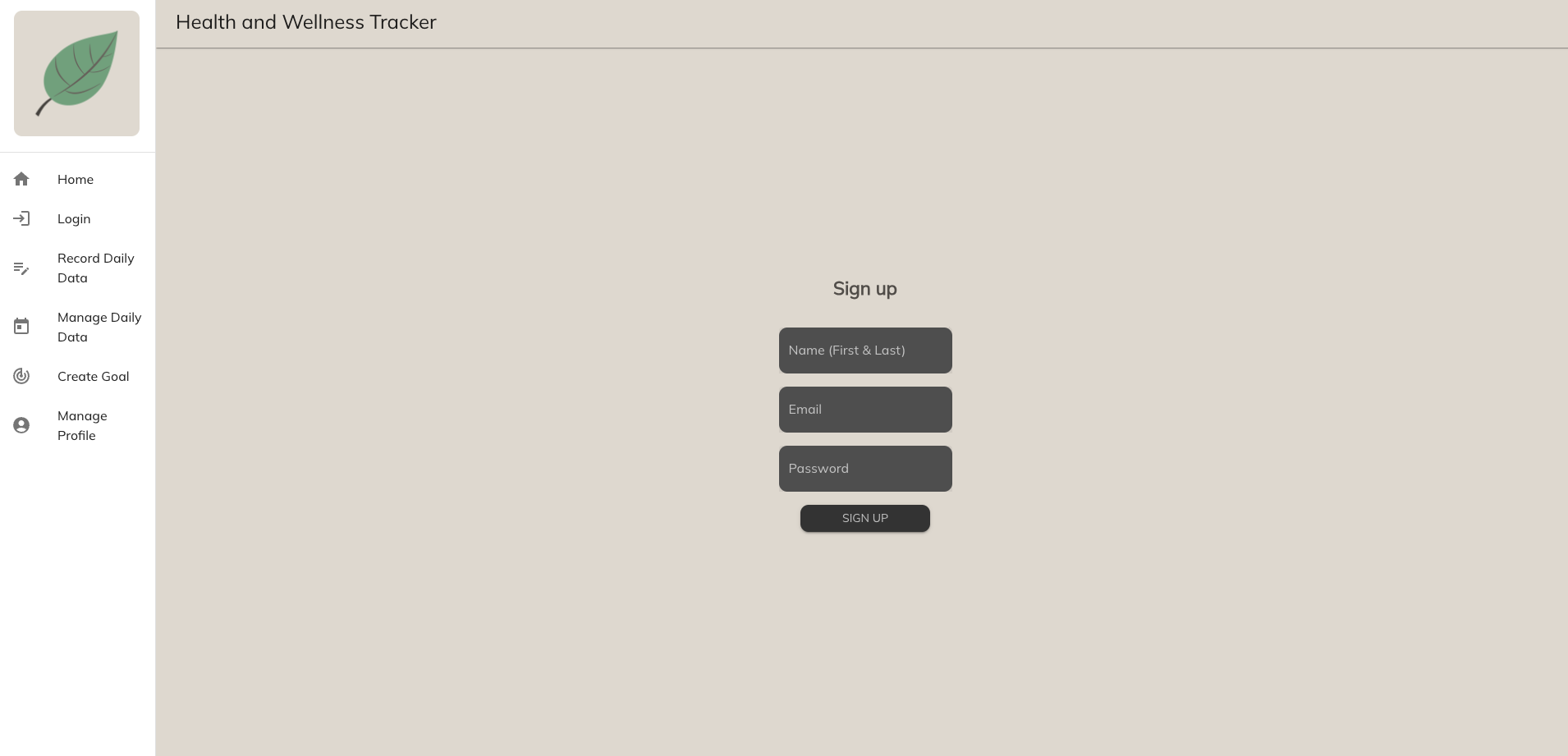
* + Buttons are designed with rounded corners and clear hover states
  + Forms use **Material-UI** components with filled inputs, providing a smooth user experience

