Elevate Fit

A Comprehensive Fitness and Nutrition Web Application

Team Members:

Name	Email
Samuel Mbasela	mbaselasam@gmail.com
Semba Mbasela	mbaselasemba@gmail.com
Abdirahman Moalin	amn370@uregina.ca
Vatsal Badhiwala	vatsalbadhiwala99@gmail.com
Mark Bircher	markbircher@hotmail.com

Computer Science 476: Software Development Project

Department of Computer Science

University of Regina

Fall 2024

Submission Date: 17 November 2024

Course Instructor: Ali Bayeh

Project Description: Elevate Fit is a fitness and nutrition web application designed to support users in managing fitness goals, tracking nutrition plans, and optimizing overall wellness.

Final Project Report

1. Problem definition

Our project aims to assist individuals in tracking their activity and nutrition to promote healthier lifestyles. Many people face challenges in staying consistent with their health and fitness goals due to a lack of integrated and convenient tools to monitor physical activity, dietary intake, and progress over time. These challenges can get worse due to busy lifestyles, and the absence of insights derived from their health data.

While there are existing solutions in the market, they often focus exclusively on either nutrition or physical activity in isolation. This split approach forces users to rely on multiple applications, leading to inefficiencies and gaps in understanding how these two factors interact to impact overall health. Addressing this gap, our application combines activity tracking and nutrition monitoring into a single system, providing users with a more comprehensive and convenient way to achieve their health goals.

The application domain for this project is technology that focuses on health and fitness, as this domain includes tools, applications, and systems designed to promote physical well-being, and monitor and track health behaviors, such as dietary habits and exercise routines.

The motivations of our project encompass fostering long-term health and wellness, by connecting nutrition and fitness and promoting personalized approaches in health management. Our solution targets individuals striving to improve or maintain their physical health, recognizing the importance of consistent and accurate tracking in achieving a long and healthy life. For these users, our application aims to reduce the complexity of health management by streamlining the tracking process and integrating actionable feedback into their routines. Maintaining physical health is essential for achieving a long and fulfilling life. By providing tools that simplify health

management, our project offers valuable support to individuals striving to lead healthier lifestyles, empowering them to make informed decisions and stay consistent in their journey toward improved well-being.

2. Application benefits

Existing solutions for tracking nutritional and fitness data are great, but many services only track nutrition in isolation or fitness activity in isolation. Since we know this fact, we made sure our application would combine both tracking mechanisms for a more convenient application for users.

When compared to industry leaders like AppleHealth(https://www.apple.com/ca/health/), our application stands out by combining these two key functionalities. Users can track their nutrition and fitness activity simultaneously, creating opportunities to uncover trends that directly impact their fitness journey. This holistic approach not only empowers users to make informed decisions but also provides the added benefit of consolidating all fitness and nutritional data in one place, making it easier to use, reducing tracking errors, and enhancing motivation and accountability.

Another benefit of the application is the simplicity of the design when compared to an application like Macrofactor(https://macrofactorapp.com/), our app is deliberately made without a lot of extraneous features so that users can focus on the key features which are to track macros and fitness activity. When tracking calories we opted to use an api (Edamam) to a database that holds data on different foods so that when a user is entering the food they ate for a meal it can be a much more efficient process. The search for food is made efficient because it saves time for the user by automating the process of finding the nutritional details of a certain food and subtracting

it from their daily caloric intake goal. Furthermore, the UI shows this with a progress bar for calorie consumption that aids the user in visually seeing how much calories are left to be eaten. When tracking exercise we opted to use a manual approach of logging exercise data but made it significantly simpler by limiting logging data to only picking three modes of an exercise(cardio, strength training, stretching), the duration of the exercise, intensity of the exercise, and the user's current bodyweight. This minimalistic approach in the exercise logging page is crucial in keeping users' interest and ensures that logging fitness activity remains simple and engaging.. The results of the logged exercise data are displayed in a pie chart on the dashboard, offering a clear visual representation of activity distribution. This not only motivates users to keep logging their exercises but also provides a sense of accomplishment as they see their progress. Furthermore, the exercise page is capable of tracking the user's bodyweight and showing a trend in their body weight on the dashboard. The trend showing the users' body weight is a direct way the user can see the effects of the effort they put in when tracking their fitness activity and calorie consumption.

Why Our Application Stands Out

1. **Integration of Key Features**: Unlike many solutions that specialize in either nutrition or fitness, our application seamlessly combines both for a comprehensive user experience.
2. **Ease of Use**: Consolidating all fitness and nutrition data into one platform makes tracking simpler, minimizes errors, and improves overall usability.
3. **Motivational Visuals**: Progress bars, pie charts, and trend graphs provide users with actionable insights and a clear picture of their progress.

4. **Efficient Tracking**: Leveraging the Edamam API for nutrition and a streamlined manual approach for exercise ensures both accuracy and speed.
5. **Focus on Core Features**: By avoiding feature bloat, users can focus on tracking calories and fitness activity without distractions

By combining a clean, user-focused design with robust tracking capabilities, our application simplifies the process of managing fitness and nutrition. Whether users are working toward weight loss, muscle gain, or maintain weight, our application serves as a reliable companion in their fitness journey. It empowers users to stay accountable, discover meaningful trends, and make informed decisions to reach their goals efficiently.

3. Requirements Elicitation & Specification

User Role: Focused on personal health management, users will be able to track their activity, nutrition, and progress. They can manually log their activities on the exercise page and add meals on the log food page.

1. Account **configuration** (User doesn't exist) :

- Register
- Enter personal details *(Sex, Weight, Height, Age)*
- Choose goal *(Weight loss, Maintenance, Gain)*

2. Account "**maintenance**" (User exists)

Dashboard page with today's summary (including profile information, exercise log, and nutrition macronutrients charts), **Exercise page** where workouts can be added. Workouts can be selected based on activity type, duration and intensity, and **Log Food page** (like myfitnesspal breakfast, lunch, dinner, snacks, etc.), **Edit profile** (edit name, fitness goal etc.)

- Can view profile items on the dashboard.
- Add/remove meals.
- Add workouts for calories burned that day.
- Add weight data for a given week whenever the user feels like showing progress.

Admin Role: Admins will have oversight of the system, managing users and providing insights, such as personalized recommendations based on user data. They will also access higher-level data analytics for system-wide insights.

3. a) Functional Requirements List for Each user role:

1. **User Role (Regular User):**

    - **Register an Account:** Users can register for their account by providing personal details such as *(name, sex, weight, height and age)*. Users can also set a fitness goal *(weight loss, maintenance, gain).*
    - **Log Daily activities:** Users can manually log exercise data *(such as cardio, strength training, or stretching)*, the number of minutes spent on the activity, and the intensity. The calories spent will automatically be calculated based on the activity. They can also log their body weight for tracking.
    - **Track Nutrition:** Users can manually log their meals throughout their day *(Breakfast, Lunch, Dinner, Snacks)* and summarize their calorie intake daily/ weekly.
    - **Track Progress:** Users can track the progress of their goal by accessing various charts and graphs in the dashboard such as *(exercise log, daily macronutrients and macronutrients over time).*

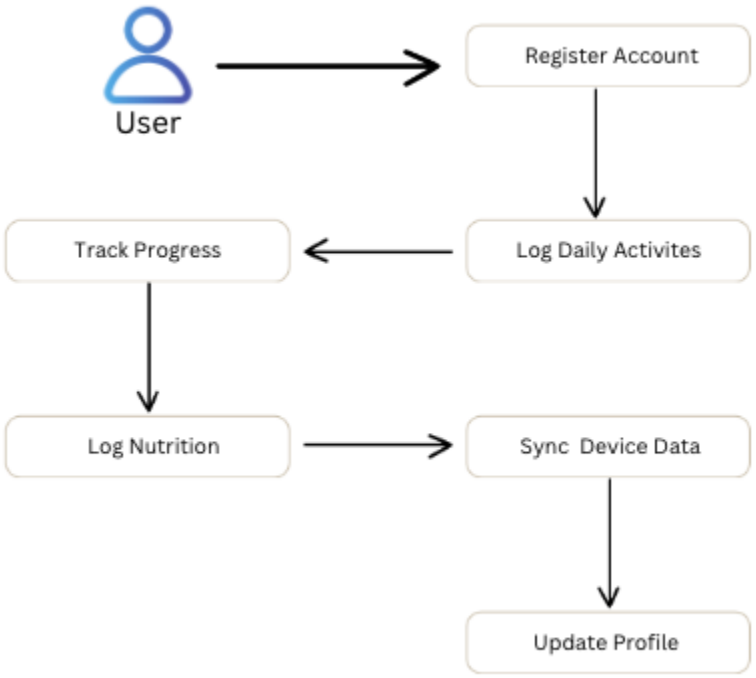
- **Update Profile Information:** Users can edit their profile and update other account details like *goals, weight etc.*

2. **Admin Role:**

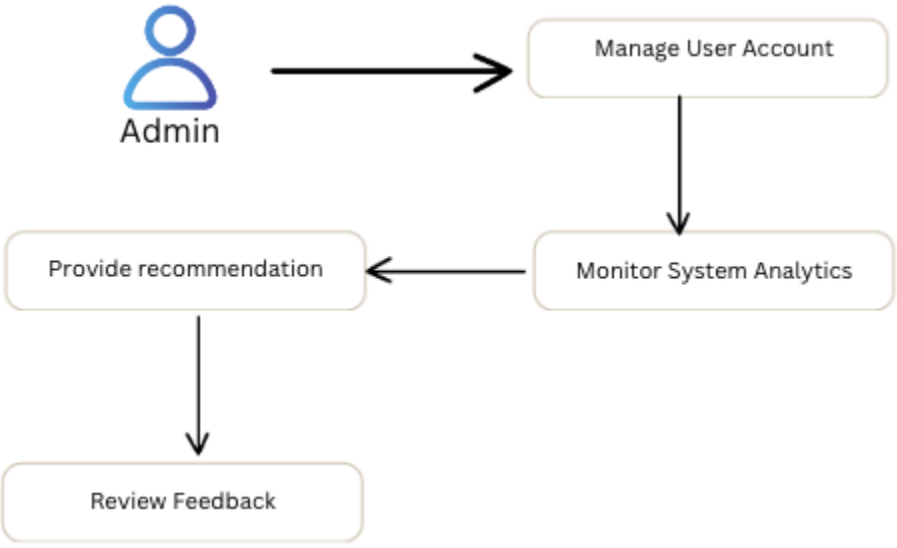
    - **Manage User Accounts:** Admins can view and modify user information *(e.g., reset passwords, remove accounts, change emails).*
    - **Monitor App Analytics:** Admins can see higher level data analytics for system wide insights.

3. b) Use Case diagrams with Use cases and actors:

1. **User Role Case Diagram:**
   Actors: *User*

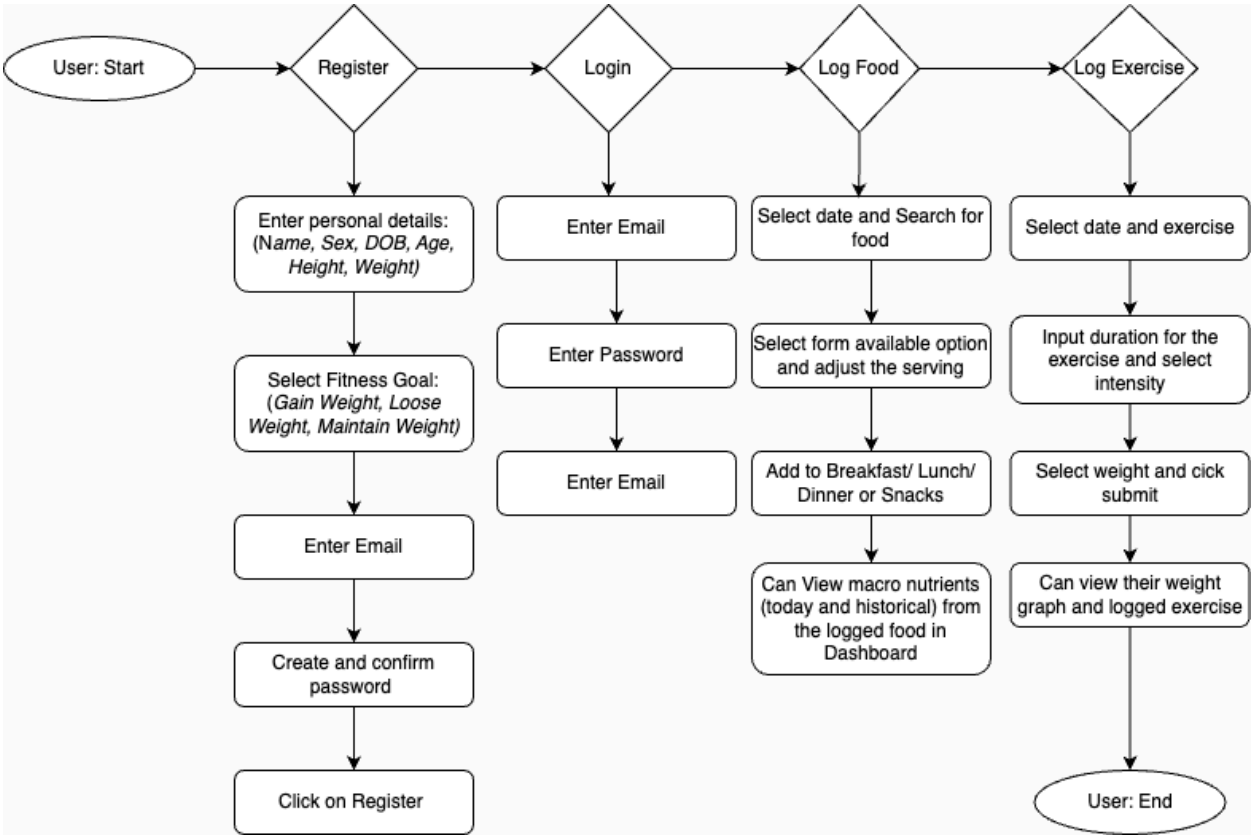


2. **Admin Role Case Diagram:**
   Actors: *Admin*

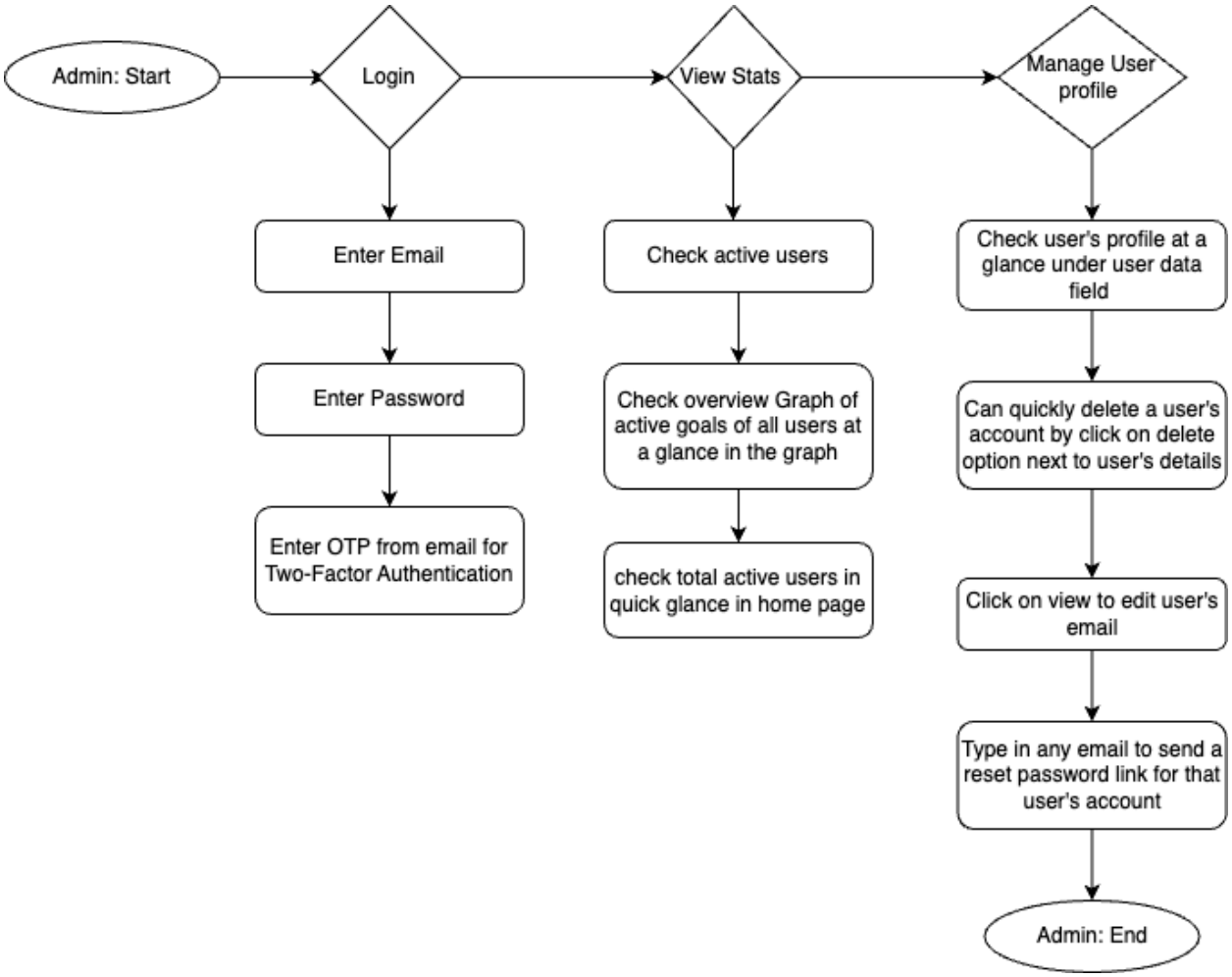


3. c) Activity diagrams for two most complex use cases:

Use case 1 : User role - User has to register for an account, Login to the account and can log their food to get the macronutrients consumed and log their exercise to keep track of their activities. Users can also view the data on the dashboard to see their overall performance to achieve their goal.



Use case 2 : Admin Role - Admin logs into their account securely using the 2FA, Can view the stats of the overall app (*like total active users, active goals the user's have assigned and also the distribution of the goal among users in a graph to get an idea of how many users are using the app for what goal),* admin can manage user profile under user data field, Can delete user's profile, update user's email on the account and can reset their account password.



3. d) Software qualities:

1. **Correctness:**
    - **User Role**
        - When a user logs their meals, the caloric value is retrieved and added to the daily total correctly.
        - Progress tracking accurately reflects weight changes based on weigh-in data.
    - **Admin Role:**
        - Admins receive accurate user analytics.
        - Admins can assist users with account issues, such as changing their password
2. **Time Efficiency:**
    - **User Role:**
        - Calories burned calculations happen automatically in the background, minimizing the need for manual input.
        - Meal logging has item suggestions, reducing the time spent on input.
    - **Admin Role:**
        - Reports for system wide analytics are generated on user data without delay.
        - Recommendations are sent automatically to multiple users at once.

3. **Robustness:**

- **User Role:**
    - The System prevents users from entering invalid data (e.g., negative weight values, invalid dates).
- **Admin Role:**
    - Data validation prevents the admin from accidentally deleting user information or generating incorrect reports.
    - Admin panels are protected against unauthorized access through extra security features. (such as frequent password change requirements compared to user profile, Stricter password requirements, Multi-Factor Authentication etc.)

4. Top-level and low-level software design

Covering the implementation of the Observer and Factory design patterns, explain their usability, and include class diagrams for each pattern and specifying the data types of attributes, method prototypes, and provide a class diagram of the entire system.

4. a) MERN Stack Multi-Layer Architecture

1. **<u>Presentation Layer</u>**

   Frontend Using React.js : UI Components, Responsive Design, State Management using React Hooks and Redux.

   Responsibility: Friendly User Interface, Interaction, Ease of Use, and Enjoyable Experience

   Features:

   - **User Registration Page:** New users have to register to access the application.
   - **User Login Page:** Registered users can log in.
   - **Home Page Dashboard:** Displays the logs of exercises and macronutrient intake. Also displays corresponding graphs of exercises and macronutrient logs.
   - **Log food page:** Users can search and track their meals for breakfast, snacks lunch, dinner, and snacks with macronutrient intake.
   - **Exercise Page:** Users can log their workouts with exercise type, duration, intensity, and weight options and calculate calories burned.
   - **User Profile page:** Users can see their profile information and be able to edit it.

2. **Application Layer**

   Backend Using Node.js and Express.js: Node.js for server logic and Express.js for routing.

   Responsibility: Manage business logic, API routing, and external APIs interaction.

   Features:

   - **User management:** User registration, authentication and profile updates.
   - **Macronutrients & Exercise Calculations:** Tracks macronutrients, calorie intake, and workout progress.
   - **Admin Features:** Edit and delete users.
   - **External Integration:** Use Edamam API for syncing nutritional data.

3. **Database Layer**

   Database using MongoDB: Database storage to store user data, exercise and macronutrient logs, and account information.

   Responsibility: Manage and store application data.

   Features:

   - **User data:** Users will be able to store their personal information and progress.
   - **Logs:** Users will be able to log their workouts and meals.

4. **External API/Integration Layer**

   Responsibility: Integrate with third-party services for nutritional data.

   Features:

   - **Edamam API:** This external API will be used to log meals and display macronutrient information.

Three Benefits

1. **Clear Separation of Responsibilities**

   Each part of the system will be responsible for doing specific and individual tasks. For example, the frontend will handle what the user will interact with, which includes registering for an account, logging their exercises and meals, and editing their profile information. The backend will handle the business logic, including storing and fetching user information, storing and fetching macronutrient information, and storing exercise workouts. The separation of these two responsibilities will make the code easier to handle and update if necessary. This will also make it easier for individual group members to tackle a different task and be able to work on each task separately.

2. **Easy to Grow and Add New Features**

   With the multilayer architecture that we choose we will also be able to easily add new functions/features. We can integrate a new feature like a different tracking tool for our exercise page without having to change the entire app. We can simply integrate this new tracking tool on our backend without having to change the front end part of the code. If we also had the need of adding more users because our application became popular, we will be able to add more users and data by simply adding more servers without needing to redesign the whole system.

3. **More Secure**

   With the multilayer architecture we choose our application is also more secure and safer to use. The app is safer to use because important information like user data and

user information is managed on the backend instead of the frontend. Each part of our system also works independently, which makes our app even more reliable since there isn't a single point of failure.

4. b) Observer and Factory design patterns

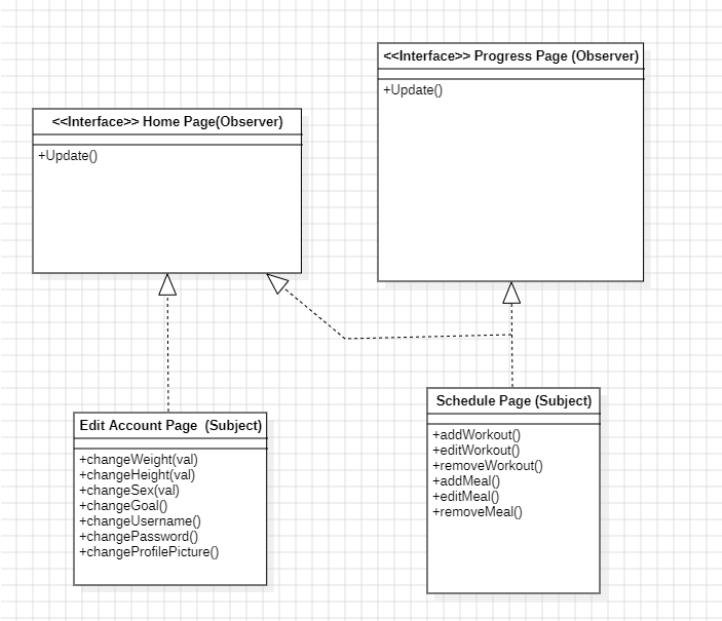
Observer Pattern for the User Role

Daily Summary Updates: The app's daily summary (e.g., calories consumed, workouts done, progress toward goals) could serve as an example. The user interacts with the app by logging meals or workouts. Each time data is logged, the daily summary gets updated accordingly.

- Subject: User's meal and workout logs
- Observer: Daily summary and progress dashboard

1. User logs a workout or meal (Subject)
2. App is notified (Observer)
3. App updates daily summary and progress stats (Observers)

Class Diagram for the Observer Design Pattern



Smart Device Syncing (for future implementation): The user's smart device can also be viewed as a subject that automatically syncs data like steps or calories to the app. The app acts as an observer, waiting for changes in the device's data. Whenever the smart device updates fitness information (e.g., steps, calories burned), it notifies the app, which automatically updates the user’s profile in response.

- Subject: Smart device (steps, calories burned)
- Observer: Fitness app (listening for changes to update user's profile, stats, progress graphs, etc.)

Observer Pattern for the Admin Role

Monitoring: Admins may monitor users based on data. In this case, users act as subjects when their data (e.g., weight, activity levels) is updated. The admin system becomes an observer, analyzing users updates.

- **Subject**: User data (weight change, activity level)
- **Observer**: Admin systems monitoring user data.

Account Management: When users are having technical issues, the admin dashboard can be used, allowing real-time support and assistance.

- Subject: User requests support
- Observer: Admin systems assist user

Factory Design Pattern

Usability: The overall intention of this design pattern is to outline a flexible framework that creates different objects. For example, our fitness app could utilize a class that creates user objects (namely, users and admin), and a static method that calls a class constructor (such as creating a fitness log or nutrition log). The object creation could then be handled in different subclasses, where each subclass represents a family.