**Pages**

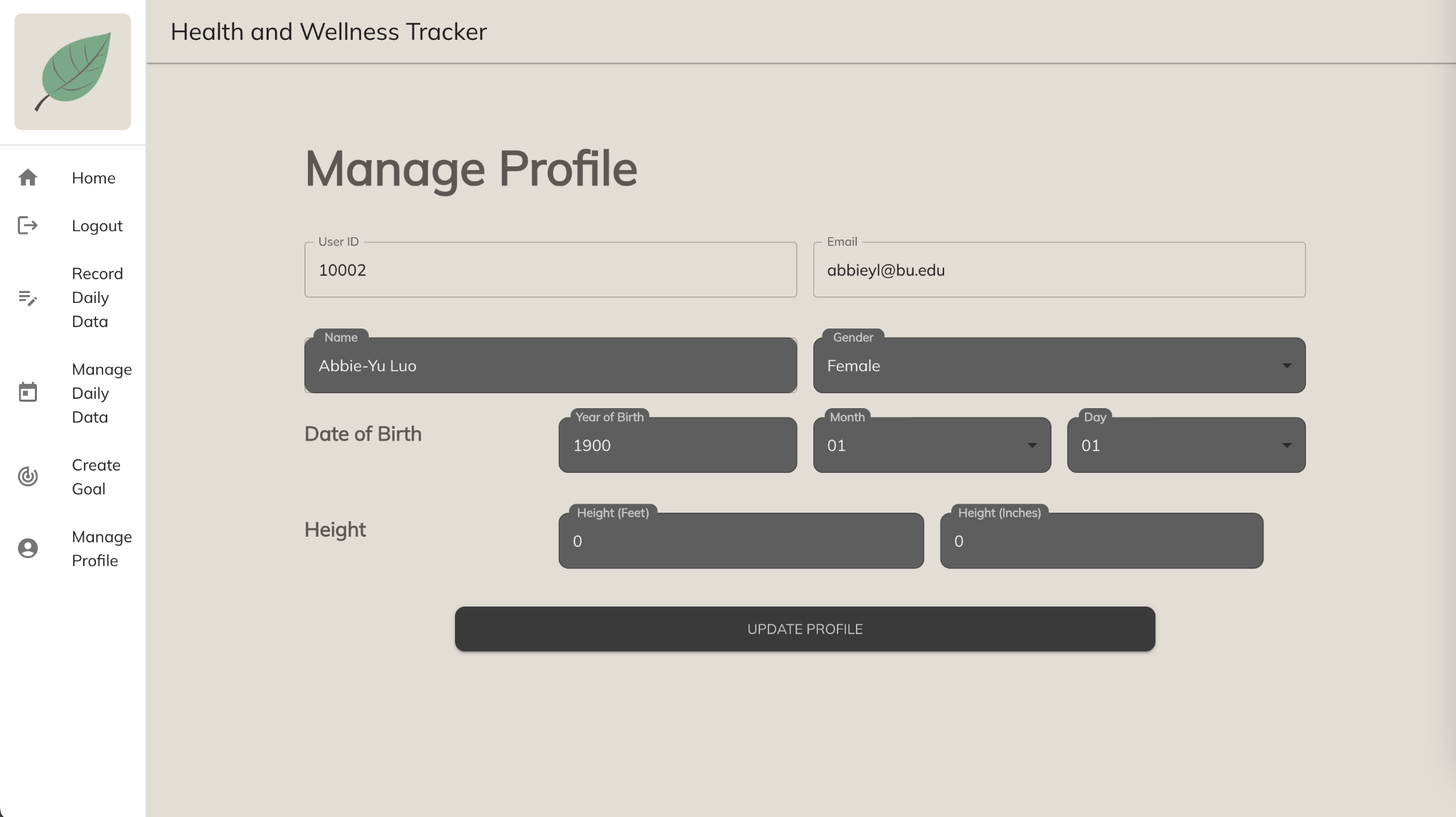
* + **Login Page** - This is where users can log in to their account



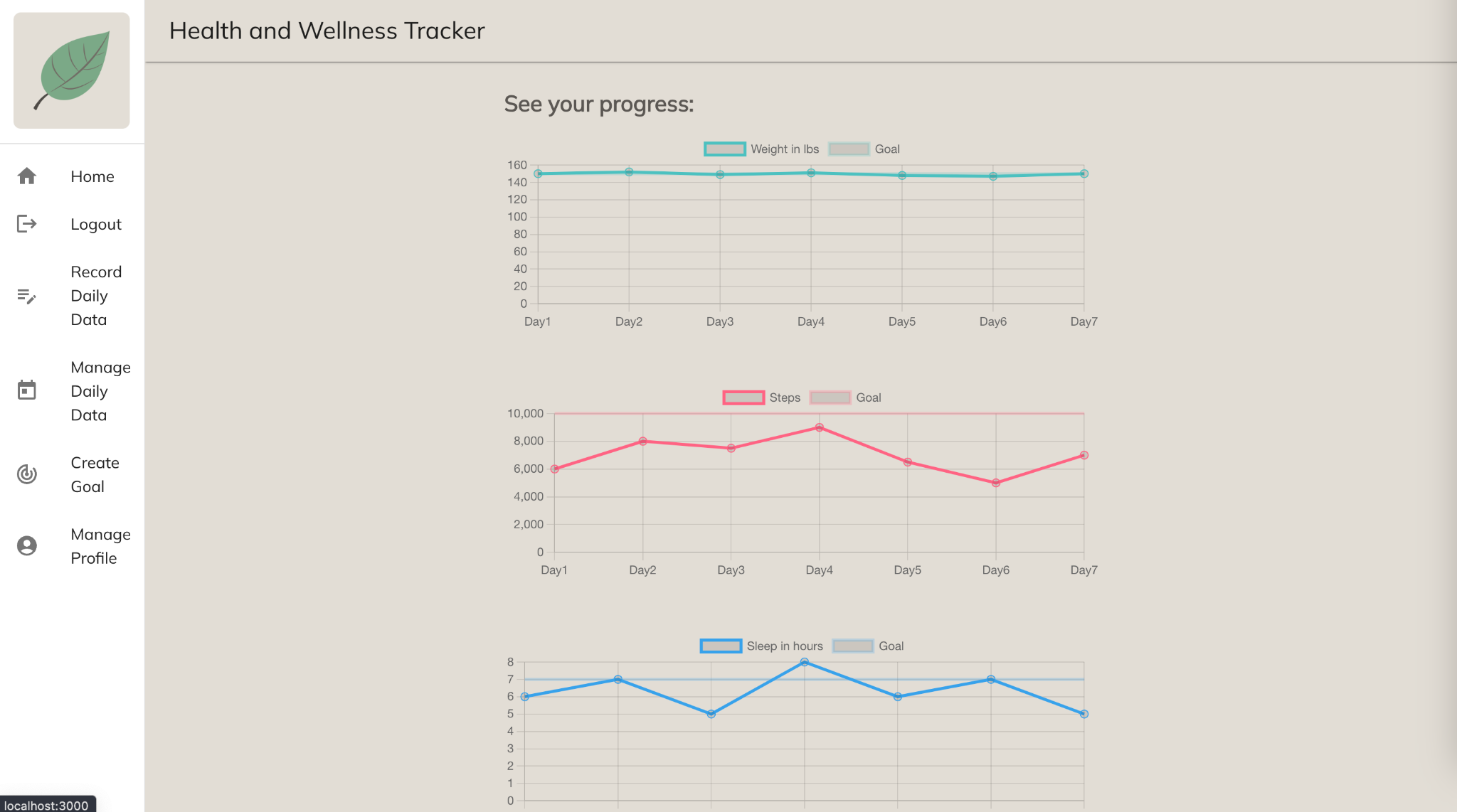
* + **Register Page** - This is where users can register the account and enter basic information