Components of the factory design pattern:

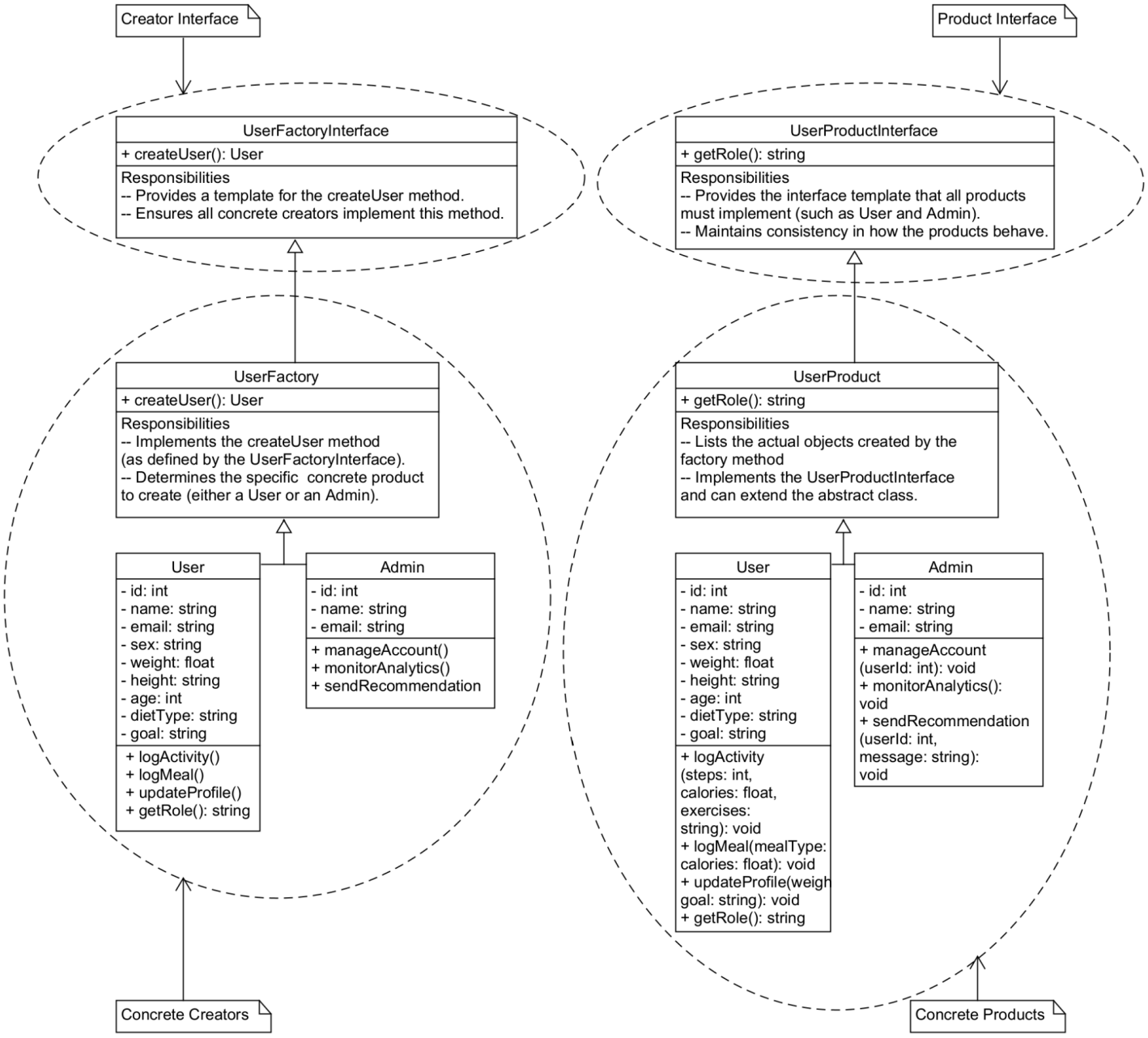
- **Creator Interface:** This is an abstract class that defines the factory method in charge of generating objects. It provides a template for how the methods should be implemented. In our factory class diagram, this is represented by the UserFactory Interface.
- **Concrete Creator:** Concrete Creators inherit from the abstract product. They are the subclasses that implement the factory method. They also provide the framework to build

and return objects based on the requirements. In our factory class diagram, this is represented by the UserFactory Concrete Creator.

- **Product Interface:** This defines the abstract class representing the objects produced by the factory method. All of the concrete products follow this common build, ensuring consistency in the objects created. In our factory class diagram, this is represented by the UserProduct Interface.
- **Concrete Product:** Concrete Products are the actual objects created by the factory method. Each Concrete Product can extend the product abstract class. In our factory class diagram, this is represented by the UserProduct Concrete Product.

Implementation of the Factory Design Pattern: Consider our software application, which must handle the creation of various user roles, such as the user entity and the admin entity. Each entity has its own specific properties and attributes.

Class Diagram for the Factory Design Pattern



Data types of the attributes:

Class: User

Attributes:

- id: int – Unique identifier.

- name: string – User's name.
- email: string – User's email.
- sex: string – User's sex.
- weight: float – User's weight in kilograms or pounds.
- height: string – User's height in cm or ft/in.
- age: int – User's age.
- dietType: string – User's diet type.
- goal: string – User's fitness goal.

Class: Admin

Attributes:

- id: int – Unique identifier.
- name: string – Admin's name.
- email: string – Admin's email.

Prototypes of the methods:

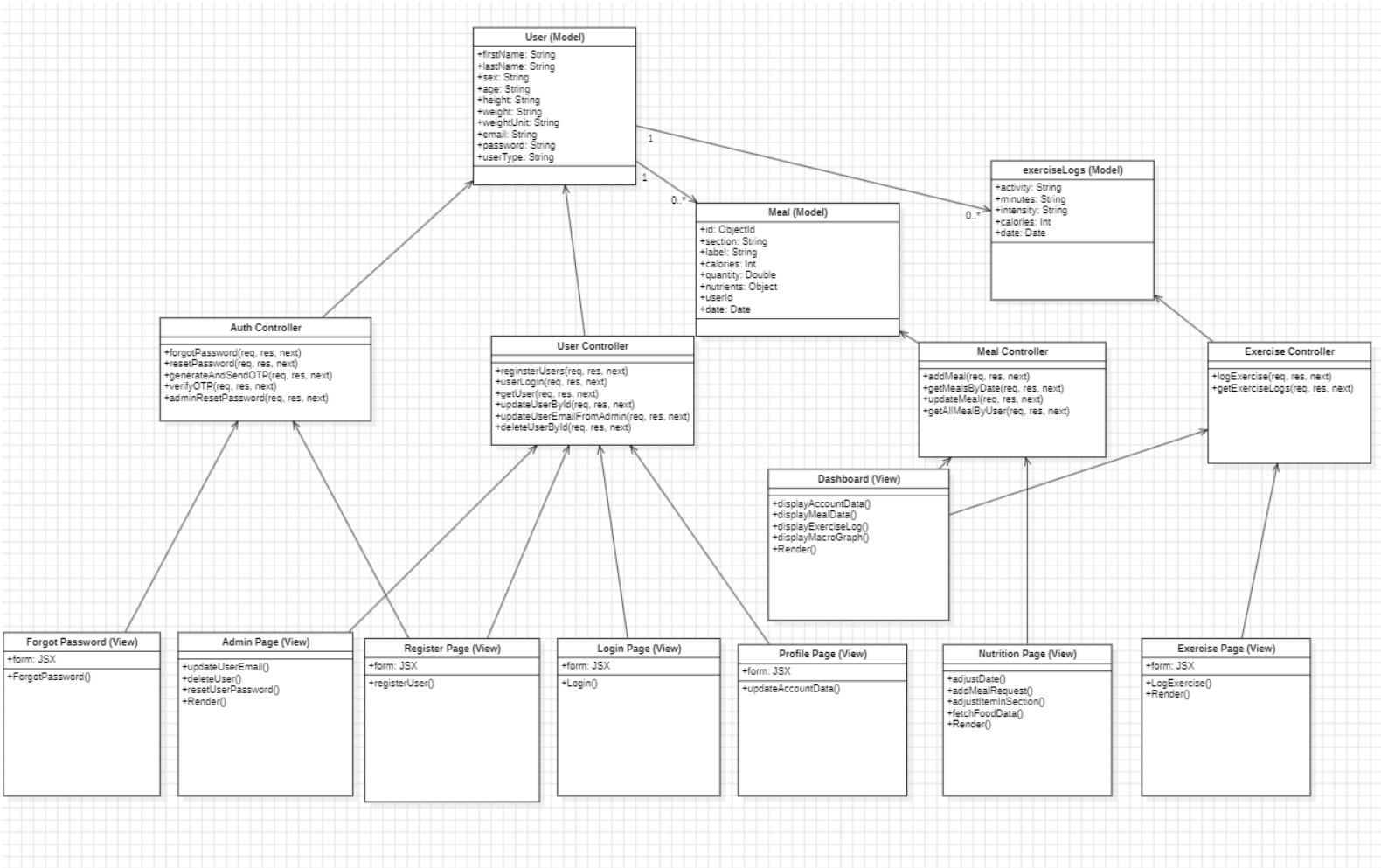
Class: User

```
public function logActivity(int $steps, float $calories, string $exercises): void
public function logMeal(string $mealType, float $calories): void
public function updateProfile(float $weight, string $goal): void
public function getRole(): string
```

Class: Admin

```
public function manageUserAccount(int $userId): void
public function monitorAnalytics(): void
public function sendRecommendation(int $userId, string $message): void
public function getRole(): string
```

4. c) Class diagram of the whole system by incorporating the implemented design patterns:



5. Software construction

5. a) Entire code for the design patterns:

Originally, our project's design approach was to implement the Factory and Observer design patterns to accomplish code reusability and modularity. As the project progressed, we identified the need to transition from this object-oriented approach in order to line up with the modern Model-View-Controller (MVC) architecture. The MERN stack approach that we used for development fits well under the multi-layered MVC architecture, enabling us to have an observable distinction between the backend and frontend components and responsibilities.

Model: For the Model layer, we used MongoDB for backend modeling and data storage, creating methods that defined relationships between entities such as users, meal logs, and exercise logs.

View: The frontend, using React.js, serves as the View layer. This allows for dynamically displaying user interfaces. This approach minimized the need for manual updates typical of the Observer pattern.

Controller: The Express.js server is used as the Controller, dividing code into User Controllers and Admin Controllers as well as routing using User Routes and Admin Routes. This simplified the code structure for managing requests, as compared to a Factory method for object creation.

This transition allowed for a more efficient development process. The result was a streamlined MERN system that is maintainable, scalable, and in line with the functional requirements of our app. Implementing MVC architecture also ensures our system remains in line with modern industry standards and best practices.

The following pages will contain the code for both the original implementations of the Observer and Factory design plans, followed by the code using MVC architecture. There will be a few examples of code incorporating the design patterns, however the remaining code can be viewed using the github link posted below.

Code for the Observer and Factory Design Patterns

```php
require_once 'User.php';
require_once 'Admin.php';
require_once 'UserFactory.php';

try {
    // Create a user
    $user = UserFactory::createUser(
        'user', 'John Smith', 'john@fakemail.com', 'Male', 190, '5 ft 11 inch', 25, 'High protein', 'Lose 10lbs'
    );
    echo $user->getRole() . " created successfully.\n";
    $user->logActivity(3000, 250, 'Stair Master');
    $user->logMeal('Lunch', 550);
    $user->updateProfile(180, 'Maintain Weight');

    // Create an admin
    $admin = UserFactory::createUser('admin', 'Ben Stone', 'admin@fakemail.com');
    echo $admin->getRole() . " created successfully.\n";
    $admin->monitorAnalytics();
    $admin->sendRecommendation('8675309', 'Excellent progress, keep it up!');
} catch (Exception $e) {
    echo "Error: " . $e->getMessage();
}
```

```php
class User {
    private $id, $name, $email, $sex, $weight, $height, $age, $dietType, $goal;

    public function __construct($name, $email, $sex, $weight,
                                $height, $age, $dietType, $goal,) {
        $this->id = $id;
        $this->name = $name;
        $this->email = $email;
        $this->sex = $sex;
        $this->weight = $weight;
        $this->height = $height;
        $this->age = $age;
        $this->dietType = $dietType;
        $this->goal = $goal;
    }

    public function logActivity($steps, $calories, $exercises) {
        // Log user workout activities manually or with smart device.
    }

    public function logMeal($mealType, $calories) {
        // Log meals manually or via an integrated API.
    }

    public function updateProfile($weight, $goal) {
        // Update profile information.
        $this->weight = $weight;
        $this->goal = $goal;
        echo "Profile updated";
    }

    public function getRole() {
        return "User";
    }
}
```

```php
class Admin {
    private $id, $name, $email;

    public function __construct($name, $email) {
        $this->id = $id;
        $this->name = $name;
        $this->email = $email;
    }

    public function manageUserAccount($userId) {
        // Admin can reset password or remove an account.
        echo "Managed account of User ID: $userId.";
    }

    public function monitorAnalytics() {
        // Admin monitors app analytics.
    }

    public function sendRecommendation($userId, $message) {
        // Admin can send personalized recommendations
        echo "Recommendation send to User ID: $userId.";
    }

    public function getRole() {
        return "Admin";
    }
}


class UserFactory {
    public static function createUser($role, $name, $email, $sex = null,
                                      $weight = null, $height = null, $age = null,
                                      $dietType = null, $goal = null,) {
        switch (strtolower($role)) {
            case 'user':
                return new User($name, $email, $sex, $weight, $height,
                                $age, $dietType, $goal);
            case 'admin':
                return new Admin($name, $email);
            default:
                throw new Exception("Invalid user role: $role");
        }
    }
}
```

Code using Model-View-Controller (MVC) architecture:

Model:

database.js

```
var MongoClient = require('mongodb').MongoClient;
require('dotenv').config();
let { mongoUrl } = process.env;
// Connection URL
var url = mongoUrl;
exports.connection = () => {
    return new Promise(async (resolve, reject) => {
        await MongoClient.connect(url, { useNewUrlParser: true,
useUnifiedTopology: true }, (err, client) => {
            if (err) console.log("error", err);
            console.log("Connected correctly to database");
            // resolve(db.db("Optimum"));
            var db = client.db('FitnessApp');
            resolve(db);
        });
    })
}
// Use connect method to connect to the database
```

APIError.js

```
class ExtendableError extends Error {
    constructor(message, status, isPublic) {
        super(message);
        this.name = this.constructor.name;
        this.message = message;
        this.status = status;
        this.isPublic = isPublic;
        Error.captureStackTrace(this, this.constructor);
    }
}  // API Error handling

/**
 * Class representing an API error.
 * @extends ExtendableError
 */
class APIError extends ExtendableError {
```

```
    /**
     * Creates an API error.
     * @param {string} message - Error message.
     * @param {number} status - HTTP status code of error.
     * @param {boolean} isPublic - Whether the message should be visible
to user or not.
     */
    constructor(message, status = httpStatus.INTERNAL_SERVER_ERROR,
isPublic = false) {
        super(message, status, isPublic);
    } // API error Handling
}
```

query.js

```
exports.findOne = (collection, query, additionalParameter) => {
    return new Promise((resolve, reject) => {
        //
console.log("additionalParameter",additionalParameter,query,collection);
        additionalParameter == undefined ?
            collection.findOne(query, (err, queryResult) => {
                if (err) {
                    return reject({ message: "DB query Failed" });
                } else {
                    // console.log("queryResult",queryResult)
                    resolve(queryResult);
                }
            }) : collection.findOne(query, additionalParameter, (err,
queryResult) => {
                if (err) {
                    return reject({ message: "DB query Failed" });
                } else {
                    // console.log("queryResult",queryResult)
                    resolve(queryResult);
                }
            })
    })
}

exports.insertMany = (collection, query) => {
    // query.createdAt= moment().utc().format();
```

```
    return new Promise((resolve, reject) => {
        collection.insertMany(query, (err, recordSaved) => {
            if (recordSaved) {
                // console.log("recordSaved",recordSaved)
                resolve(recordSaved)
            } else {
                console.log("err", err)
                reject(err)
            }
            // err ? reject(err) : resolve(recordSaved)
        })
    })
}

exports.deleteOne = (collection, query) => {
    return new Promise((resolve, reject) => {
        collection.deleteOne(query, (err, deletedRecords) => {
            if (err) {
                reject({ message: "DB query Failed" });
            } else {
                resolve(deletedRecords)
            }
            // err ? reject(err) : resolve(deletedRecords)
        })
    })
}
```

View:

Login.js

```
function Login() {
    const [show, setShow] = useState(false)
    const [email, setEmail] = useState('')
    const [password, setPassword] = useState('')
    const [emailError, setEmailError] = useState('')
    const [passwordError, setPasswordError] = useState('')
    const toast = useToast()
    const navigate = useNavigate()
    const toastIdRef = useRef()
```

```
    const handleClick = () => setShow(!show)

    const validateEmail = (email) => {
        const re = /^[^\s@]+@[^\s@]+\.[^\s@]+$/
        if (!email) {
            setEmailError('Email is required')
            return false
        }
        if (!re.test(email)) {
            setEmailError('Invalid email format')
            return false
        }
        setEmailError('')
        return true
    }
```

Exercise.js

```
function Exercise() {
  const [formData, setFormData] = useState({
    exercise: '',
    duration: '',
    intensity: '',
    weight: '',
    unit: 'lbs' //Default is pounds
  });
  const [selectedDate, setSelectedDate] = useState(new Date());
  const [message, setMessage] = useState('');

  const token = localStorage.getItem('token');
  const userId = jwtDecode(token)._id;

  const handleChange = (event) => {
    const { name, value } = event.target;
    setFormData((prevData) => ({
      ...prevData,
      [name]: value
    }));
  };
```

NutritionForm.js

```
 // Update meal whenever meals changes (happens on different day or
whenever meals are added/adjusted)
  useEffect(() => {

    if (meals.length > 0) {
      // Populate section items based on fetched meal data
      const breakfast = meals.filter(meal => meal.section ===
'Breakfast');
      const lunch = meals.filter(meal => meal.section === 'Lunch');
      const dinner = meals.filter(meal => meal.section === 'Dinner');
      const snacks = meals.filter(meal => meal.section === 'Snacks');

      setBreakfastItems(breakfast);
      setLunchItems(lunch);
      setDinnerItems(dinner);
      setSnackItems(snacks);
    }
    else{
      setBreakfastItems([]);
      setLunchItems([]);
      setDinnerItems([]);
      setSnackItems([]);
    }
  }, [meals]);
```

Dashboard.js

```
function Dashboard() {
    const [profile, setProfile] = useState(null);
    const [loading, setLoading] = useState(true);
    const [macroNutrients, setMacroNutrients] = useState([]);
    const [historicalMacros, setHistoricalMacros] = useState([]);
    const [exerciseLogs, setExerciseLogs] = useState([]);
    const [currentPage, setCurrentPage] = useState(1);
    const [unit, setUnit] = useState({ height: 'cm', weight: 'kg' });
    const itemsPerPage = 10;
    const token = localStorage.getItem('token');
    const decodedToken = jwtDecode(token);
    const userId = decodedToken._id;
```

```
    const today = new Date().toISOString().split('T')[0];

    const [dailyChartOptions, setDailyChartOptions] = useState({
        autoSize: true,
        title: { text: "Daily Macronutrient Distribution" },
        data: [],
        series: [
            {
                type: "pie",
                angleKey: "value",
                calloutLabelKey: "name",
                sectorLabelKey: "value",
                fills: [
                    'rgba(54, 162, 235, 0.6)', // Protein
                    'rgba(255, 206, 86, 0.6)', // Carbs
                    'rgba(255, 99, 132, 0.6)', // Fats
                ],
                strokes: [
                    'rgba(54, 162, 235, 1)', // Protein
                    'rgba(255, 206, 86, 1)', // Carbs
                    'rgba(255, 99, 132, 1)', // Fats
                ],
            },
        ],
    });
```

Controllers:

auth.controllers.js

```
const resetPassword = async (req, res, next) => {
    try {
        const reqData = req.body
        let userData = await query.findOne(userColl, { email:
reqData.email });

        if (!userData || userData.password == null) {
            const message = `Incorrect email or password.`;
            return next(new APIError(`${message}`, httpStatus.BAD_REQUEST,
true));
        }
```

```
        const isValidate = validPassword(userData.password, reqData.code)
        if (isValidate) {
            const encryptPass = generatePassword(reqData.newPassword);

            let updateUserData = await query.findOneAndUpdate(userColl, {
email: reqData.email }, { $set: { password: encryptPass } });
            let obj = resPattern.successPattern(httpStatus.OK,
updateUserData.ops, 'success');
            return res.status(obj.code).json(obj)
        } else {
            let obj = resPattern.errorPattern(httpStatus.BAD_REQUEST, 'The
Code is no longer valid');
            return res.status(obj.code).json(obj)
        }
    } catch (e) {
        return next(new APIError(`${e.message}`, httpStatus.BAD_REQUEST,
true));
    }
}
```

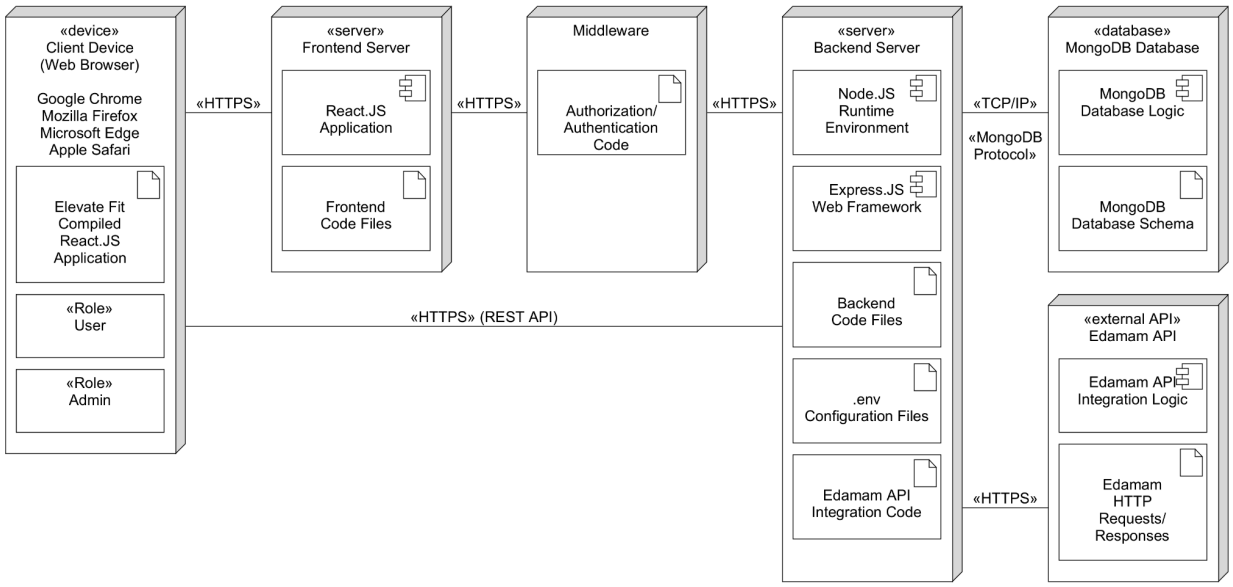
meal.controllers.js

```
// Add a new meal for a specific user
const addMeal = async (req, res, next) => {
    try {
        const userId = req.params.userId; // User ID is passed in request
parameters
        const mealData = req.body;

        mealData.userId = ObjectId(userId); // Add userId to meal data
        mealData.date = req.query.date; // Assign querie'd date to meal

        const insertResponse = await mealsColl.insertOne(mealData);
        let obj = resPattern.successPattern(httpStatus.OK,
insertResponse.ops[0], 'Meal added successfully');
        return res.status(obj.code).json(obj);
    } catch (e) {
        return next(new APIError(`${e.message}`, httpStatus.BAD_REQUEST,
true));
    }
};
```

user.controllers.js

```
const userLogin = async (req, res, next) => {
    try {
        const { password } = req.body;
        const reqData = { email: req.body.email }
        let userData = await query.findOne(userColl, reqData);
        if (!userData || userData.password == null) {
            const message = `Incorrect email or password.`;
            return next(new APIError(`${message}`, httpStatus.BAD_REQUEST,
true));
        }
        const isMatch = validPassword(userData.password, password)
        if (isMatch) {
            const token = jwt.sign({ _id: userData._id, email:
userData.email, userType: userData.userType }, process.env.JWT_SECRET)
            delete userData.password
            let obj = resPattern.successPattern(httpStatus.OK, { userData,
token }, 'success');
            return res.status(obj.code).json(obj)
        } else {
            const message = `Incorrect email or password.`;
            return next(new APIError(`${message}`, httpStatus.BAD_REQUEST,
true));
        }
    } catch (e) {
        return next(new APIError(`${e.message}`, httpStatus.BAD_REQUEST,
true));
    }
}
```

b) Deployment diagram regarding the hardware configuration of the code



5. c) GitHub link of the entire program:

https://github.com/1samzi/fitness-app

5. d) Code Quality : Error handling, Unit tests, and Best Practices

Error Handling: We used error handling in our project to make sure that users receive the right feedback in case something goes wrong. Wrong input or invalid characters will give direct messages to the user so they know the steps they need to take to fix the mistakes they've made.

Form Validation: In our RegsitrationForm component, we validate the email field to ensure it had a correct format, and the password field was also checked to ensure that it met specific requirements. If there is wrong input or invalid characters, an error message would pop up right next to that field.

```
•••
    const nameRegex = /^[a-zA-Z]{2,30}$/
    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/
    const passwordRegex = /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$/
•••
        return nameRegex.test(value) ? '' : 'Must be 2-30 characters long and contain only letters.'
      case 'email':
        return emailRegex.test(value) ? '' : 'Please enter a valid email address.'
      case 'password':
        return passwordRegex.test(value)? ''
          : 'Password must be at least 8 characters long and include uppercase, lowercase, number, and special character.'
      case 'confirmPassword':
        return value === formData.password ? '' : 'Passwords do not match.'
•••
```

Network Errors: The handleSubmit function in our TwoFactorAuth.js component, we used a try-catch block to handle possible errors when making API requests. If a network failure occurs, the user is notified with an error message.

```
try {
    const response = await fetch('http://localhost:3001/api/auth/verify-otp', {
•••
if (response.ok) {
      const data = await response.json()
      toast({
        title: "Authentication successful",
        description: "You have been successfully authenticated.",
        status: "success",
        duration: 5000,
        isClosable: true,
•••
else {
        const errorData = await response.json()
      throw new Error(errorData.message || 'Authentication failed')
    }
  } catch (error) {
    toast({
      title: "Authentication failed",
      description: error instanceof Error ? error.message : "The code you entered is incorrect. Please try again.",
      status: "error",
      duration: 5000,
      isClosable: true,
```

Unit Testing

Form Validation Tests: We implemented unit tests on our functions validateField and validateForm. The unit tests for these functions did verify that they returned the correct error messages that we needed. We also tested with various different inputs to also ensure robustness for our form validation.

API Request Tests: We also implemented unit tests on our functions handleSubmit and handleResendCode in our TwoFactorAuth.js component. These functions are structured in a way to handle different server responses like success and failure. We used the Jest tool to try different fetch requests and the application returned the correct behaviour that we desired. Our first unit test we implemented a test to simulate a failed OTP verification in our TwoFactorAuth.js component and we were able to verify that the error message was correctly displayed.

Best Practices

State Management: In designing our code we used best practices with the use of React's useState management for managing form data and error messages for simplicity. This best practice ensured that our code was designed in a way that the user interface was always showing the current state of the form fields.

Flexibility: In designing our functions in all of our components we implemented our code to be flexible. For example in our handleUnitChange function, this was written to allow for the conversion of units, height and weight without the need of hardcoding specific logic for each unit type. With this flexible approach, we were able to reduce repetition and made the code easier to maintain.

Maintainability: In designing our code, we structured our validation logic in separate functions, which made it easier to update and extend more validation rules if needed.