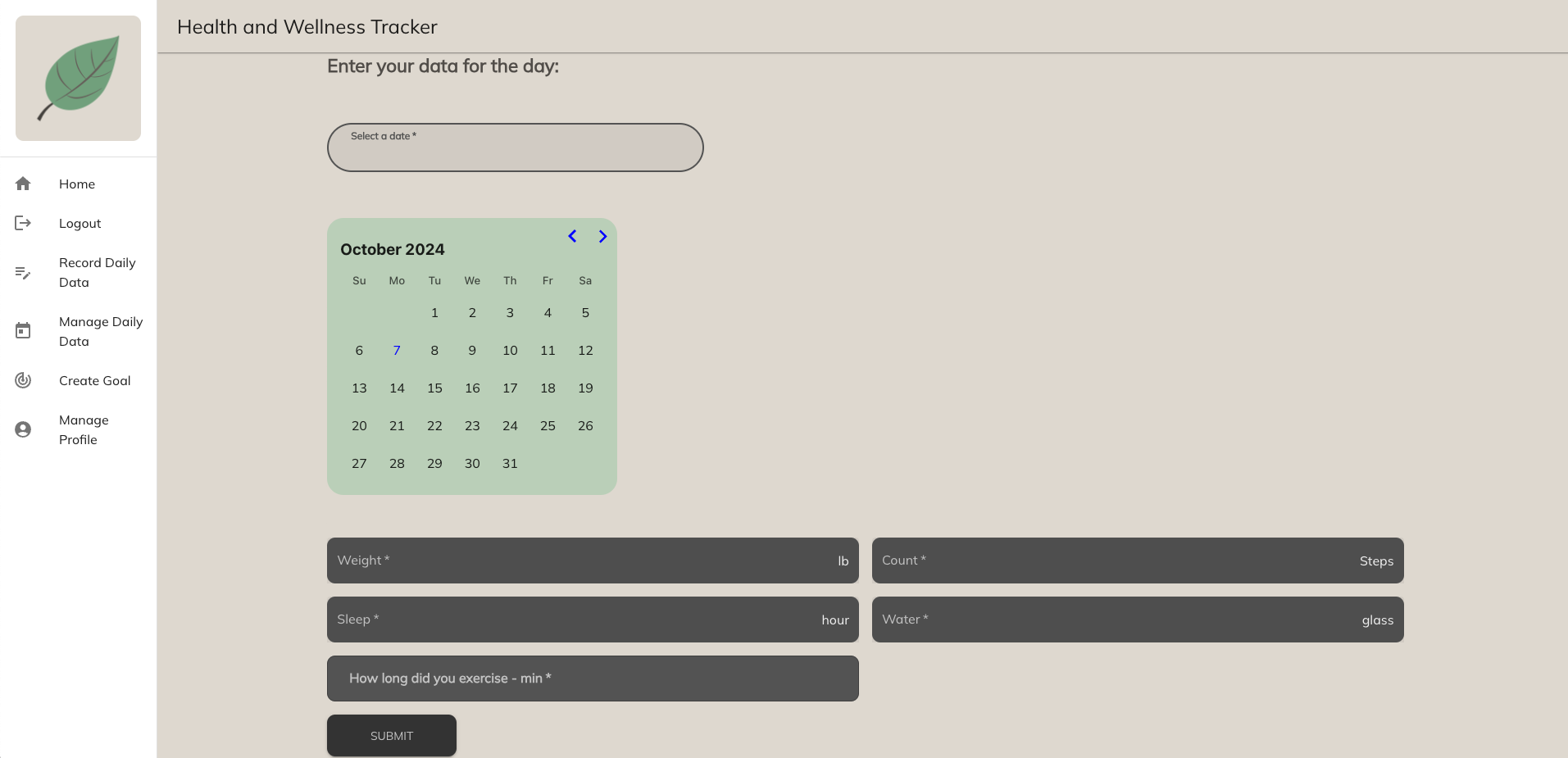
* + **Manage Profile** - This page allows users to view and update their personal details. The form includes fields for essential user information such as User ID, Email, Name, Gender, Date of Birth, and Height.



* + **Main Page** - This is where users can view an overview of their health data, including key metrics and insights into their daily habits and progress.



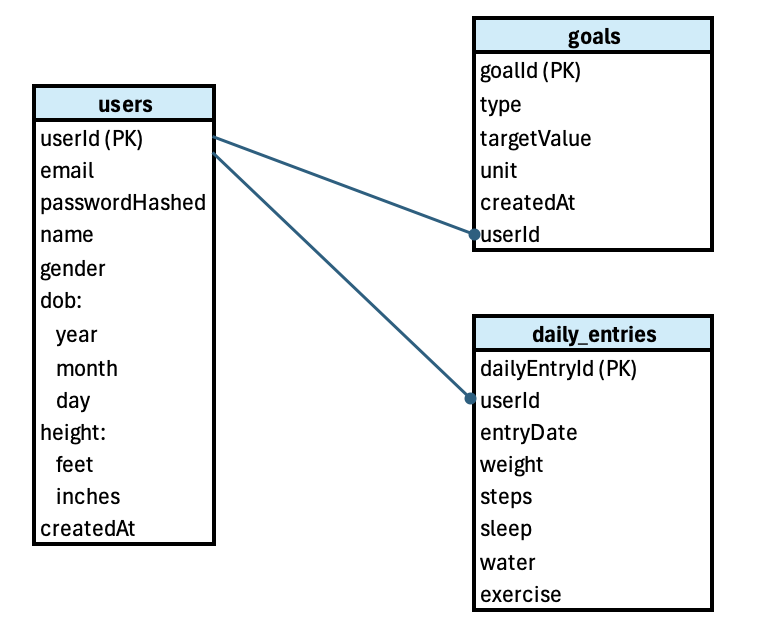
* + **Daily Health Data Submission Page** - This is where users can select a specific date and input their daily health data, tracking progress and maintaining a record of their wellness activities.



# Database Design (if applicable)

*In this section, you shall describe any database schema if used in your software system.*

We are using a MongoDB Atlas database. Below is a diagram of our database, along with a screenshot of an example document in each of the collections:



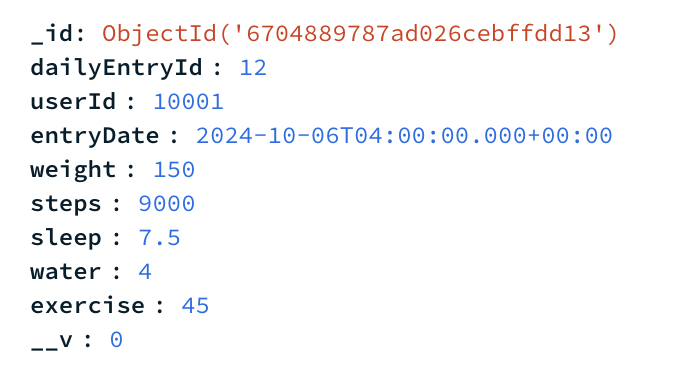
*users* example:



*goals* example:



*daily\_entries* example:



We are defining schema with the Mongoose library in Javascript as follows:

const userSchema = new Schema({

userId: { type: Number, required: true },

email: { type: String, required: true },

passwordHashed: { type: String, required: true },

name: { type: String, required: true },

gender: { type: String, default: "na" },

dob: {

year: { type: Number, default: 1900, min: 1900, max: 2024 },

month: { type: Number, default: 1, min: 1, max: 12 },

day: { type: Number, default: 1, min: 1, max: 31 },

},

height: {

feet: { type: Number, default: 0, min: 0, max: 7 },

inches: { type: Number, default: 0, min: 0, max: 12 },

},

createdAt: { type: Date, default: Date.now }

}, {

collection: 'users',

});

const goalSchema = new Schema({

goalId: { type: Number, required: true },

type: { type: String, required: true },

targetValue: { type: Number, required: true },

unit: { type: String, required: true },

createdAt: { type: Date, default: Date.now },

userId: { type: Number, required: false }

}, {

collection: 'goals',

});

const dailyEntrySchema = new Schema({

dailyEntryId: { type: Number, required: true },

userId: { type: Number, required: true },

entryDate: { type: Date, default: Date.now },

weight: { type: Number },

steps: { type: Number },

sleep: { type: Number },

water: { type: Number },

exercise: { type: Number },

}, {

collection: 'daily\_entries',

});

# Security Design

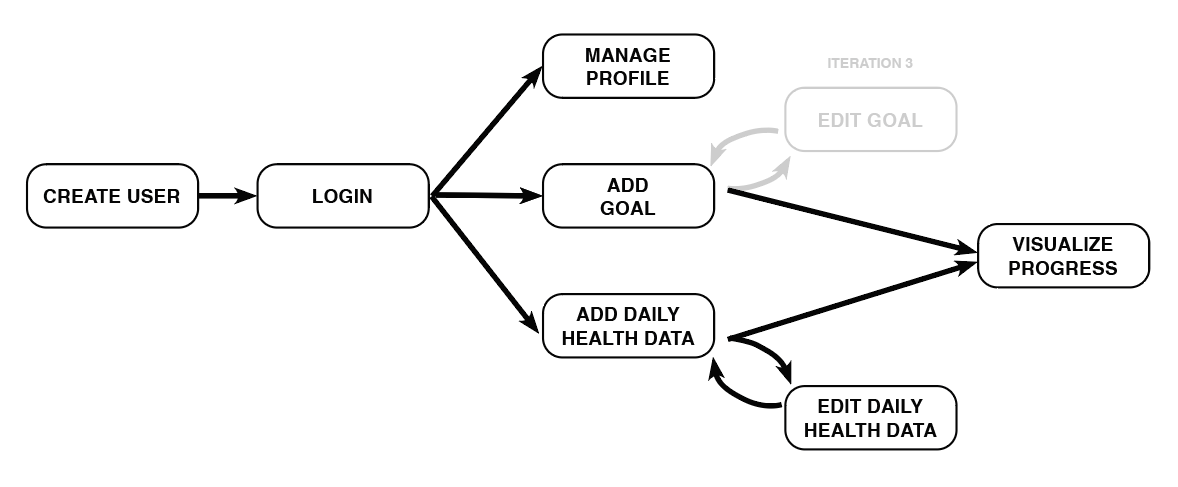
* All pages will redirect to login (with a create profile subpage) until the user is authenticated.
* Securely encrypt user passwords stored in our MongoDB database, using **bcrypt**. This is a library that will hash the password before saving to the database. This is a one-way process, so you cannot retrieve the original password.
* Enforce strong password policies.
  + Use **zxcvbn** to provide password strength feedback.
  + Use **password-validator** to define password rules and validate passwords against those rules.
* Implement HTTPS for secure data transmission. Heroku provides this by default.
* Use **DOMPurify** (sanitization library) to cleanse form input data to mitigate the risk of cross-site scripting (XSS) attacks. Control user permissions when implementing multiple roles.
* Use **express-validator** for validating and sanitizing input on the server side.
* Use **dotenv** to manage environment variables. Ensure the environment variable files (.env\*) are not included in our version control.
* Use GitHub & Heroku to store secrets. This can be used to store information like API keys, database credentials, and secret tokens.
* Use JWT to store user ID to query for information specific to the logged in user. Tokens will expire after 1 hour.
* Login password masking - When a user types their password in the login screen, it is displayed as dots or asterisks (● or \*), which prevents anyone nearby from seeing the actual characters being typed.

# Business Logic and/or Key Algorithms

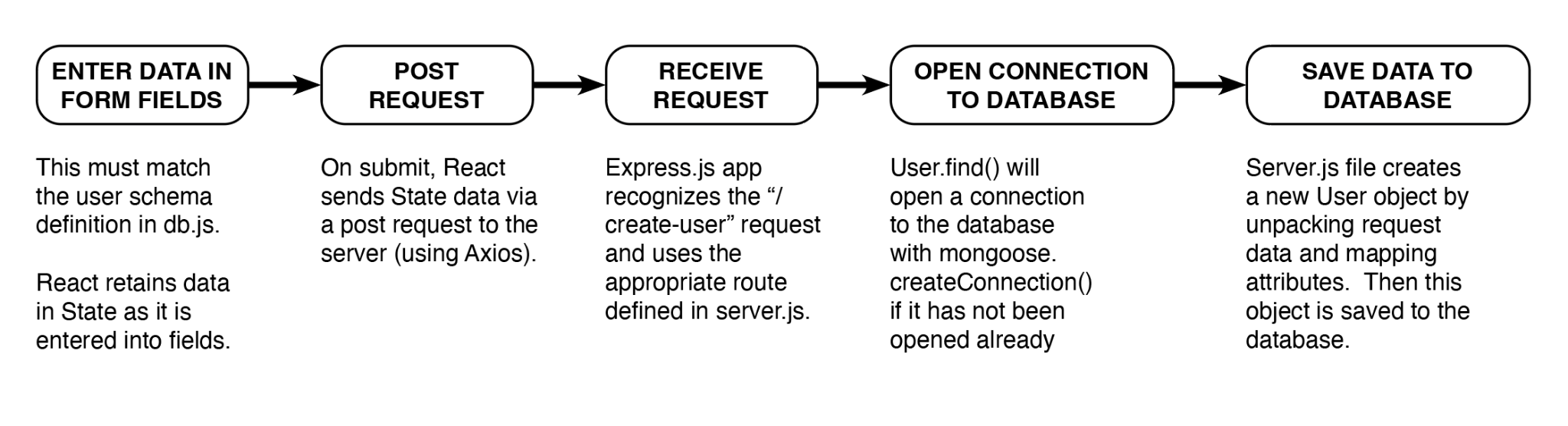
*In this section, you shall describe any key algorithms used in your software system, either in terms of pseudocode or flowchart, or sequence diagrams.*

## 1. Business Logic

The key application flow is:



We are developing detailed flows for each of these features. Currently, the app has the following functionality:

* Create User Flow:
  + 
* Create Goal Flow:
  + Similar logic as Create User Flow with initial authentication step when server receives request
* Add Daily Health Data Flow:
  + Similar logic as Create User Flow with initial authentication step when server receives request
* Other flows to be detailed in Iteration 3

## 2. Key Algorithms

* + Automatic generation of user IDs:
    - **let** newUserId = lastUser[0].userId + 1
  + Automatic generation of goal IDs:
    - **let** newGoalId = lastGoal[0].goalId + 1
  + Automatic generation of daily data IDs if no entry exists for the current day:
    - **let** newEntryId = lastEntry.length > 0 ? lastEntry[0].dailyEntryId + 1 : 1;
  + We may implement a simple algorithm for estimating calories burned from exercise in future iterations:
    - **Calories Burned = (MET value) × (Weight in lbs) × (Duration in hours)**
      * **MET value**: This is a value that represents the intensity of the exercise. For example, walking might be around 3-4 METs, while running could be 7-10 METs. We could estimate this generally by asking the user to estimate Low, Medium, or High intensity of their workout.
      * **Weight in lbs**: Your body weight.
      * **Duration in hours**: Convert minutes to hours by dividing by 60.

# Design Patterns

*In this section, you shall describe any design patterns used in your software system.*

We haven’t yet implemented any design patterns in our code. If we were to implement any design patterns, the following are potential options:

* Factory Method pattern for generation of HealthDataEntry classes e.g. a sleep entry, steps entry
* Observer pattern for user notifications
* Decorator pattern to add onto health data entries e.g. adding a comment about that particularly entry

# Rest APIs

* + GET /
    - Purpose:
      * Useful for testing connection to server
    - Request Headers:
      * N/A
    - Request Body:
      * N/A
    - Response Status Codes:
      * 200 ok
    - Response Data:
      * ‘Server is running…’
  + POST /api/users/create-user
    - Purpose:
      * Creates user in the database
    - Request Headers:
      * N/A
    - Request Body:
      * Name, email, password as per user model
    - Response Status:
      * 201 User created successfully
      * 500 Error creating user
    - Response Data:
      * N/A
  + POST /api/users/login
    - Purpose:
      * Authenticate user
    - Request Headers:
      * N/A
    - Request Body
      * email and password as per user model
    - Response Status:
      * 200 ok
      * 401 Invalid email or password
      * 500 Internal server error
    - Response Data:
      * {token: String, message: “Login successful”}
  + GET /api/users/view-users
    - Purpose:
      * Returns an array of all users from database in JSON format
    - Request Headers:
      * N/A
    - Request Body
      * N/A
    - Response Status:
      * 200 ok
      * 500 Error retrieving users
    - Response Data:
      * [

{

"dob": {

"year": 1900,

"month": 1,

"day": 1

},

"height": {

"feet": 0,

"inches": 0

},

"\_id": "6702f17c92cea23344096648",

"userId": 10004,

"email": "elee27@bu.edu",

"passwordHashed": "$2b$10$MUk35zK49rTvK45JXaH5peCdSmKmDUsXq4jEnPF1fmSK/zpwy39J.",

"name": "Eddie Lee",

"gender": "na",

"createdAt": "2024-10-06T20:22:20.218Z",

"\_\_v": 0

},

]

* + GET /api/users/manage-profile
    - Purpose:
      * Retrieves user’s existing profile data from the database
    - Request Headers:
      * Authorization: Bearer token
    - Request Body
      * N/A
    - Response Status:
      * 200 ok
      * 401 authorization or token is missing
      * 404 user not found
      * 500 server error
    - Response Data:
      * [

{

"dob": {

"year": 1900,

"month": 1,

"day": 1

},

"height": {

"feet": 0,

"inches": 0

},

"\_id": "6702f17c92cea23344096648",

"userId": 10004,

"email": "elee27@bu.edu",

"passwordHashed": "$2b$10$MUk35zK49rTvK45JXaH5peCdSmKmDUsXq4jEnPF1fmSK/zpwy39J.",

"name": "Eddie Lee",

"gender": "na",

"createdAt": "2024-10-06T20:22:20.218Z",

"\_\_v": 0

},

]

* + POST /api/users/update-profile
    - Purpose:
      * Update user’s profile data in the database
    - Request Headers:
      * Authorization: Bearer token
    - Request Body
      * userId, name, gender, dob, height as per user model
    - Response Status:
      * 200 profile updated successfully
      * 404 user not found
      * 500 error updating profile
    - Response Data:
      * {

"dob": {

"year": 1900,

"month": 1,

"day": 1

},

"height": {

"feet": 0,

"inches": 0

},

"\_id": "6702f17c92cea23344096648",

"userId": 10004,

"email": "elee27@bu.edu",

"passwordHashed": "$2b$10$MUk35zK49rTvK45JXaH5peCdSmKmDUsXq4jEnPF1fmSK/zpwy39J.",

"name": "Eddie Lee",

"gender": "na",

"createdAt": "2024-10-06T20:22:20.218Z",

"\_\_v": 0

}

* + POST /api/goals/create-goal
    - Purpose:
      * Create a goal object associated with your user id in the database
    - Request Headers:
      * Authorization: Bearer token
    - Request Body
      * Type and target value as per goal model
    - Response Status:
      * 201 new goal created successfully
      * 401 authorization or token is missing
      * 500 failed to create goal
    - Response Data:
      * { message: 'New goal created successfully', goalId: newGoalId }
  + POST /api/daily-entry/enter-daily-data
    - Purpose:
      * Create or update daily health data entries in the database
    - Request Headers:
      * Authorization: Bearer token
    - Request Body
      * weight, steps, sleep, water, exercise, entryDate as per dailyEntry model
    - Response Status:
      * 200 Daily entry updated successfully
      * 201 Daily entry created successfully
      * 400 Entry date is missing
      * 500 Error creating or updating daily entry
    - Response Data:
      * N/A
  + GET /api/daily-entry/view-daily-data
    - Purpose:
      * View the health data entries created by a given user in the database
    - Request Headers:
      * Authorization: Bearer token
    - Request Body
      * N/A
    - Response Status:
      * 200 ok
      * 401 Authorization or token is missing
      * 404 User daily data not found
      * 500 Server error
    - Response Data:
      * [

{

"\_id": "670351c5c0ef236410e50a11",

"dailyEntryId": 9,

"userId": 10003,

"entryDate": "2024-10-06T04:00:00.000Z",

"weight": 1,

"steps": 1,

"sleep": 1,

"water": 1,

"exercise": 1,

"\_\_v": 0

},

{

"\_id": "670351cdc0ef236410e50a17",

"dailyEntryId": 10,

"userId": 10003,

"entryDate": "2024-10-08T04:00:00.000Z",

"weight": 9,

"steps": 9,

"sleep": 9,

"water": 9,

"exercise": 9,

"\_\_v": 0

}

]

* + DELETE /api/daily-entry/delete-entry
    - Purpose:
      * Delete a health data entry in the database
    - Request Headers:
      * Authorization: Bearer token
    - Request Body
      * dailyEntryId
    - Response Status:
      * 200 daily entry deleted successfully
      * 401 authorization or token is missing
      * 404 daily entry not found
      * 500 error deleting daily entry
    - Response Data:
      * N/A

# Any Additional Topics You Would Like to Include

* + We tested deployment via Heroku and hit a paywall. Despite this, we’ve decided to move forward with it, as the cost is low, and we already have this set up.

# References

* + Module 3 notes re: Model/View/Controller architecture

# Glossary

* + N/A