6. Technical documentation

6. a) List of programming languages:

The programming languages used in our code are the following:

- **JavaScript**: Widely utilized in both frontend (React components) and backend (Node.js / express API routes and controllers).
- **JSON (JavaScript Object Notation):** Employed to define data in React's useState and for exchanging data with APIs (such as in fetch requests).
- **Framework:**
  -
  - **Chakra UI** - Widely used react frameworks/components library.
  - **Ag charts** - For better visualization and to implement a wide range of charts/graphs.
  - **Express -** This is a backend framework which is used to build on top of node.js that helps manage servers and routes.

6. b) List of reused algorithms and small programs:

The reused algorithms and small programs used in our code are the following:

- **API Response Formatting**

  Algorithm: Standardized format for API success and error responses.

  Reference: https://www.baeldung.com/rest-api-error-handling-best-practices

- **Custom Error Handling with APIError**

  Algorithm: Custom error generation and handling.

  Reference: https://www.baeldung.com/exception-handling-for-rest-with-spring

- **Database Query Wrappers**

  Algorithm: MongoDB CRUD operations via an abstraction layer.

  Reference: https://www.mongodb.com/docs/manual/crud/

- **JWT Token Generation and Validation**

  Algorithm: JWT (JsonWebToken) for token creation and verification.

  Reference: https://jwt.io/introduction/

- **React.js Library**

  useState Hook: Used to manage component state for formData, selectedDate, and messages.

  Reference: https://react.dev/reference/react/useState#usage

- **Chakra UI Library**

  Components used: Box, FormControl, FormLabel, Input, Select, Button, Text,Hstack, Vstack etc. for user interface.

  References: https://v2.chakra-ui.com/docs/components

- **React DatePicker Library**

  DatePicker Component: Reused the DatePicker from react-datepicker package to select dates on a calendar.

  References: https://reactdatepicker.com/

- **Password Hashing and Validation**

  Algorithm: Hashing passwords with bcrypt.

  Reference: https://www.npmjs.com/package/bcrypt

- **Password Reset Logic**

  Algorithm: Generating emails with a temporary password or OTP.

  Reference: https://www.nodemailer.com/

- **Unit Conversion (Height and Weight)**

  Algorithm: Conversion logic for height (inches to cm) and weight (kg to lbs).

  Reference: Own algorithm

- **User Input Validation (Email, Password, etc.)**

  Algorithm: Using regular expressions to validate email and password formats.

  Reference: Frontend validation

6. c) List of software tools and environments:

The software tools and environments used in our code are the following:

- **MongoDB**

  Benefit: As a NoSQL database, MongoDB is ideal for storing varied and unstructured data, like user profiles, exercise records, and food logs, without rigid constraints. This edocument-based model enables easy access and updates of user-specific data, which is useful for analytics and goal tracking.

- **Express**

  Benefit: Express provides efficient route and middleware management, facilitating

organized development of features like user authentication, exercise tracking, nutrition logging, and password recovery. It supports rapid prototyping and deployment.

- **React**

  Benefit: React is a library designed for creating interactive, responsive user interfaces. It simplifies creating complex UIs for our app, such as the dashboard for tracking user progress, forms for logging exercises and meals, and dynamic elements for visualizing data. React's design components also enhance reusability, ensuring consistency throughout the app.

- **Node.js**

  Benefit: Node.js offers a powerful runtime for producing scalable backend APIs, which is useful for handling real-time updates (like fitness progress tracking) and managing tasks efficiently (such as sending OTPs and processing login requests).

- **bcrypt**

  Benefit: Secures passwords by hashing them before storage, protecting credentials and preventing unauthorized access. Security is essential in our app, especially for metrics like health or dietary habits.

- **Body-Parser**

  Benefit: Efficiently parses incoming request data, especially JSON payloads (e.g., exercise or nutrition logs), allowing for consistent and secure handling of user input.

- **CORS**

  Benefit: Allows backend communication with a frontend hosted on a different origin, which is integral for web-based apps. This ensures smooth, secure communication across platforms.

- **Express-Session**

  Benefit: Manages user sessions on the server, ensuring continuous authentication across requests. This is critical for a secure, seamless experience in our app where users may frequently access sensitive personal information such as weight and fitness goals.

- **jsonwebtoken (JWT)**

  Benefit: Provides stateless authentication, ensuring users remain logged in without session storage which is also a benefit for cross-platform access. JWTs provide secure user identification with minimal overhead.

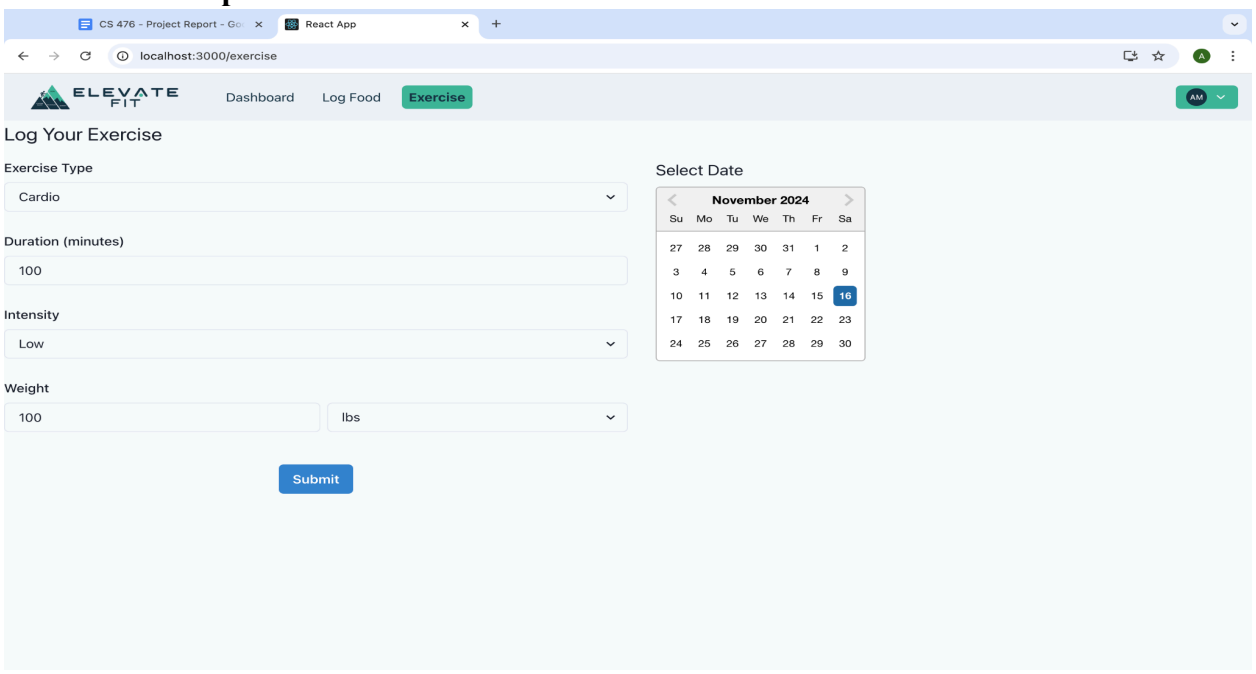
- **Moment.js**

  Benefit: Simplifies date and time handling, which is essential for tracking progress over different time periods. It aids in calculating time intervals (e.g., time since the last exercise logged) and formatting timestamps in user-friendly ways for exercise logs or nutrition logs.

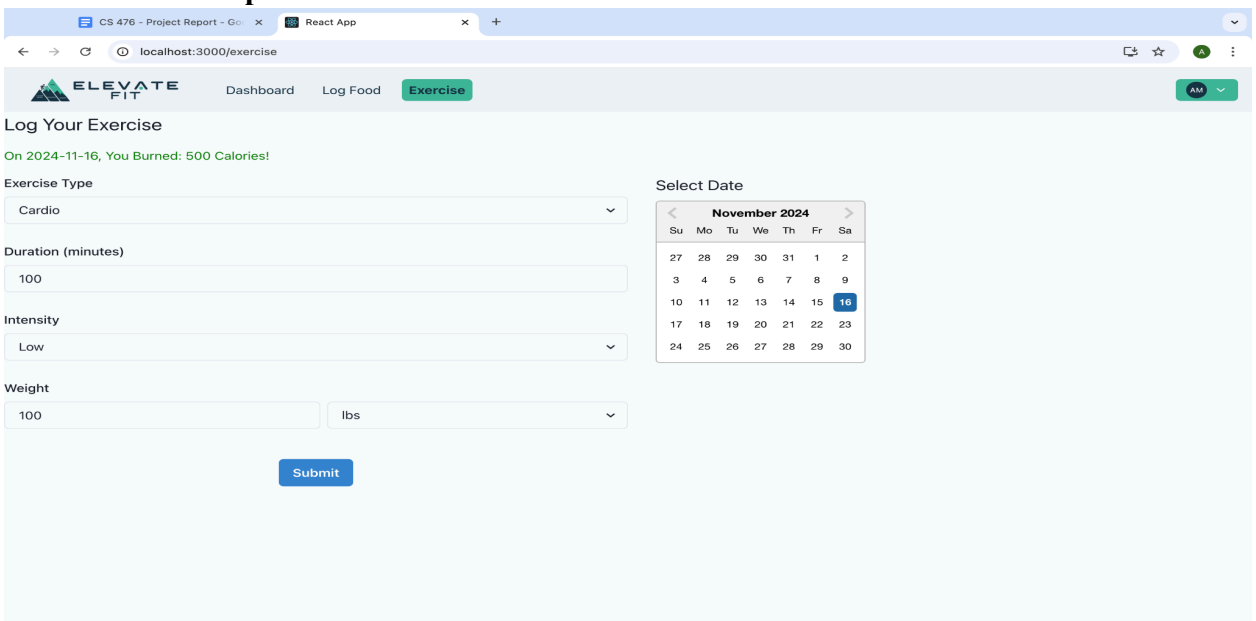
7. Acceptance testing

7. a) Correctness Testing

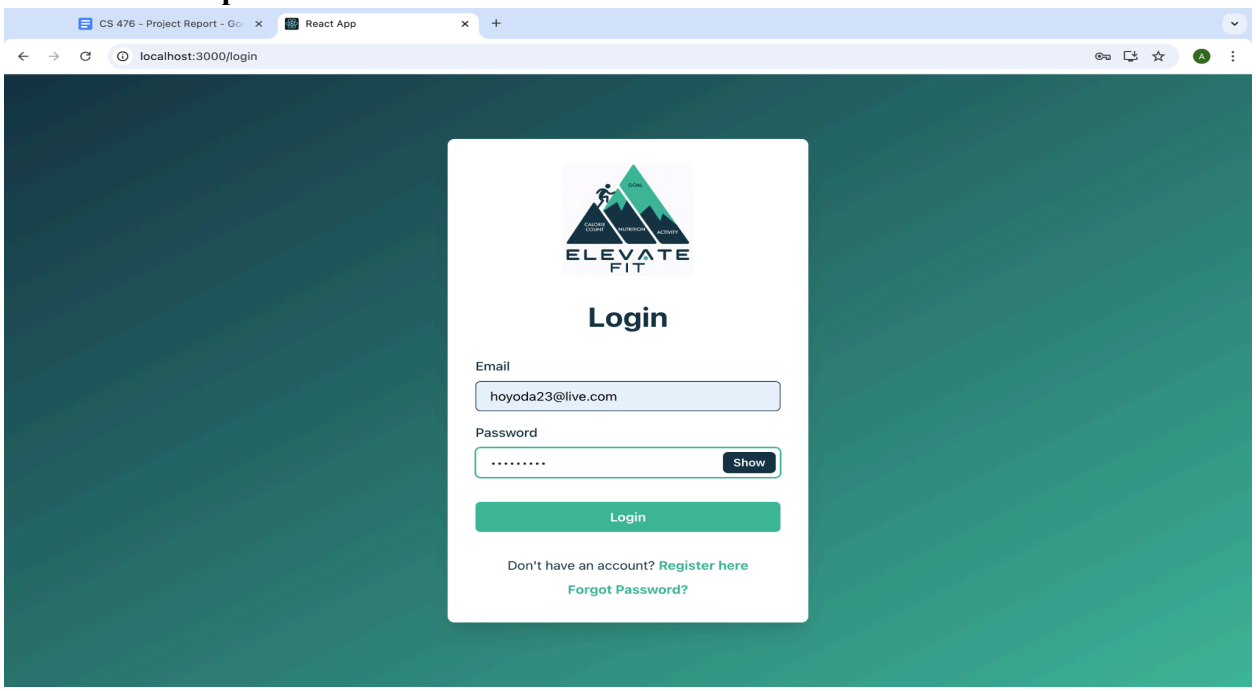
Test Case 1: Exercise Logging Form.
Input: Exercise type: Cardio. Duration: 100. Intensity: Low. Weight: 100
Select Date: November 14, 2024
Expected Output: On 11/16/2024, You Burned: "System Calculated Calories"
Screenshot of Input:



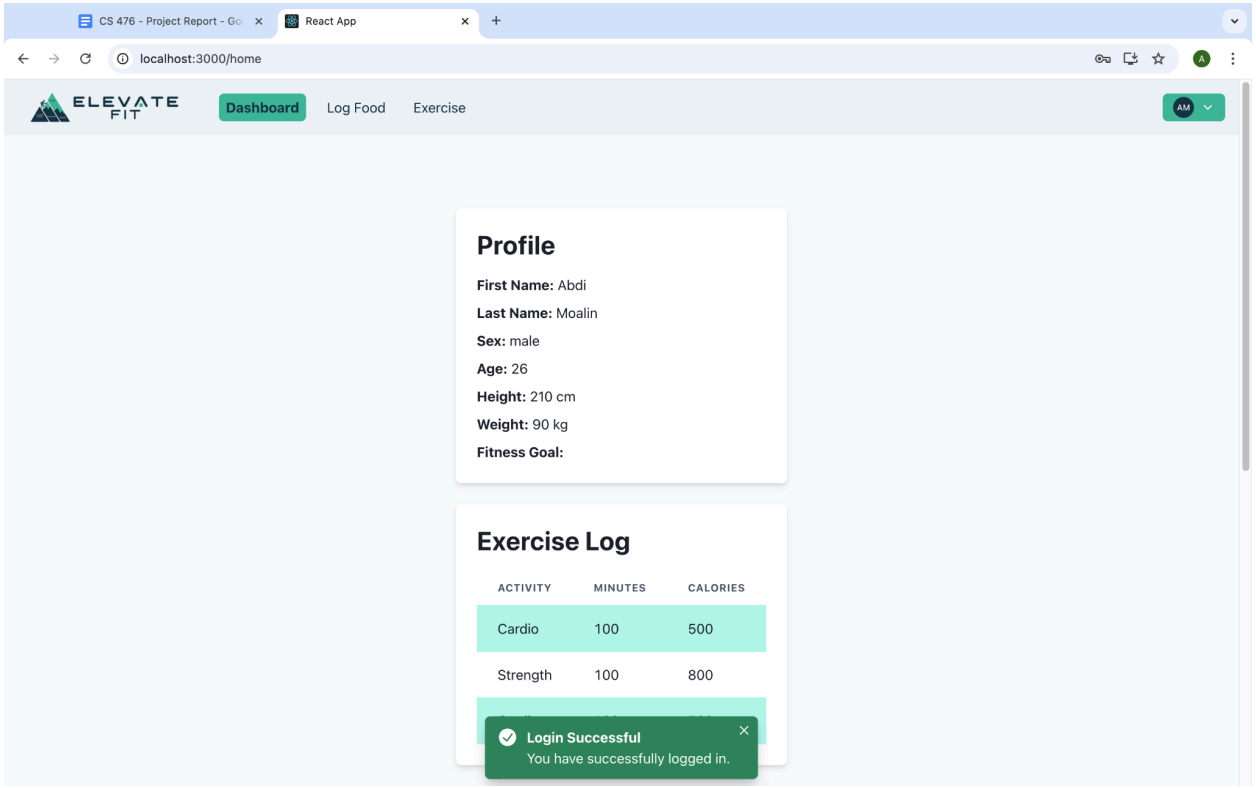
Screenshot of Output:



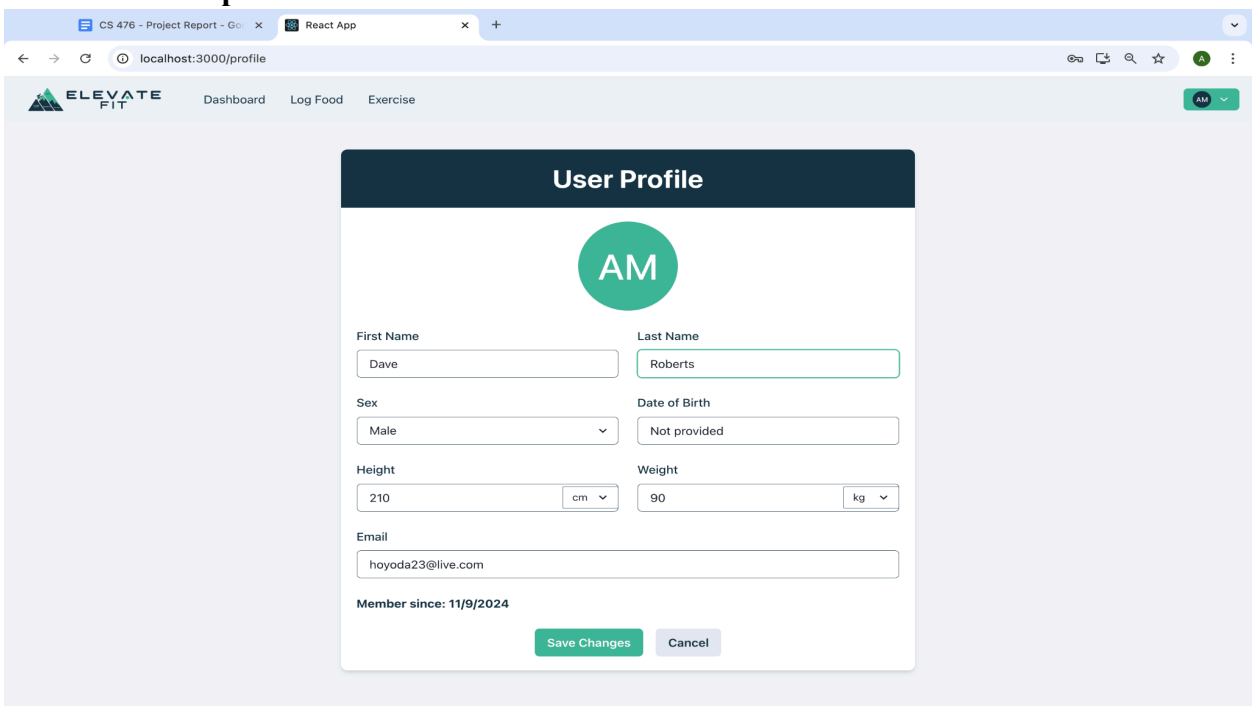
Test Case 2: Login Page
Input: Enter email. Enter Password. Press login.
Expected Output: Redirected to the dashboard home page with a success login message.
Screenshot of Input:



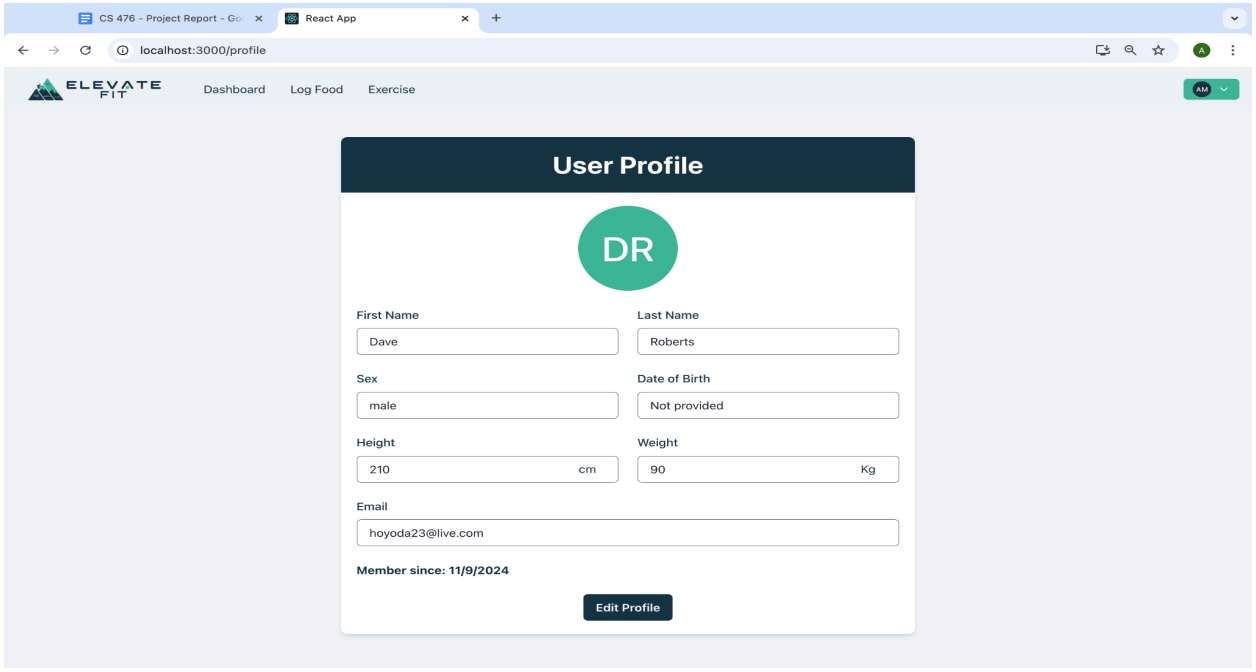
Screenshot of Output:



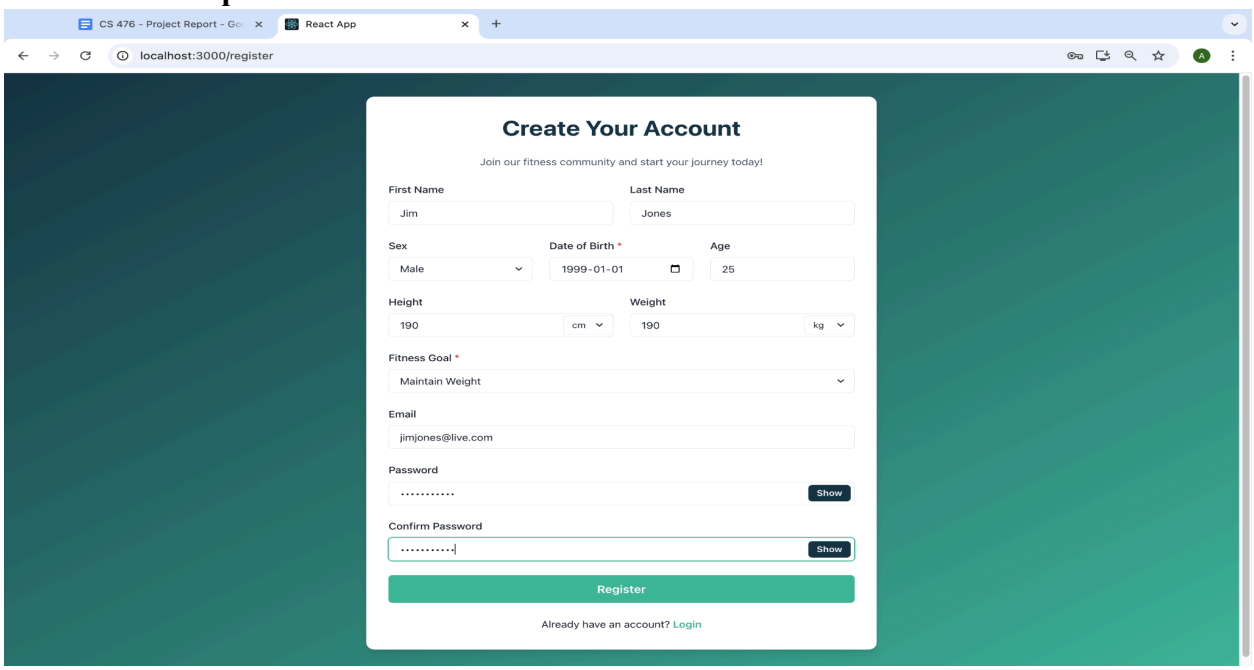
Test Case 3: Admin Edits User Profile.
Input: Edit Previous first name,last name, height, weight and press save.
Expected Output: Dave Roberts, with profile picture changed to DR.
Screenshot of Input:



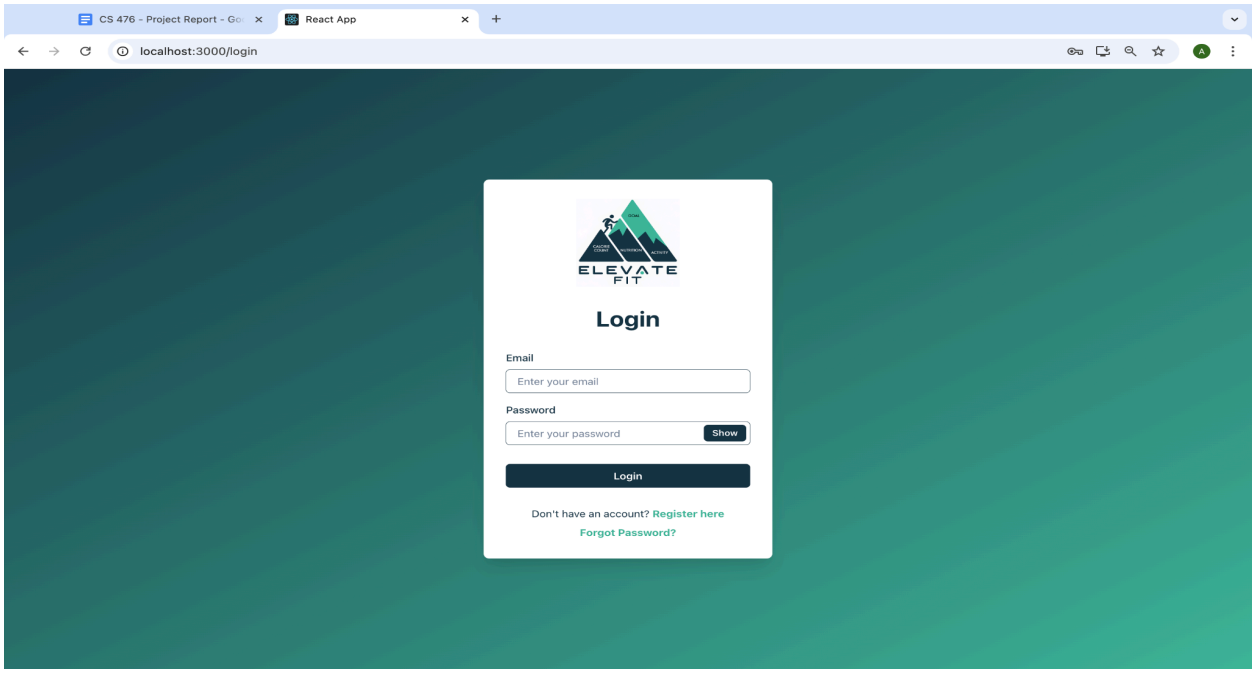
Screenshot of Output:



Test Case 4: Create Account
Input: Enter first name,last name, sex, DOB, Age, Height Weight & Fitness goal.
Enter Email and password. Enter Confirm Password. Press Register.
Expected Output: Redirected to the Login Page with a successful account created message.
Screenshot of Input:

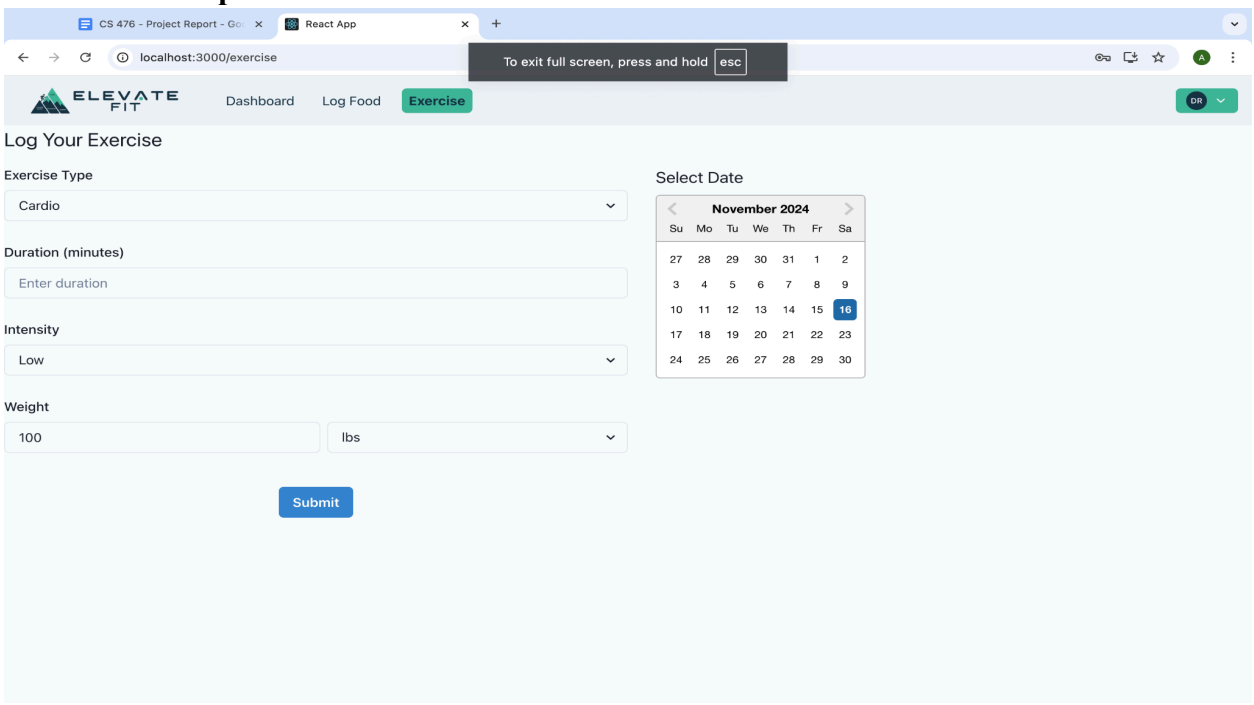


Screenshot of Output:

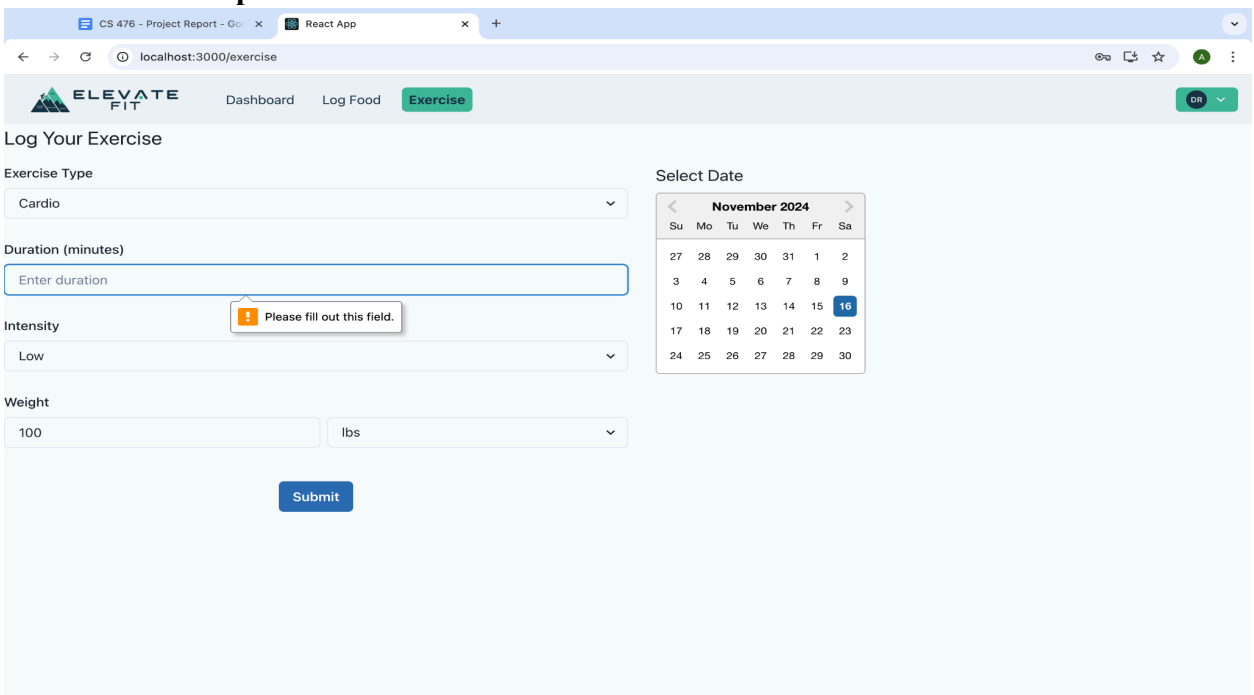


7. b) Robustness Testing

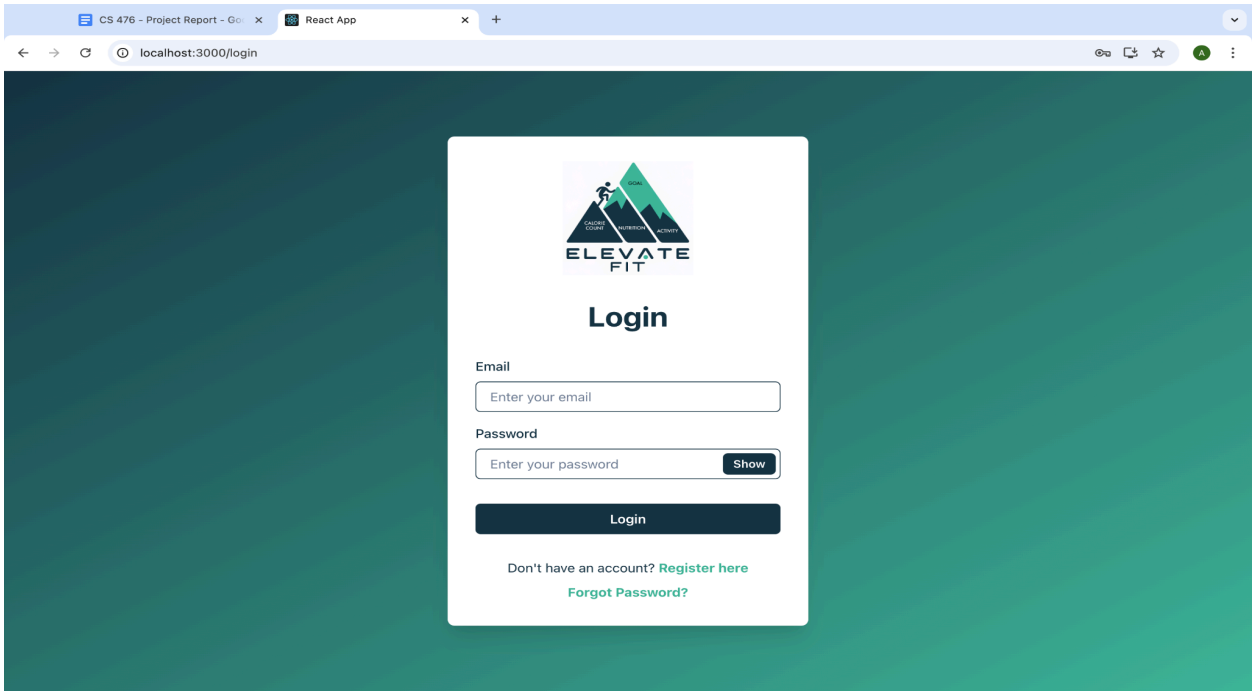
Test Case 1: Submit exercise logging form without entering duration of exercise.
Input: Leave duration field blank, and fill all other fields.
Expected Output: Error message “Please fill out this field”
Screenshot of input:



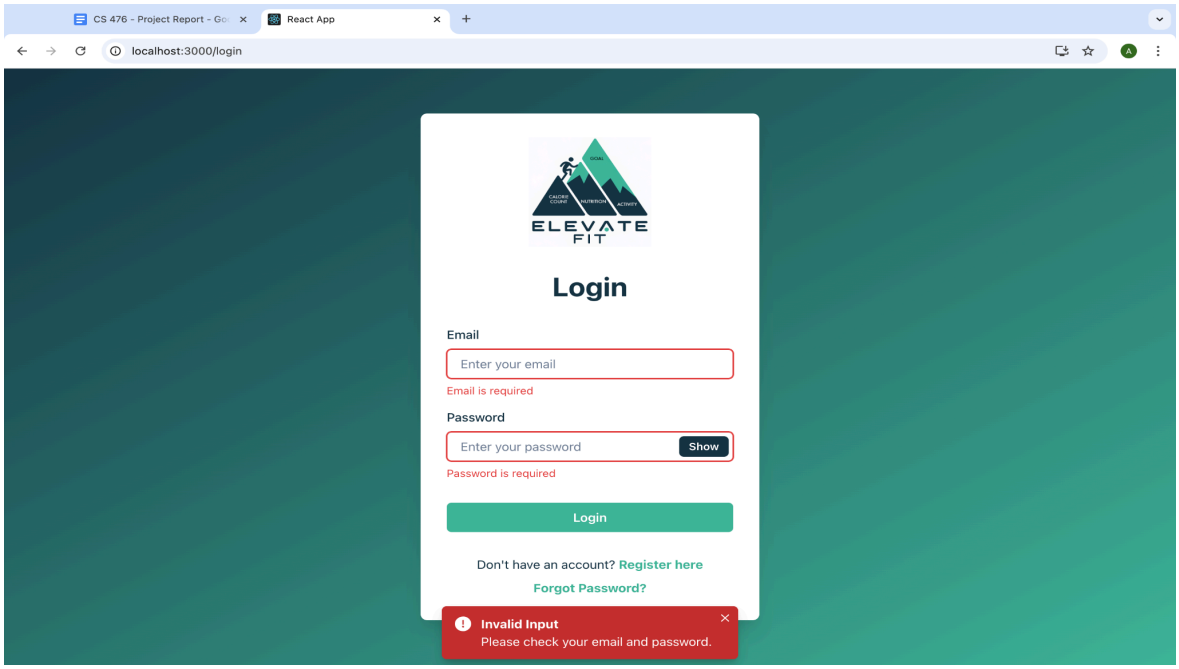
Screenshot of output:



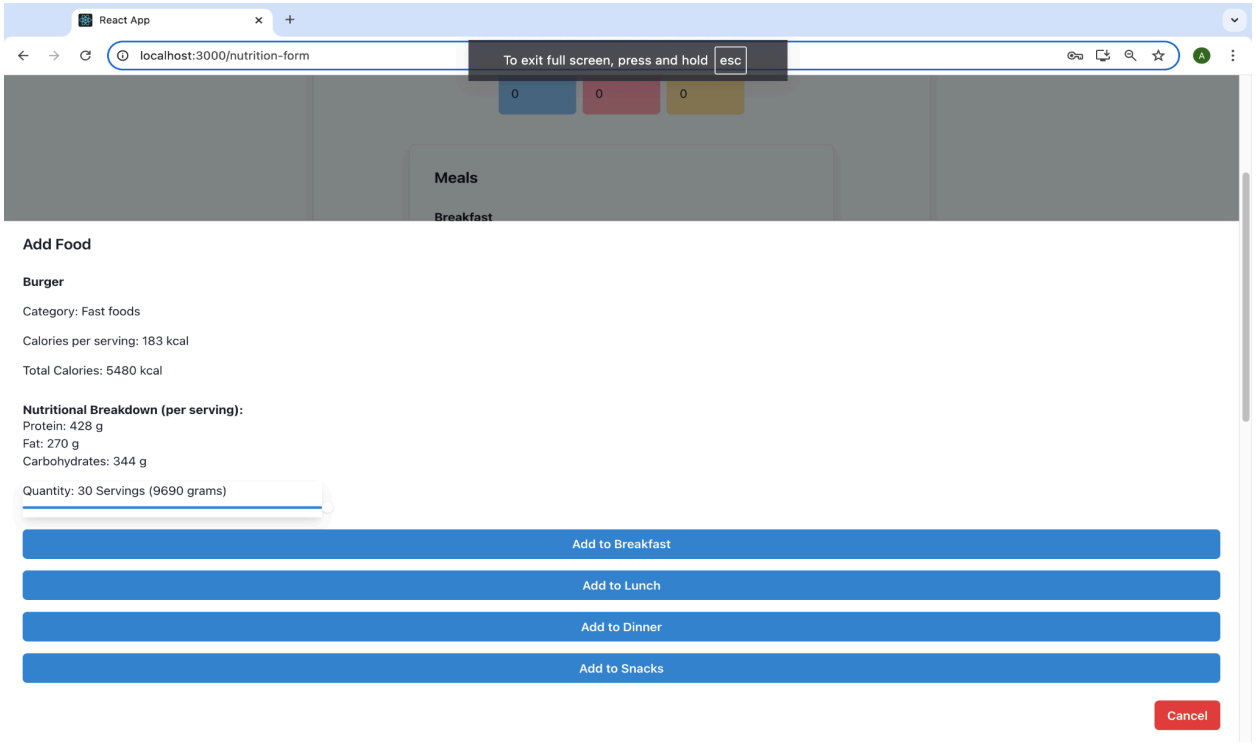
Test Case 2: Logging in without entering email or password
Input: Leave email and password field blank
Expected Output: Error message “Invalid Input, please check your email and password”
Screenshot of Input:



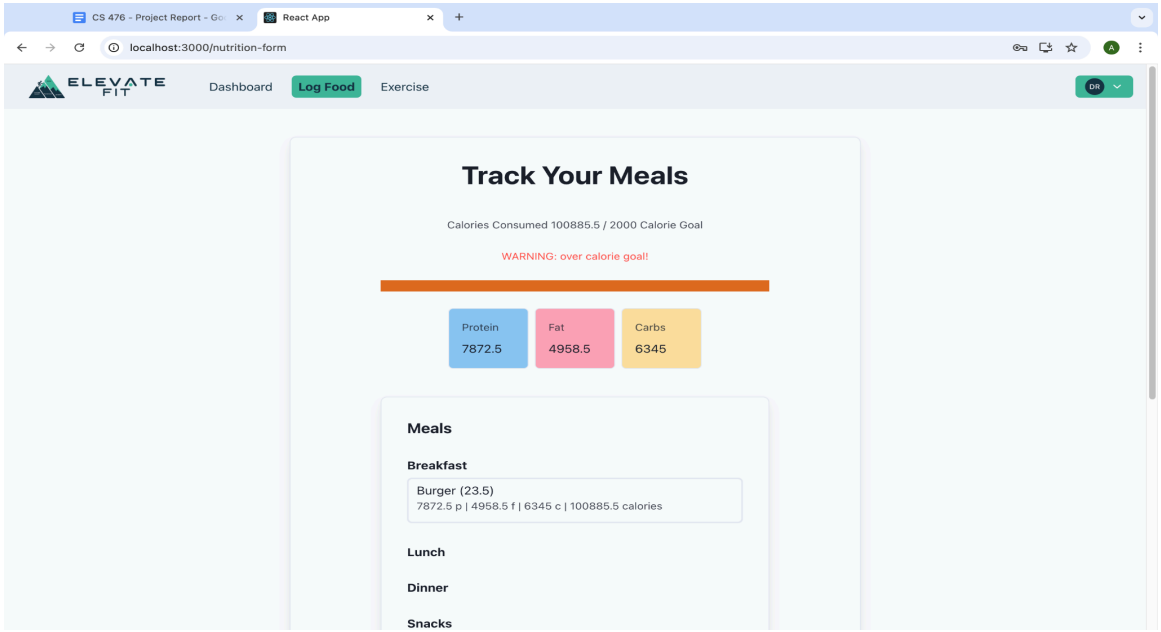
Screenshot of Output:



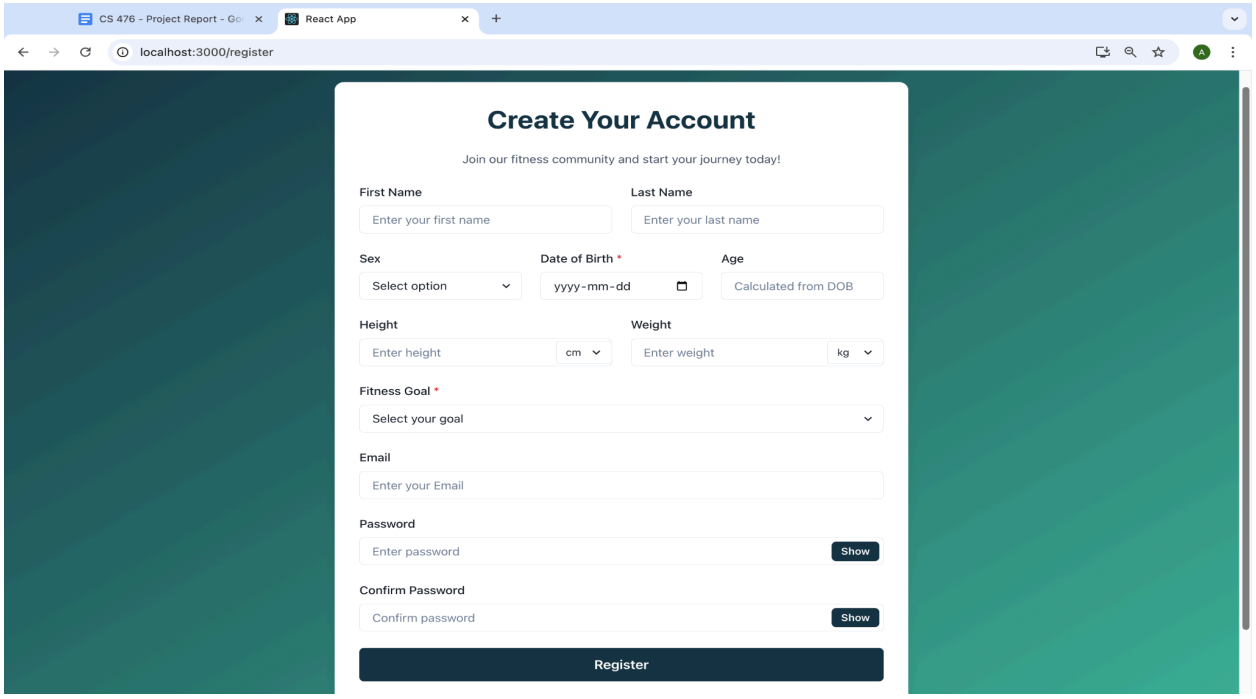
Test case 3: Have 30 burgers for breakfast
Input: Enter and search for a burger. Use the slider to add 30 burgers to your breakfast meal.
Expected Output: Warning Message “Over calorie goal!”
Screenshot of Input:



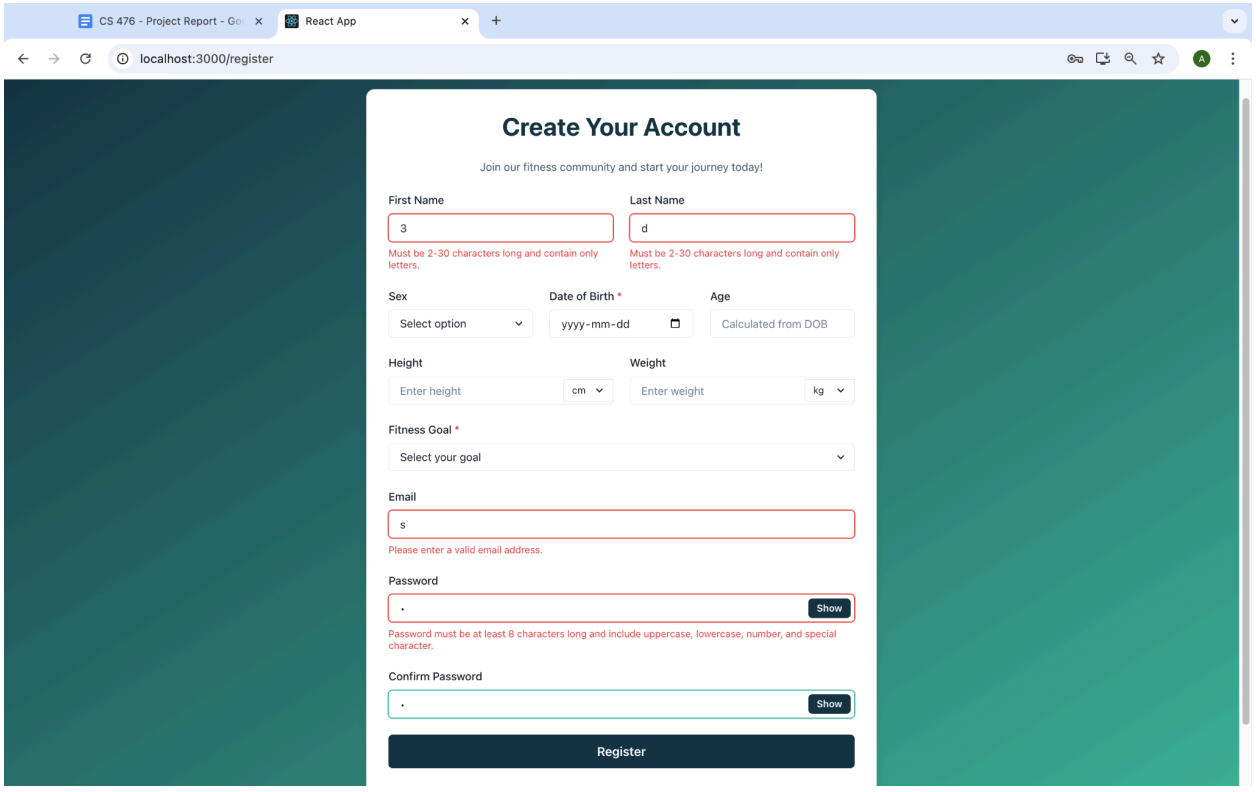
Screenshot of Oupt:



Test Case 4 : Incorrect registration fields
Input: Enter invalid fields in registration field
Expected Output: Error messages with red warning lines around boxes.
Screenshot of input:

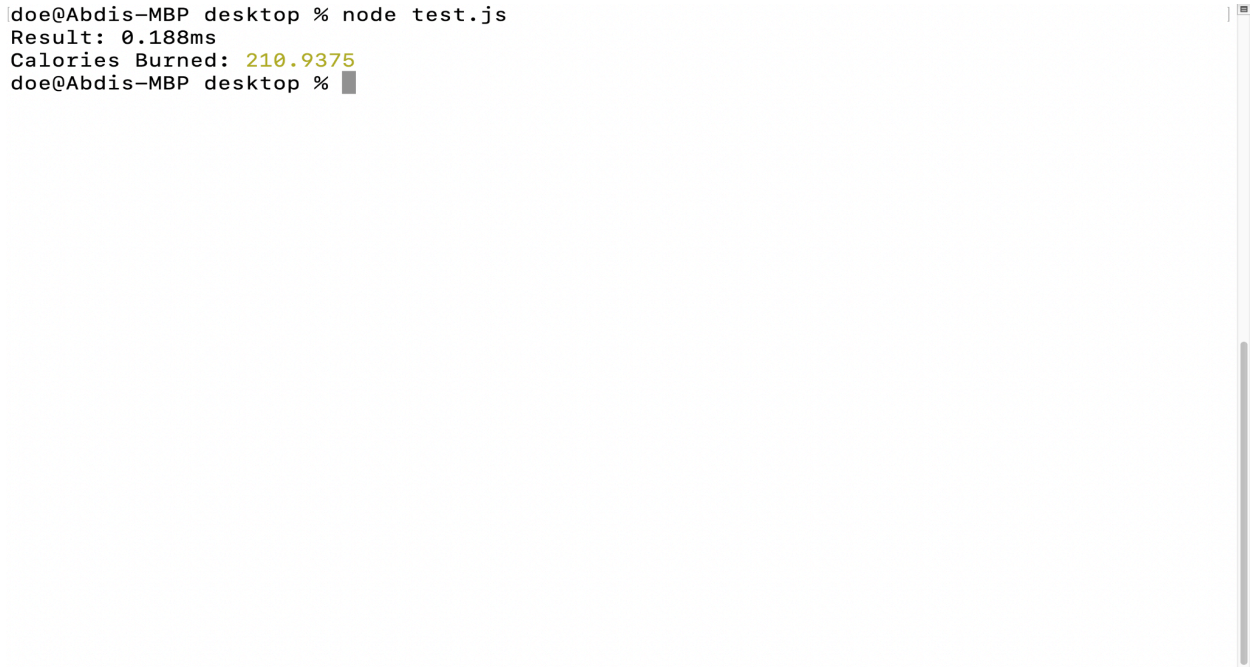


Screenshot of output:



7. C) Time Efficiency Testing

Test Case 1: Testing the time efficiency of the function CalculateCalories() from Exercise.js File.
Method: The method I used to test this case was the console.time() and console.timeEnd() methods in node.js to calculate the exact execution time.
ScreenShot of Output:



Test Case 2: Testing the time efficiency of the function ValidateEMail() from Login.js.
Method: The method I used to test this case was the console.time() and console.timeEnd() methods in node.js to calculate the exact execution time.
ScreenShot of Output: