**GYM LISTING APP**

**MINI PROJECT REPORT**

**SUBMITTED IN PARTIAL FULFILLMENT**

**OF**

**THE REQUIREMENT FOR THE DEGREE OF**

**BACHELOR OF TECHNOLOGY**

**IN**

**INFORMATION TECHNOLOGY**

**BY**

**DOMINIC VANSANGZUALA**

**Roll Number: IT/21/09**

**Registration Number: 2107916**

**UNDER THE SUPERVISION OF**

**Ms. Vanlalmuansangi Khenglawt**

**DEPARTMENT**

**OF**

**INFORMATION TECHNOLOGY**

****

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**MIZORAM UNIVERSITY**

**TANHRIL, AIZAWL,**

**MIZORAM, 796004**



**CERTIFICATE**

This is to certify that the Industrial Training on

*“GYM LISTING APPLICATION”* submitted by DOMINIC VANSANGZUALA in partial fulfilment of the award of the degree of Bachelor of Technology in Information Technology to Mizoram University is a record of bonafide work carried out by his under the supervision of Ms. Vanlalmuansangi Khenglawt, Department of Information Technology. The matter embodied in this Mini-Project report has not been submitted earlier for award of any degree or diploma to the best of my knowledge and belief.

**Signature of Head of External**

**Coordinator Department Examiner**

**ACKNOWLEDGEMENTS**

I am pleased to acknowledge my sincere thanks to the team in Department of Information Technology for their kind encouragement and support both morally and technically towards the completion of this project.

I would like to express my sincere and deep sense of gratitude to my Project Supervisor Ms. Vanlalmuansangi Khenglawt, for her valuable guidance, suggestions and constant encouragement that paved the way for the successful completion of my project work.

I would like to thank God, my parents for their support, and my group members for their co-operation which lead to the completion of this project.

Table of Contents

Table of Contents iv

1. Introduction 1

1.1 Purpose 1

1.2 Scope 1

1.3 Intended Audience 1

1.4 User Classes and Characteristics 2

1.5 Constraints 2

1.6 Assumptions and Dependencies 2

1.7 Problems in Existing System 3

1.8 Proposed System 3

2. Literature Survey 4

2.1 Overview of Existing Fee Collection Systems 4

2.2 Need for Digital Fee Collection System 5

2.3 Technologies and Frameworks 5

2.3.1 HTML 5

2.3.2 npm 7

2.3.3 Tailwind CSS 8

2.3.4 JavaScript and Node.js 10

2.3.5 Vue.js 12

2.3.6 Express.js 14

2.3.7 XAMPP 15

2.3.8 MySQL 16

2.3.8.1 PHPMyAdmin 17

3. Software and Hardware Interface and Requirements 17

3.1 Software and Hardware Requirements 17

3.2 Software Interface 18

3.3 Hardware Interface 20

4. Software Requirement Analysis 21

4.1 Functional Requirements 21

4.2 Non-functional Requirements 23

5. System Analysis and Design 23

5.1 System Description 23

5.3 Use Case Model 25

5.3 Database Design 30

6. Code Snippets 32

6.1 Backend 32

6.1.1 Models 40

6.1.2 Conroltrollers 65

6.2 Frontend 79

6.2.1 Stores 83

6.2.2 Views 107

7. Output Screens 132

8. Testing 144

8.1 Testing Theory 144

8.1.1 Unit Testing 145

8.1.2 Unit Testing 145

8.1.3 Unit Testing 145

8.1.4 Unit Testing 145

8.1.5 Unit Testing 145

9. Future Enhancements 147

10. Conslusions 149

References 149

# Introduction

The Gym Listing and Management App project was developed to streamline the process of finding and managing gym memberships. Traditionally, finding a gym involved extensive research, while gym owners relied on manual methods to attract members and manage operations. This app simplifies the process by allowing users to search for and sign up for gyms online, while gym owners can register their gyms, showcase services, and manage their facilities digitally. This automation enhances convenience, improves accessibility, and fosters efficient management for both gym-goers and gym owners.

* 1. **Purpose**

The Gym Listing and Management App is a mobile and web-based application designed to simplify and automate the process of finding gyms and managing gym memberships. This app allows users to browse and sign up for gyms that meet their needs while providing gym owners with tools to register their gyms, display services, and manage memberships efficiently. The primary goal of this project is to enhance the convenience of connecting gym-goers with facilities while improving operational efficiency for gym owners by reducing manual tasks and streamlining management processes.

* 1. **Scope**

This app is designed to cater to both gym-goers and gym owners, providing a comprehensive platform for gym management and discovery. Key features include a searchable gym directory, user registration for memberships, and a user-friendly interface for seamless navigation. The app integrates with secure payment gateways, enabling users to make online payments for memberships and other services. For gym owners, it offers tools to manage gym details, track memberships, and showcase facilities and services. This system ensures efficient communication between users and gym owners while maintaining secure data handling and transaction processes.

* 1. **Intended Audience**

a) Gym Owners: Fitness facility owners seeking to promote their gyms, manage memberships, and streamline administrative tasks.  
b) Gym-Goers: Individuals looking for convenient options to discover gyms, view services, and sign up for memberships easily and securely.

* 1. **User Classes and Characteristics**

|  |  |
| --- | --- |
| **User Class** | **Characteristics** |
| Gym-Goers/Users | The gym-goers/users will be able to register to a gym for memberships. |
| Gym owners | Gym owners will be able to manage various aspects of their gym through the app. Key tasks will include managing gym profiles, showcasing services and facilities, handling membership plans, monitoring member sign-ups, tracking payments, and responding to inquiries and FAQs. |

* 1. **Constraints**

## Users have reliable internet access to the system, as it is a web-based application.

## Users must access the system from compatible devices.

## Users must have basic knowledge of using web browsers and navigating online platforms.

## The integrated payment gateway must be reliable.

## Users must input accurate information into the system.

## Gym owners must have Razorpay Account.

* 1. **Assumptions and Dependencies**

1. Assumptions:

a) This app is designed to be used by gym owners and gym-goers within a defined geographical region.  
b) Users have reliable internet access, as the app is web and mobile-based.  
c) Users will access the app from compatible devices, such as smartphones, tablets, or computers.  
d) Users have basic knowledge of navigating online platforms and mobile apps.  
e) The integrated Razorpay payment gateway is reliable for handling transactions.  
f) Users will provide accurate information during registration and transactions.  
g) Gym owners have a Razorpay account to facilitate payment processing.

2. Dependencies

1. Payment Gateway: The system depends on third-party payment gateway.
   1. **Problems in Existing System**

The traditional system for finding and managing gym memberships faces several inefficiencies and inconveniences. Manual processes for registering members and managing payments are time-consuming and prone to errors, leading to inaccurate records. Gym-goers often struggle to find suitable gyms due to limited online presence, while gym owners rely on outdated methods to attract members. Payment methods usually require in-person visits, creating inconvenience for users. Additionally, fragmented record-keeping makes it difficult for gym owners to track memberships, payments, and create reports effectively.

* 1. **Proposed System**

The proposed Gym Listing and Management App is a digital solution designed to streamline the process of finding gyms and managing memberships. By automating member registration and payment processing, it reduces manual tasks and errors, ensuring accurate records and saving time for gym owners. Users can browse gym listings, view services, and sign up online from anywhere, offering greater convenience. Payments are updated instantly, eliminating delays, and all records are centralized for easy access and reporting. The system integrates secure payment options through Razorpay, ensuring safety and flexibility. Overall, it provides a faster, easier, and more efficient way to connect users with gyms.

# Literature Survey

* 1. **Overview of Existing Gym Management Systems**

Many gym management systems still rely on manual processes or outdated software, making operations inefficient. Gym-goers often need to visit facilities in person to sign up or make payments, which can be inconvenient, especially for those with busy schedules. These systems require extensive manual data entry, increasing the risk of errors in member records and payment tracking while consuming valuable time for gym owners and staff.

* 1. **Need for Digital Gym Management System**

A digital gym management system is essential for simplifying and accelerating the membership and payment process for gyms. Traditional methods are time-consuming, prone to errors, and often require gym-goers to visit facilities in person, which can be inconvenient. With a digital system, users can browse gyms, register, and make payments online, saving time for both members and gym owners while reducing manual work, leading to more accurate records.

The system provides real-time updates, allowing gym owners to track memberships and payments instantly. Centralized record-keeping makes it easy to access data and generate reports when needed. Secure online payment options, integrated with Razorpay, offer users a convenient and modern way to pay. Overall, a digital system makes gym management faster, more reliable, and more user-friendly for everyone involved.

* 1. **Technologies and Frameworks**

This section explains the technologies and frameworks used in the gym management app. Each tool was chosen to improve user experience, make the system scalable, and increase efficiency and reliability.

* + 1. **HTML**

HTML (HyperText Markup Language) is the foundational language used to build the structure and layout of all web pages within the fee collection system. This project leverages HTML5, the latest version, which introduces new semantic tags and capabilities that make web content more accessible and organized. Semantic elements like <header>, <nav>, <section>, and <footer> provide a logical hierarchy to each page, enhancing both user experience and search engine optimization (SEO). By using these tags, the fee collection system’s interface becomes more intuitive for users and screen readers, ensuring that individuals with disabilities can navigate it effectively.

HTML5's integration with modern accessibility standards ensures that each element’s role on the page is well-defined. This clarity not only helps with accessibility but also improves search engines’ understanding of page content, enhancing the system’s visibility online. Key components of the fee collection interface, such as payment forms and user registration pages, are built using HTML structures that simplify navigation and improve readability.

A key feature of the fee collection system is the use of tables to display important information. HTML tables, constructed with <table>, <tr>, <th>, and <td> tags, are utilized to present data such as student fees, payment history, and fee structures in a structured manner. This organized format allows users to quickly understand and access the information they need, thereby improving overall efficiency.

For administrative functionalities, HTML forms play a crucial role. Admin users can interact with the system through forms created with the <form> tag, which includes various input fields such as <input>, <select>, and <textarea>. These forms enable administrators to manage fee records, update payment statuses, and perform other essential tasks. By using descriptive labels with the <label> tag, the forms become more accessible, ensuring that users know what information is required.

Buttons, created with the <button> tag, are used throughout the system to facilitate various actions. For instance, in the admin panel, buttons allow administrators to submit forms, delete records, or refresh data. On the student interface, buttons are provided for quick payment actions, making it easy for students to complete their transactions with just a click. This functionality enhances user interaction and simplifies navigation within the application.

Form validation is another important aspect of the HTML implementation in this system. HTML5 includes built-in validation features that help ensure users submit correctly formatted data. Attributes like required, minlength, and pattern are used in input fields to prevent incomplete or incorrect submissions. This reduces the likelihood of errors and enhances the reliability of the data collected through the system.

Furthermore, HTML’s ability to integrate with CSS and JavaScript enhances the overall functionality of the fee collection system. CSS is used to style the tables, forms, and buttons, making them visually appealing and ensuring that they are responsive across different devices. JavaScript adds dynamic behavior, such as providing real-time feedback on form validation or updating displayed data without needing to reload the page.

* + 1. **npm**

npm, which stands for Node Package Manager, is an essential tool for developers who work with JavaScript and Node.js. It helps manage libraries and packages that we use in our projects, making it easier to add new features and tools. In our fee collection system, npm was crucial for keeping everything organized and running smoothly.

#### Key Features of npm

1. **Package Management**: npm allows us to easily install and manage software packages. This means we can quickly add libraries that help our project. For instance, we used npm to install important tools like Express for handling server requests.
2. **Version Control**: With npm, we can control which versions of packages we use. This helps avoid problems if a package changes in a way that could break our code. We specify the versions in our package.json file, ensuring our project remains stable.
3. **Script Management**: npm lets us create custom scripts in the package.json file. This simplifies running common tasks, such as starting the server or running tests. For example, we set up a script to start our server with a single command.
4. **Dependency Tracking**: npm tracks all the packages and their dependencies we use in our project. This helps manage complex projects and ensures everything is installed correctly.

#### Setting up npm in our Fee Collection System

1. **Install Node.js**: Before using npm, we need to have Node.js installed on our machine. This is because npm comes bundled with Node.js. We can download and install Node.js from the official website.
2. **Initialize npm**: Once Node.js is installed, we create a new directory for our fee collection system project. Inside this directory, we run the following command in the terminal:

npm init -y

This command initializes a new npm project and creates a package.json file with default settings. The package.json file is important because it keeps track of our project's dependencies and configurations.

1. **Installing Packages**: After initializing npm, we can start adding packages that our project needs. For example, to install Express, we run:

npm install express

This command installs the Express package and adds it to our package.json file under "dependencies."

1. **Creating Scripts**: In our package.json, we can define scripts to simplify our workflow. For example, we added a script to start the server:

"scripts": {

"start": "node server.js"

}

This allows us to start our application by running npm start in the terminal.

1. **Updating Packages**: As we develop our project, we can keep our packages up to date. We run:

npm update

This command updates all our installed packages to their latest versions according to the version rules defined in package.json.

* + 1. **Tailwind CSS**

Tailwind CSS is a modern framework that simplifies web design and development by providing utility classes for styling elements. Unlike traditional CSS frameworks that come with pre-designed components, Tailwind focuses on giving developers low-level utility classes that can be used directly in HTML. This approach allows for more flexibility and creativity in designing user interfaces.

#### Key Features of Tailwind CSS

1. **Utility-First Approach**: Tailwind CSS is built on the principle of using utility classes to style elements. This means that instead of writing custom CSS styles for each component, developers can apply classes directly in the HTML markup. For example, classes like bg-blue-500 or text-center can be used to style an element without writing additional CSS.
2. **Customization**: Tailwind CSS is highly customizable. Developers can easily modify the default design system by changing the configuration file. This allows for tailoring colors, spacing, and other design aspects to fit the specific needs of the project.
3. **Responsive Design**: Tailwind makes it easy to create responsive designs. It provides responsive utility classes that allow developers to specify styles for different screen sizes. For instance, using md:bg-blue-500 will apply a blue background only on medium-sized screens and above.

#### Implementing Tailwind CSS in Our Fee Collection System

1. **Installation**: To use Tailwind CSS in our project, we first need to install it via npm. After setting up npm as mentioned earlier, we can run the following command:

npm install tailwindcss

1. **Configuration**: After installing Tailwind, we create a configuration file by running:

npx tailwindcss init

This command generates a tailwind.config.js file, which allows us to customize Tailwind's default settings according to our design needs.

1. **Creating a Stylesheet**: We then create a CSS file (e.g., styles.css) where we import Tailwind’s base styles, components, and utilities. The content of the file looks like this:

@tailwind base;

@tailwind components;

@tailwind utilities;

1. **Building CSS**: Next, we need to build our CSS file to include Tailwind's styles. We can do this by adding a build script in the package.json:

"scripts": {

"build:css": "tailwindcss build styles.css -o output.css"

}

We can then run the script with:

npm run build:css

This command generates a final CSS file (output.css) that we link to in our HTML

1. **Using Tailwind Classes: With Tailwind CSS integrated, we start applying utility classes directly in our HTML elements. For example, when creating a button for fee payments, we might use:**

**<button class="bg-primary text-white font-bold py-2 px-4 rounded">Pay Fee</button>**

**This applies a green background, white text, bold font, padding, and rounded corners, all in one line.**

* + 1. **JavaScript and NodeJS**

JavaScript is a versatile programming language that enables developers to create dynamic and interactive web applications. It plays a crucial role in both client-side and server-side development. In our fee collection system, JavaScript enhances user experience and provides functionality through real-time interactions, data validation, and seamless communication with the server.

#### JavaScript

1. **Client-Side Scripting**: In our project, we utilize JavaScript for client-side scripting to create dynamic content that updates without requiring a full page reload. For example, when students check their fee details, JavaScript retrieves this information and updates the web page seamlessly, ensuring an efficient and user-friendly experience.
2. **DOM Manipulation**: JavaScript allows us to manipulate the Document Object Model (DOM), enabling real-time changes to the web page structure and content. In the fee collection system, we use JavaScript to create interactive admin functionalities, such as adding or editing fee records. This interaction improves user engagement and streamlines administrative tasks.
3. **Event Handling**: We implement event-driven programming using JavaScript, which captures user actions like button clicks. For instance, when an admin clicks the "Add Fee" button, JavaScript handles the event to display a form for entering fee details, making the system more intuitive.
4. **Form Validation**: To ensure data integrity, we implement client-side form validation using JavaScript. This process checks user inputs in real-time, ensuring that all required fields are filled out correctly before submission. By validating data before it reaches the server, we reduce errors and improve the overall reliability of the system.
5. **AJAX and Fetch API**: We use the Fetch API to make asynchronous requests to the server. For example, when students view their payment history, JavaScript fetches the data from the server and displays it without needing to refresh the page. This feature enhances user experience by providing immediate feedback and updates.

#### Node.js

Node.js is a runtime environment that allows JavaScript to be used on the server side. It enables developers to build scalable and high-performance applications using JavaScript. In our fee collection system, Node.js is integral to our server-side logic.

1. **Server-Side Development**: We write server-side code in JavaScript using Node.js, creating a unified codebase that simplifies development and maintenance. Node.js handles HTTP requests and responses, manages sessions, and interacts with the MySQL database to efficiently manage fee-related data.
2. **Non-Blocking I/O**: Node.js's non-blocking architecture allows our system to handle multiple requests simultaneously. For example, when several students access their fee details at the same time, Node.js processes each request without slowing down the server, ensuring a smooth experience for all users.
3. **Express.js Framework**: We use the Express.js framework to simplify server-side development. Express helps us manage routing, middleware, and API creation effectively. In our application, we set up RESTful APIs to facilitate communication between the frontend and backend, such as fetching student fee records and processing payments.
4. **Middleware Functions**: In our project, we implement middleware functions in Express to handle various tasks, such as authentication and logging. For instance, before processing a payment, we use middleware to validate the incoming data, ensuring that only correctly formatted information is sent to the server.
5. **Database Interaction**: Node.js enables smooth interaction with our MySQL database. We use packages like mysql2 to perform CRUD operations on fee records and student information. This integration allows us to retrieve and update data in real time, providing users with accurate and up-to-date information.
6. **API Development**: We develop RESTful APIs using Node.js and Express to serve as the communication layer between the client-side application and the server. These APIs handle operations such as fetching fee details, submitting payments, and updating records. This modular architecture promotes ease of maintenance and scalability, allowing us to adapt the system as needed.
   * 1. **Vue.js**

Vue.js is a JavaScript framework used to build user interfaces and single-page applications (SPAs). It's easy to learn and can be used for both small and large projects, making it popular among developers.

#### Key Features of Vue.js

1. **Reactive Data Binding**: Vue.js automatically updates the user interface when the data changes. This makes it easy to keep everything in sync without extra work.
2. **Component-Based Structure**: Vue encourages the use of reusable components, which helps organize the code better. Each component contains its template, logic, and styles.
3. **Directives**: Vue has built-in tools like v-if and v-for that simplify how developers manage the HTML and data. These tools make it easy to show or hide elements and loop through data.
4. **Vue Router**: This library helps manage navigation between different parts of the application, allowing users to switch views without reloading the page.
5. **Pinia**: Pinia is used for state management. It keeps track of the application’s data so that different components can access and share it easily.

#### How Vue.js is used in the Gym Management System

1. **Building Components**: The interface is made up of Vue components that separate different functions, like showing student fees or payment history.
2. **Automatic Updates**: When a fee is paid, the information displayed updates automatically. This makes the system more user-friendly.
3. **Managing Forms**: Vue helps handle forms for admin tasks easily. The v-model tool links form inputs directly to data, making it simple to collect and validate user input.
4. **Navigation**: Vue Router is used to move between different views in the application, providing a smooth user experience without page reloads.
5. **State Management with Pinia**: Pinia keeps track of the application's data, ensuring all parts of the app use the latest information. This is vital for managing fee payments and student records.

#### Setting up Vue.js

To set up Vue.js in the project, the following steps were taken:

1. **Create the Project**: The project was started with this command:

npx create-vue@latest

1. **Install Libraries**: Additional tools like Vue Router and Pinia were added using these commands:

npm install vue-router

npm install pinia

1. **Start the Development Server**: A local server was set up to test the application in real time with this command:

npm run dev

* + 1. **Express.js**

Express.js is a fast and minimalist web framework for Node.js. It provides a robust set of features for building web applications and APIs, making it a popular choice for developers. Express simplifies the process of handling HTTP requests and routing, allowing developers to create server-side applications quickly and efficiently.

#### Key Features of Express.js

1. **Middleware Support**: Express uses middleware functions to process requests. These functions can handle tasks such as logging, authentication, and error handling. Middleware allows developers to customize the request-response cycle and add functionality to their applications.
2. **Routing**: Express makes it easy to define routes for different HTTP methods (GET, POST, PUT, DELETE). Developers can create clean and organized routes for various endpoints in their applications, improving maintainability.
3. **Static File Serving**: Express can serve static files, such as images, CSS, and JavaScript. This feature allows developers to create single-page applications that deliver a seamless user experience.
4. **Template Engines**: Express supports various template engines, such as Pug and EJS, which enable dynamic content rendering. This makes it easy to create HTML pages that change based on user input or data.
5. **Error Handling**: Express provides built-in error handling features, allowing developers to manage errors effectively and respond with appropriate messages. This is crucial for improving user experience and debugging.

#### How Express.js is Used in the Fee Collection System

In the fee collection system, Express.js serves as the backbone for handling server-side logic:

1. **Creating the API**: Express is used to build the RESTful API that manages all interactions between the frontend and the backend. It processes requests related to student fees, payments, and other data.
2. **Routing**: Different endpoints are set up for various functions, such as retrieving fee details, submitting payments, and managing admin functionalities. This organization keeps the code clean and easy to navigate.
3. **Middleware Implementation**: Middleware functions are employed for tasks like validating incoming data, authenticating users, and handling errors. This ensures that the system operates smoothly and securely.
4. **Database Interaction**: Express works with MySQL to perform database operations. It retrieves and updates information based on user actions, such as fee payments.
5. **Error Management**: The error-handling features of Express help respond appropriately when something goes wrong, improving the user experience and simplifying troubleshooting.

#### Setting up Express.js

To set up Express.js in the project, the following steps were taken:

1. **Initialize the Project**: The project was initialized with a package.json file using this command:

npm init -y

1. **Install Express**: The Express framework was installed by running:

npm install express

1. **Create the Server**: A basic server was created in a file called server.js, which listens for incoming requests and routes them to the appropriate handlers.
2. **Define Routes**: Routes for handling different functionalities, such as getting fee details and processing payments, were defined in separate files for better organization.
3. **Connect to Database**: Express was configured to connect with MySQL for data management, enabling the application to store and retrieve student fee information.
   * 1. **React Native With Expo**

React Native with Expo is a framework for building mobile applications using JavaScript and React, enhanced with the Expo platform. Expo provides additional tools and services that simplify the development process, such as pre-configured development environments, easy deployment, and access to device APIs. It enables developers to create high-quality, cross-platform mobile apps for both iOS and Android with a single codebase.

**Key Features of React Native with Expo**

1. **Cross-Platform Development**: React Native with Expo allows for writing code once and deploying on both iOS and Android platforms, minimizing development time and effort.
2. **Expo SDK**: Expo comes with a set of built-in APIs for common mobile features like camera access, push notifications, geolocation, and more, without needing native code.
3. **Hot Reloading**: Changes made in the code instantly appear in the app during development, providing a faster iteration cycle.
4. **Declarative UI**: React Native’s declarative approach allows developers to design UIs that are flexible and efficient, updating automatically based on state changes.
5. **Simplified Build and Deployment**: Expo makes the build and deployment process easier with tools to directly create app builds for both iOS and Android, without requiring complex native configurations.

**How React Native with Expo is Used in the Gym Management System**  
a) **Building Components**: The user interface is created using React Native components that handle different aspects of the app, such as gym registration, payment processing, and displaying gym details. Expo simplifies building and testing the UI.  
b) **Cross-Platform Development**: Expo allows for a single codebase to be deployed to both Android and iOS devices, ensuring that the app works seamlessly across platforms.  
c) **Navigation**: React Navigation, integrated with Expo, is used to manage smooth transitions between different app screens like gym listings, registration, and payment processing.  
d) **State Management with Redux**: Redux, used with Expo, helps manage the app’s state, ensuring that data like gym memberships, payment status, and user information are consistently shared across components.  
e) **Form Handling and Validation**: React Native and Expo simplify form handling, making it easy to collect, validate, and store user input for gym sign-ups and payment transactions.

**Setting up React Native with Expo**  
To set up React Native with Expo in the project, the following steps were taken:

* + 1. **Create the Project**: The project was started with Expo using the following command:

npx create-expo-app@latest

* + 1. **Install Libraries:** Libraries such as React Navigation and Redux were installed using the following commands:

npm install @react-navigation/native  
npm install redux react-redux

* + 1. **Start the Development Server**: Expo provides a fast development environment. To start the project on your device or emulator, the following command was used:

npx expo start

* + 1. **XAMPP**

XAMPP is a free and open-source software package that provides an easy way to set up a local server environment on a computer. It includes Apache, MySQL, PHP, and Perl, which are essential components for developing web applications. In the fee collection system project, XAMPP plays a crucial role in providing the necessary tools for building and testing the application locally.

When using XAMPP, both the Apache web server and MySQL database can be started with just a few clicks. This setup allows developers to run their web applications and access the database without needing to configure complex server settings. The simplicity of XAMPP makes it accessible for developers of all skill levels.

XAMPP also includes PHPMyAdmin, a web-based interface for managing MySQL databases, which further simplifies the process of creating and modifying database structures. This integration allows for quick access to database management tools right from the local server environment.

* + 1. **MySQL**

PostgreSQL is a robust and open-source relational database management system widely used for managing structured data. In this project, PostgreSQL plays a vital role in organizing and tracking key information related to gyms, memberships, and payments. The system supports multiple transactions, ensuring that all data is accurate, consistent, and readily accessible.

To set up PostgreSQL, the PostgreSQL server is installed on a local machine, complemented by **pgAdmin**, a user-friendly, web-based interface that simplifies database management. This setup facilitates tasks such as adding new gym members, updating membership plans, and monitoring payment records.

PostgreSQL efficiently handles operations such as inserting new records when users register for a gym and processing their payments. It uses foreign keys to maintain data integrity, ensuring that payments are correctly associated with the relevant gym member.

PostgreSQL is seamlessly integrated with Express.js, enabling smooth communication between the application and the database. This integration ensures that the gym management system operates reliably and efficiently.

# Software and Hardware Interface and Requirements

* 1. **Software and Hardware Requirements**

1. HTTP: Hypertext Transfer Protocol is a transaction-oriented client/server protocol between a web browser and a web server
2. HTTPS: Secure Hypertext Transfer Protocol is HTTP over SSL (Secure Socket Layer), providing secure communication over the internet.
3. React Native: A framework that allows developers to build cross-platform mobile applications for iOS and Android using JavaScript and React. It provides a high-performance and responsive user interface for mobile devices.
4. Express.js: Express is a web application framework for Node.js, used to build the server-side application and handle HTTP requests efficiently.
5. JSON: A lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It is commonly used to transmit data between a server and a web application.
6. PostgreSQL: A powerful, open-source relational database management system (RDBMS) used to store and manage data. PostgreSQL is known for its robustness, scalability, and support for advanced data types and queries.
   1. **Software Interface**

**Frontend**

1. **HTML:** HTML is a markup language for structuring and presenting content for the World Wide Web and a core technology of the Internet. It is the fifth revision of the HTML standard. The purpose of a web browser is to read HTML documents and compose them into visible or audible web pages. The browser does not display the HTML tags, but used the tags to interpret the content of the page. HTML elements form the building blocks of all websites. It provides a means to create structured documents by denoting structural semantics for text such as headings, paragraphs, lists, forms and other items. It can embed scripts written in languages such as JavaScript which affect the behavior of HTML web pages.
2. **JavaScript:** JavaScript is and interpreted computer programming language. It was originally implemented as part of web browsers so the client-side scripts could interact with the user, control the browser, communicate asynchronously, and alter the document content that was displayed. JavaScript is a prototype-based scripting language that is dynamic, weakly typed, and has first-class function. Its syntax was influenced by the language C. JavaScript copies many names and naming conventions from Java, but the two languages are otherwise unrelated and have very different semantics. The key design principles within JavaScript are taken from the self and Scheme programming languages. It is a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles.
3. **Vue.js:** Vue.js (version 3.x) is the primary framework utilized in the fee collection system to create a dynamic and responsive user interface. This framework enables the development of single-page applications (SPAs), which are designed to load quickly and deliver a seamless user experience. By minimizing page reloads, Vue.js enhances interaction and ensures that users can navigate the application smoothly without interruption.

One of the key features of Vue.js is its reactive data-binding system. This allows the UI to automatically update in response to changes in the underlying data, providing users with real-time feedback and ensuring that they always see the most current information. For instance, when a student views their fee status, any updates made in the backend are instantly reflected in the frontend without the need for manual refreshes.

Vue.js also promotes a component-based architecture, which encourages the reuse of code and improves maintainability.

1. **Tailwind CSS:** Tailwind CSS is employed for styling the application, enabling developers to create custom designs using utility classes. This approach streamlines the styling process, promotes consistency across the application, and allows for rapid prototyping of UI components.
2. **React Native:** React Native is a framework for building mobile applications using JavaScript and React. It enables developers to create cross-platform apps for both iOS and Android with a single codebase. React Native uses native components for high performance and offers features like hot reloading, declarative UI, and easy integration with native modules, making development faster and more efficient**.**

**Backend**

1. **Node.js:** The application runs on Node.js (version 14.x or higher), which allows for server-side JavaScript execution. Node.js's non-blocking architecture enables efficient handling of multiple requests simultaneously, improving the application's performance and scalability.
2. **Express.**js: Express.js (version 4.x) is utilized to build the server-side application. It

simplifies the process of setting up middleware to respond to HTTP requests, enabling the creation of robust RESTful APIs for seamless communication between the client and server.

**Web Server**

1. **Web Server Software**: Apache or Nginx is recommended for hosting the application. Both servers are well-suited for serving static and dynamic content, providing high reliability and support for modern web standards such as HTTPS for secure data transmission.
2. **Server Configuration**: Appropriate configurations, including the use of virtual hosts and SSL certificates, are essential to ensure secure connections and optimal performance for users accessing the fee collection system.

**Database Server**

**a) Database Management System:** PostgreSQL (version 13 or higher) is used to handle the application's database operations, including data storage, retrieval, and management of complex queries. PostgreSQL’s robust support for transactions and advanced data types ensures data integrity and consistency across related operations.

**b) Database Management Tool:** pgAdmin is utilized for managing the PostgreSQL database through an intuitive web interface. It allows administrators to perform tasks such as adding records, executing queries, and generating reports, making database management processes efficient and user-friendly.

* 1. **Hardware Interface**

1. **Server Specifications**: A server with at least 2 CPU cores, 4 GB of RAM, and 20 GB of storage is recommended for optimal performance.
2. **Client Specifications**: Users should have a minimum of 4 GB of RAM and a modern processor for accessing the application smoothly.
3. **Network Requirements**: A stable internet connection with a minimum bandwidth of 1 Mbps is required to ensure a seamless user experience.

# Software Requirement Analysis

* 1. **Functional Requirements**

### User Functions

a) Account Creation & Verification: The system must allow gym goers to create an account and verify their email or phone number for secure access.

b) Search and View Gym Details: Once registered, gym goers can search for gyms and view details such as location, facilities, membership plans, and reviews.

c) Membership Enrollment: The system must provide a list of available membership plans for selected gyms. Gym goers can select a plan and make payments directly through the app.

d) Payment Receipts: After payment, the system must generate a digital receipt for the gym goer, which can be accessed anytime.

e) Contact Support: The system must offer a "Contact Us" page where gym goers can find the contact information for gym administrators or app support. It should also provide a form where users can submit their contact details and inquiries for assistance.

### Gym Owners Functions

a) **Admin Login**: The system must provide a login form for gym owners. After a registered gym owner receives their login credentials, they can log in through the admin login page.

b) **Verification**: The system must verify the gym owner’s login credentials.

c) **Admin Dashboard**: Upon successful login, the system must direct the gym owner to an admin dashboard that provides tools for managing their gym operations.

d) **Manage Membership Plans**: The system must allow gym owners to create and update membership plans, including pricing, duration, and benefits.

e) **Manage Gym Members**: The system must enable gym owners to assign and track members’ selected membership plans and monitor their enrollment status.

f) **Monitor Payments**: Gym owners can view payment history, including details of membership payments made by gym goers.

g) **Manage Staff Accounts:** The system must allow gym owners to add, edit, or remove accounts for their staff to help with gym management tasks.

h) **Manage Gym Details**: Gym owners can update information about their gym, such as location, facilities, operating hours, and images.

i) **Handle Inquiries**: The system must provide a page where gym owners can view and respond to inquiries submitted by users interested in their gym.

j) **Manage FAQs**: Gym owners can create, edit, or delete Frequently Asked Questions to assist users with common queries.

k) **Analytics Dashboard**: The system must provide a dashboard displaying statistics and insights about gym performance, such as member sign-ups, payments, and inquiries.

l) **Admin Levels**: The system must support two types of admin roles – Head Admin and Assistant Admin – with distinct permissions to streamline gym management.

* 1. **Non-functional Requirements**

## Performance Requirements

1. The system should respond to user actions within 5 seconds under normal operating conditions.
2. Responses to queries shall take no longer than 7 seconds to load onto the screen after user submits query.

## 

## Security Requirements

1. All sensitive data must be encrypted.

## 

## Software Quality Attributes

1. Availability: The system must be available 99% of the time, excluding scheduled maintenance periods.
2. Maintainability: Coding standards must be followed for easier maintenance.
3. Portability: The system must be compatible with major web browsers
4. Usability: The system must be designed by following basic UX principles.

# System Analysis and Design

* 1. **System Description**

Gym Goer Interaction:The home page of this mobile app consists of navigation links that lead to different sections of the app, such as gym goer registration, FAQs, contact information, terms and conditions, and gym owner login.

Steps for Gym Registration

a) Click on the button ‘Register for a Gym’.  
b) Search for and select a gym from the available list.  
c) Choose a membership plan.  
d) Fill in your personal details, including contact information.  
e) Click the ‘Pay Now’ button to complete the payment for the selected plan.  
f) Click the **‘**View Receipt’ button to access and download your registration receipt.

**Steps for contacting Gym Owner**

1. Click on the ‘Contact Us’ link.
2. Fill up the required field and click on submit button.

**Steps for viewing Frequently Asked Questions**

1. Click on the link ‘FAQs’

**Steps for viewing Terms and Conditions**

1. Click on the link ‘Terms and Conditions’

**Gym Owner Interaction**: Gym owners manage their gym operations by inputting membership plans, monitoring member enrollments, and viewing payment reports. They log into the system with secure credentials to update gym details, track membership statuses, and analyze payment histories, ensuring efficient management and operationaltransparency**.**

**Payment Gateway Interaction**:The system integrates with a secure payment gateway to process gym membership payments. Once a gym goer initiates a payment, the system sends a request to the payment gateway. After processing, the gateway provides a payment confirmation, which is then recorded in the system to ensure accurate tracking of membership payments.

.

**The following shows a system context diagram for this project-**

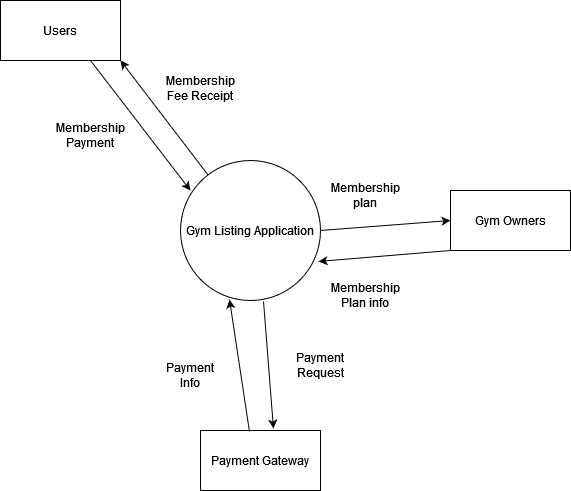


Fig: 5.1: System Context Diagram

* 1. **Use Case Model**

The following figure presents a Use Case diagram for the Payment Collection System for Educational Institutions.

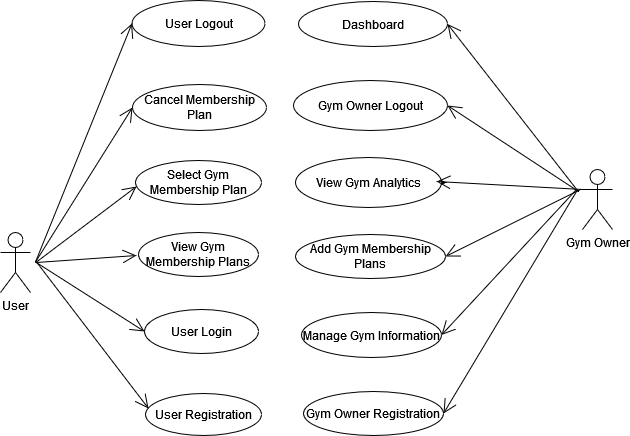


Fig 4.3.1: Use Case Model

### Use case Description

**User Registration**

|  |  |
| --- | --- |
| Actor | User |
| Description | A new user registers on the gym app by providing necessary details to create an account. |
| Precondition | User must not have an existing account on the gym app. |
| Postcondition | A new user account is created. |
| Normal Flow | 1. The user accesses the gym app. 2. The user selects the "Register" option. 3. The user enters necessary details (e.g., name, email, password, contact number). 4. The system verifies the input data. 5. The user receives a confirmation message. 6. The system creates a new user account. 7. The user is logged in and redirected to the main page. |

**User Login**

|  |  |
| --- | --- |
| Actor | User |
| Description | A registered user logs into the gym app using their credentials. |
| Precondition | User must have a valid registered account |
| Postcondition | The user is logged into the app |
| Normal Flow | 1. The user accesses the gym app. 2. The user selects the "Login" option. 3. The user enters their username and password. 4. The system verifies the credentials. 5. The user is granted access to the app's main page. |

**View Gym Membership Plans**

|  |  |
| --- | --- |
| Actor | User |
| Description | A user views available gym membership plans |
| Precondition | User must be logged into the app |
| Postcondition | The user sees a list of available gym membership plans. |
| Normal Flow | 1. The user logs into the gym app. 2. The user selects the "Membership Plans" option. 3. The system displays a list of available membership plans, including details such as price, duration, and benefits. 4. The user browses through the plans and selects one for more details. |

**Select Gym Membership Plan**

|  |  |
| --- | --- |
| Actor | User |
| Description | A user selects a gym membership plan to purchase or subscribe to |
| Precondition | User must be logged into the app and view available membership plans |
| Postcondition | The user subscribes to the selected membership plan |
| Normal Flow | 1. The user logs into the gym app. 2. The user selects the "Membership Plans" option. 3. The user selects a membership plan they want to subscribe to. 4. The system displays a summary of the plan and the total cost. 5. The user confirms the subscription. 6. The system processes the subscription and activates the selected membership plan. 7. The user receives a confirmation notification. |

**View Membership Plan Details**

|  |  |
| --- | --- |
| Actor | User |
| Description | A user cancels their active gym membership plan. |
| Precondition | User must be logged into the app and have an active membership plan. |
| Postcondition | The user views detailed information about their membership. |
| Normal Flow | 1. The user logs into the gym app. 2. The user selects the "My Membership" option. 3. The system displays details of the user's active membership plan (e.g., start date, end date, benefits, payment history). |

**Cancel Membership Plan**

|  |  |
| --- | --- |
| Actor | User |
| Description | The admin adds, edit, delete student, allocate each student to their fee structures. |
| Precondition | User must be logged into the app and have an active membership plan. |
| Postcondition | The user’s membership plan is canceled and access is revoked. |
| Normal Flow | 1. The user logs into the gym app. 2. The user selects the "My Membership" option. 3. The user selects the "Cancel Membership" option. 4. The system confirms the cancellation. 5. The user's membership is canceled, and they are notified of the cancellation. |

**Gym Owner Registration**

|  |  |
| --- | --- |
| Actor | Gym Owner |
| Description | A gym owner registers their gym in the app, providing necessary details. |
| Precondition | The gym owner must not have a registered gym. |
| Postcondition | The gym is registered in the app with its details available for users to see. |
| Normal Flow | 1. The gym owner accesses the gym app. 2. The gym owner selects the "Register Gym" option. 3. The gym owner enters details such as gym name, location, contact information, and available facilities. 4. The system verifies the data. 5. The gym is registered in the system. 6. The gym owner receives a confirmation message and can manage the gym details from the admin panel. |

**Manage Gym Information**

|  |  |
| --- | --- |
| Actor | Gym Owner |
| Description | The gym owner updates and manages their gym information. |
| Precondition | Gym owner must be logged in and have a registered gym. |
| Postcondition | Gym details are updated in the system. |
| Normal Flow | 1. The gym owner logs into the gym app. 2. The gym owner selects the "Manage Gym Info" option. 3. The gym owner updates details such as contact information, facilities, and gym services. 4. The system saves the updated information.. |

**Add Gym Membership Plans**

|  |  |
| --- | --- |
| Actor | Gym Owner |
| Description | The gym owner adds new membership plans for users to subscribe to. |
| Precondition | Gym owner must be logged in and have a registered gym. |
| Postcondition | The new membership plans are added to the system and available for users to choose from.. |
| Normal Flow | 1. The gym owner logs into the gym app. 2. The gym owner selects the "Add Membership Plan" option. 3. The gym owner enters membership details such as name, duration, price, and benefits. 4. The system saves the new membership plan and makes it available for users to subscribe to. |

**View Gym Analytics**

|  |  |
| --- | --- |
| Actor | Owner |
| Description | The gym owner views analytics such as the number of memberships sold, revenue, and user engagement. |
| Precondition | Gym owner must be logged into the app. |
| Postcondition | The gym owner can view relevant statistics and analytics. |
| Normal Flow | 1. The gym owner logs into the gym app. 2. The gym owner selects the "Analytics" or "Reports" option. 3. The system displays detailed statistics, such as total memberships sold, revenue generated, and the number of active users. |

**User Logout**

|  |  |
| --- | --- |
| Actor | User |
| Description | The user logs out of the app. |
| Precondition | User must be logged into the app. |
| Postcondition | User is logged out, and the session is terminated. |
| Normal Flow | 1. The user selects the "Logout" option. 2. The system logs the user out and terminates the session. |

**Gym Owner Logout**

|  |  |
| --- | --- |
| Actor | Gym Owner |
| Description | The gym owner logs out of the app. |
| Precondition | Gym owner must be logged into the app. |
| Postcondition | Gym owner is logged out, and the session is terminated. |
| Normal Flow | 1. The gym owner selects the "Logout" option. 2. The system logs the gym owner out and terminates the session. |

**Dashboard (Gym Owner)**

|  |  |
| --- | --- |
| Actor | Gym Owner |
| Description | The gym owner views a comprehensive dashboard displaying key metrics, such as the total number of memberships, revenue, class attendance, and other gym activities. |
| Precondition | Gym owner must be logged into the app. |
| Postcondition | The gym owner can track the performance of their gym and make data-driven decisions. |
| Normal Flow | 1. The gym owner logs into the gym app. 2. The gym owner selects the "Dashboard" option. 3. The system displays the gym owner’s dashboard with key metrics such as the number of active memberships, total revenue, new subscriptions, and engagement levels. 4. The gym owner reviews the statistics and can drill down into specific data points for more detailed insights. |
|  |  |

* 1. **Database Design**

Database design depicts the tables and relationships between them for sharing different data.

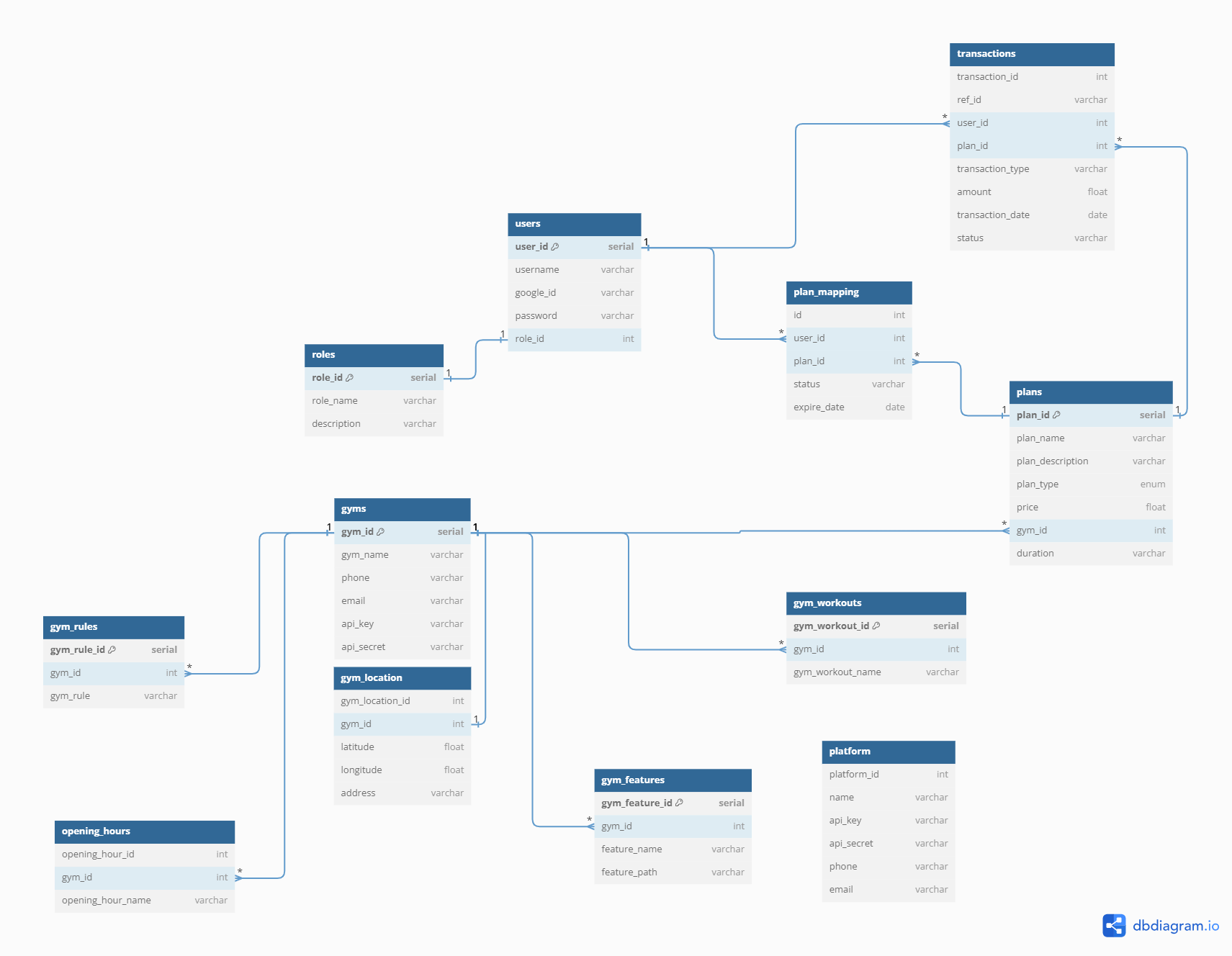


Fig: 5.2: ER Diagram

# Code Snippets

* 1. **Backend**

**db.js**

import dotenv from 'dotenv'

dotenv.config();

const dbPassword = process.env.DB\_PASSWORD;

const dbName = process.env.DB\_NAME;

const dbType = process.env.DB\_TYPE;

const dbHost = process.env.DB\_HOST;

import { Sequelize } from 'sequelize';

const sequelize = new Sequelize(dbName, dbType, dbPassword, {

  host: dbHost,

  dialect: 'postgres',

  logging: false,

});

sequelize.authenticate()

  .then(() => {

    console.log('Connection has been established successfully.');

  })

  .catch(err => {

    console.error('Unable to connect to the database:', err);

  });

export default sequelize;

**index.js**

import express from "express";

import db from './config/db.js'

import sequelize from "./config/db.js";

import { Gym } from "./models/Gym.js";

import { User } from "./models/User.js";

import { GymLocation } from "./models/Gym.js";

import { Plan } from "./models/Gym.js";

import { PlanMapping } from "./models/Plans.js";

import { Transaction } from "./models/Transaction.js";

import { Platform } from "./models/Platform.js";

import { OTP } from "./models/OTP.js";

import bodyParser from 'body-parser'

import bcrypt from 'bcrypt'

import router from "./routes/routes.js";

import cors from 'cors';

import path from "path";

import multer from "multer";

import { fileURLToPath } from "url";

const \_\_filename = fileURLToPath(import.meta.url);

const \_\_dirname = path.dirname(\_\_filename);

const app = express();

const corsOptions = {

  origin: 'http://localhost:5173',

  methods: ['GET', 'POST', 'PUT', 'DELETE'],

  credentials: true,

};

// Use the CORS middleware globally

app.use(cors());

// Middleware to parse JSON requests

app.use(express.json());

app.use(router);

app.use('/uploads', express.static(path.join(\_\_dirname, 'uploads')));

app.use(express.json());

app.use(bodyParser.urlencoded({ extended: true }))

// Synchronize the database

sequelize.sync({ alter: true })

  .then(() => {

    app.listen(3000, () => console.log("Listening at port 3000"));

  })

  .catch((error) => {

    console.error('Error syncing database:', error);

  });

app.get('/', async (req, res) => {

  try {

    res.send("hello")

  } catch (err) {

    res.send(err)

  }})

**routes.js**

import express from "express";

import { createNotif, delUserByEmail, getAllGymMembers, getAllUsers, googleLogin, insertUser } from "../controllers/userController.js";

import { sendOTP } from "../controllers/otp.js";

import { verifyOTP } from "../controllers/otp.js";

import { userLogin, findUserFromToken } from "../controllers/login.js";

import { checkPlan, getPlanByMap, getSubscriptionPlans, getUserPlan, insertPlanMapping, insertPlanMappingDraft, rechargePlanMapping, startPlanMapping } from "../controllers/planController.js";

import { authenticateToken } from "../controllers/login.js";

import { getSubscriptionPlanById } from "../controllers/planController.js";

import { decodeToken } from "../controllers/login.js";

import { addGymImage, addMembershipPlan, changeGymInfo, changeGymPp, checkGymAdmin, countData, deleteGymImage, deleteMembershipPlan, getAllGyms, getFeatures, getGymAdminFeatures, getGymAdminWorkouts, getGymTransactions, getMemberships, getMyGyms,  getNotif, getSaved, getUserSavedGyms, getWorkouts, insertGym, saveGym, searchGyms, setGymVerified, updateGymAdminFeature, updateGymAdminWorkout, updateMembershipPlan } from "../controllers/gymController.js"; // New controller for gym registration

import { getGymById, getGyms } from "../controllers/gymController.js";

import { PaymentGateway } from "../models/paymentGateway.js";

import { createOrder, getPublicKey, paymentDetails, verifyPayment } from "../controllers/paymentGateway.js";

import '../controllers/paymentGateway.js'

import path from "path";

import multer from "multer";

import { fileURLToPath } from "url";

import { insertTransaction } from "../controllers/transactionController.js";

const router = express.Router();

const \_\_filename = fileURLToPath(import.meta.url);

const \_\_dirname = path.dirname(\_\_filename);

// Multer configuration for file uploads

const storage = multer.diskStorage({

  destination: (req, file, cb) => {

    cb(null, 'uploads');

  },

  filename: (req, file, cb) => {

    cb(null, Date.now() + path.extname(file.originalname)); // Generate unique filename

  }

});

const upload = multer({ storage: storage });

router.post('/register-gym', upload.fields([

  { name: 'gymImages', maxCount: 10 },

  { name: 'gymProfileImage', maxCount: 1 },

  { name: 'gymLicense', maxCount: 1 }

]), insertGym);

router.post('/notif/add', createNotif)

router.post('/admin/gym/set-verified', setGymVerified)

router.post('/register', insertUser);

// OTP

router.post('/send-otp', sendOTP);

router.post('/verify-otp', verifyOTP);

// LOGIN

router.post('/login', userLogin);

router.post('/user-token', findUserFromToken);

// google login

router.post('/api/auth/google', googleLogin)

// USERS

router.get('/admin/users', getAllUsers)

router.delete('/user/:email', delUserByEmail)

// decode token

router.post('/decode-token', decodeToken);

// gym

router.get('/gyms', getGyms);

router.get('/gyms/:id', getGymById);

router.get('/admin/gyms', getAllGyms)

// saved

router.post('/save-gym', saveGym)

router.get('/user-saved/:userId', getSaved)

router.get('/gyms/saved/:userId', getUserSavedGyms)

// gym owned

router.get('/user/gyms/:userId', getMyGyms)

// notification

router.get('/user/notif/:userId', getNotif)

// SUBSCRIPTION PLANS

router.get('/subscription-plans', getSubscriptionPlans);

router.get('/subscription-plans/:id', getSubscriptionPlanById);

// PAYMENT GATEWAY

router.post('/create-order', createOrder)

router.post('/verify-payment', verifyPayment)

router.get('/payment-details/:id', paymentDetails)

router.post('/public-key', getPublicKey)

// TRANSACTIOn

router.post('/store-transaction', insertTransaction)

// PLAN

router.post('/insert-plan', insertPlanMapping)

// router.post('/insert-plan/draft', insertPlanMappingDraft)

router.post('/plan-mapping/recharge/:planMapId', rechargePlanMapping)

router.get('/plan-mapping/plan/:planMapId', getPlanByMap)

router.get('/user/plans/:userId', getUserPlan)

// FEATURES

router.get('/features', getFeatures)

// WORKOUTS

router.get('/workouts', getWorkouts)

// QR CODE

router.get('/check-plan/:planMapId', checkPlan);

// gym search

router.get('/search-gyms', searchGyms)

// gymadmin

router.get('/gym-admin/gym/:gymId/:ownerId', checkGymAdmin)

router.get('/gym-admin/members/:gymId', getAllGymMembers)

router.get('/gym-admin/memberships/:gymId', getMemberships)

router.post('/gym-admin/add-gym', upload.fields([

  { name: 'gymImages', maxCount: 10 }

]), addGymImage)

router.put('/gym-admin/edit-profile-pic/:gymId', upload.fields([

  { name: 'pp', maxCount: 1 }

]), changeGymPp)

router.put('/gym-admin/change-gym-info/:gymId', changeGymInfo)

router.get('/gym-admin/transactions/:gymId', getGymTransactions)

router.delete('/gym-admin/delete-image/:imageId/:gymId', deleteGymImage)

router.get('/gym-admin/workouts/:gymId', getGymAdminWorkouts)

router.get('/gym-admin/features/:gymId', getGymAdminFeatures)

router.post('/gym-admin/update-gym-feature/:gymFeatureId/:gymId', updateGymAdminFeature)

router.post('/gym-admin/update-gym-workout/:gymWorkoutId/:gymId', updateGymAdminWorkout)

router.post('/gym-admin/add-membership-plan/:gymId', addMembershipPlan)

router.post('/gym-admin/update-membership-plan/:planId', updateMembershipPlan)

router.delete('/gym-admin/delete-membership-plan/:planId', deleteMembershipPlan)

router.get('/gym-admin/dashboard/:gymId', countData)

export default router;

* + 1. **Models**

**Gym.js**

import sequelize from "../config/db.js";

import { DataTypes } from "sequelize";

import { User } from "./User.js";

// GYM

export const Gym = sequelize.define('Gym', {

    gymName: {

        type: DataTypes.STRING,

        allowNull: false

    },

    gymPhone: {

        type: DataTypes.STRING,

        allowNull: false

    },

    gymEmail: {

        type: DataTypes.STRING,

        allowNull: false

    },

    profileImage: {

        type: DataTypes.STRING,

        allowNull: true

    },

    gymLicense: {

        type: DataTypes.STRING,

        allowNull: true

    },

    ownerId: {

        type: DataTypes.INTEGER,

        allowNull: false,

        references: {

            model: User,

            key: 'id'

        }

    },

    status: {

        type: DataTypes.STRING,

        allowNull: false

    }

}, {

    timestamps: true

})

User.hasMany(Gym, { foreignKey: "ownerId" });

Gym.belongsTo(User, { foreignKey: "ownerId" });

// PLAN (MEMEBERSHIPS AND SUBSCRIPTION)

export const Plan = sequelize.define('Plan', {

    planName: {

        type: DataTypes.STRING,

        allowNull: false

    },

    planDescription: {

        type: DataTypes.STRING,

        allowNull: false

    },

    planType: {

        type: DataTypes.ENUM('membership', 'subscription'),

        allowNull: false

    },

    price: {

        type: DataTypes.FLOAT,

        allowNull: false

    },

    version: {

        type: DataTypes.INTEGER,

        allowNull: true

    },

    gymId: {

        type: DataTypes.INTEGER,

        references: {

            model: Gym,

            key: 'id'

        },

        allowNull: true

    },

    state: {

        type: DataTypes.ENUM('active', 'inactive'),

        allowNull: false

    },

    duration: {

        type: DataTypes.STRING

    }

}, { timestamps: true })

export const GymLocation = sequelize.define('GymLocation', {

    buildingNo: {

        type: DataTypes.STRING,

        allowNull: true

    },

    latitude: {

        type: DataTypes.FLOAT,

        allowNull: false

    },

    longitude: {

        type: DataTypes.FLOAT,

        allowNull: false

    },

    area: {

        type: DataTypes.STRING,

        allowNull: false

    },

    city: {

        type: DataTypes.STRING,

        allowNull: false

    },

    landmark: {

        type: DataTypes.STRING,

        allowNull: true

    },

    gymId: {

        type: DataTypes.INTEGER,

        allowNull: false,

        references: {

            model: Gym,

            key: 'id'

        }

    }

}, { timestamps: true })

export const GymRule = sequelize.define('GymRule', {

    gymRule: DataTypes.STRING,

    gymId: {

        type: DataTypes.INTEGER,

        allowNull: false,

        references: {

            model: Gym,

            key: 'id'

        }

    }

})

export const GymFeature = sequelize.define('GymFeature', {

    featureName: {

        type: DataTypes.STRING,

        allowNull: false

    },

    featureImgPath: {

        type: DataTypes.STRING,

        allowNull: true

    }

}, {

    timestamps: true

})

export const GymFeatureMapping = sequelize.define('GymFeatureMapping', {

    gymId: {

        type: DataTypes.INTEGER,

        allowNull: false,

        references: {

            model: Gym,

            key: 'id'

        }

    },

    gymFeatureId: {

        type: DataTypes.INTEGER,

        allowNull: false,

        references: {

            model: GymFeature,

            key: 'id'

        }

    }

}, { timestamps: true })

export const GymWorkout = sequelize.define('GymWorkout', {

    workoutName: {

        type: DataTypes.STRING,

        allowNull: false

    }

}, { timestamps: true })

export const GymWorkoutMapping = sequelize.define('GymWorkoutMapping', {

    gymId: {

        type: DataTypes.INTEGER,

        allowNull: false,

        references: {

            model: Gym,

            key: 'id'

        }

    },

    gymWorkoutId: {

        type: DataTypes.INTEGER,

        allowNull: false,

        references: {

            model: GymWorkout,

            key: 'id'

        }

    }

})

export const GymOpeningHours = sequelize.define('GymOpeningHour', {

    gymId: {

        type: DataTypes.INTEGER,

        allowNull: false,

        references: {

            model: Gym,

            key: 'id'

        }

    },

    morning: {

        type: DataTypes.STRING,

        allowNull: false,

    },

    evening: {

        type: DataTypes.STRING,

        allowNull: false

    }

}, { timestamps: true })

export const GymImages = sequelize.define('GymImages', {

    gymId: {

        type: DataTypes.INTEGER,

        allowNull: false,

        references: {

            model: Gym,

            key: 'id'

        }

    },

    gymImgPath: {

        type: DataTypes.STRING,

        allowNull: false,

    }

}, { timestamps: true })

export const Notification = sequelize.define('Notification', {

    userId: {

        type: DataTypes.INTEGER,

        allowNull: false,

        references: {

            model: User,

            key: 'id'

        }

    },

    senderId:{

        type: DataTypes.INTEGER,

        allowNull: true,

        references: {

            model: Gym,

            key: 'id'

        }

    },

    title: {

        type: DataTypes.STRING,

        allowNull: false

    },

    message: {

        type: DataTypes.STRING,

        allowNull: false,

    },

    linkContent: {

        type: DataTypes.STRING,

        allowNull: true

    },

    link: {

        type: DataTypes.STRING,

        allowNull: true

    },

    seen:{

        type: DataTypes.BOOLEAN,

        defaultValue: false,

        allowNull: true

    }

}, { timestamps: true })

export const Saved = sequelize.define('Saved', {

    userId: {

        type: DataTypes.INTEGER,

        allowNull: false,

        references: {

            model: User,

            key: 'id'

        }

    },

    gymId: {

        type: DataTypes.INTEGER,

        allowNull: false,

        references: {

            model: Gym,

            key: 'id'

        }

    }

}, { timestamps: true })

export const Rating = sequelize.define('Rating', {

    userId: {

        type: DataTypes.INTEGER,

        allowNull: false,

        references: {

            model: User,

          key: 'id'

        }

    },

    gymId: {

        type: DataTypes.INTEGER,

        allowNull: false,

        references: {

            model: Gym,

            key: 'id'

        }

    },

    rating: {

        type: DataTypes.FLOAT,

        allowNull: false,

    },

    comment: {

        type: DataTypes.STRING,

        allowNull: true

    }

})

User.hasMany(Rating, { foreignKey: 'userId' })

Rating.belongsTo(User, { foreignKey: 'userId' })

Gym.hasMany(Rating, { foreignKey: 'gymId' })

Rating.belongsTo(Gym, { foreignKey: 'gymId' })

User.hasMany(Saved, { foreignKey: 'userId' })

Saved.belongsTo(User, { foreignKey: 'userId' })

Gym.hasMany(Saved, { foreignKey: 'gymId' })

Saved.belongsTo(Gym, { foreignKey: 'gymId' })

User.hasMany(Notification, { foreignKey: 'userId' })

Notification.belongsTo(User, { foreignKey: 'userId' })

Gym.hasMany(Notification, {foreignKey: 'senderId'})

Notification.belongsTo(Gym, {foreignKey: 'senderId'})

Gym.hasMany(GymImages, { foreignKey: 'gymId' })

GymImages.belongsTo(Gym, { foreignKey: 'gymId' })

Gym.hasOne(GymOpeningHours, { foreignKey: 'gymId' })

GymOpeningHours.belongsTo(Gym, { foreignKey: 'gymId' })

Gym.hasMany(GymWorkoutMapping, { foreignKey: 'gymId' })

GymWorkoutMapping.belongsTo(Gym, { foreignKey: 'gymId' })

GymWorkout.hasMany(GymWorkoutMapping, { foreignKey: 'gymWorkoutId' })

GymWorkoutMapping.belongsTo(GymWorkout, { foreignKey: 'gymWorkoutId' })

Gym.hasMany(GymFeatureMapping, { foreignKey: 'gymId' })

GymFeatureMapping.belongsTo(Gym, { foreignKey: 'gymId' })

GymFeature.hasMany(GymFeatureMapping, { foreignKey: 'gymFeatureId' })

GymFeatureMapping.belongsTo(GymFeature, { foreignKey: 'gymFeatureId' })

Gym.hasMany(GymRule, { foreignKey: 'gymId' })

GymRule.belongsTo(Gym, { foreignKey: 'gymId' })

Gym.hasOne(GymLocation, { foreignKey: 'gymId' })

GymLocation.belongsTo(Gym, { foreignKey: 'gymId' })

Gym.hasMany(Plan, { foreignKey: 'gymId' })

Plan.belongsTo(Gym, { foreignKey: 'gymId' })

**User.js**

import { Sequelize, DataTypes } from "sequelize";

import sequelize from "../config/db.js";

// Define User Model

const UserRole = sequelize.define('UserRole', {

    roleName: {

        type: DataTypes.STRING,

        allowNull: false

    },

    roleDescription: {

        type: DataTypes.STRING,

        allowNull: true

    }

})

const User = sequelize.define('User', {

    firstName: {

        type: DataTypes.STRING,

        allowNull: false

    },

    lastName: {

        type: DataTypes.STRING,

        allowNull: true

    },

    email: {

        type: DataTypes.STRING,

        allowNull: false,

        unique: true

    },

    password: {

        type: DataTypes.STRING,

        allowNull: true

    },

    imgPath:{

        type: DataTypes.STRING,

        allowNull: true

    },

    googleId: {

        type: DataTypes.STRING,

        allowNull: true

    },

    roleId: {

        type: DataTypes.INTEGER,

        references: {

            model: UserRole,

            key: 'id'

        }

    },

    status: {

        type: DataTypes.STRING,

        allowNull: false

    }

}, timestamps: true,})

UserRole.hasMany(User, { foreignKey: 'roleId' })

User.belongsTo(UserRole, { foreignKey: 'roleId' })

export { User, UserRole };

* + 1. **Controllers**

**gymController.js**

import { Gym, GymFeature, GymWorkout, GymWorkoutMapping, GymOpeningHours, GymLocation, GymImages, Rating } from "../models/Gym.js";

import { User } from "../models/User.js";

import { Plan } from "../models/Gym.js";

import sequelize from "../config/db.js";

import { GymFeatureMapping } from "../models/Gym.js";

import { Op, where } from "sequelize";

import multer from 'multer';

import path from 'path';

import fs from 'fs'; // To check if the folder exists

import { PlanMapping } from "../models/Plans.js";

import { encrypt } from "./paymentGateway.js";

import { insertApiKey } from "./paymentGateway.js";

import { Saved } from "../models/Gym.js";

import { Notification } from "../models/Gym.js";

import { insertPlanMappingDraft } from "./planController.js";

import { Transaction } from "../models/Transaction.js";

const storage = multer.diskStorage({

    destination: (req, file, cb) => {

        const uploadDir = 'uploads/';

        if (!fs.existsSync(uploadDir)) {

            fs.mkdirSync(uploadDir);

        }

        cb(null, uploadDir);

    },

    filename: (req, file, cb) => {

        cb(null, Date.now() + path.extname(file.originalname));

    }

});

const upload = multer({

    storage: storage,

    fileFilter: (req, file, cb) => {

        const allowedTypes = ['image/jpeg', 'image/png'];

        if (!allowedTypes.includes(file.mimetype)) {

            return cb(new Error('Only JPG and PNG files are allowed'), false);

        }

        cb(null, true);

    }

}).fields([

    { name: 'gymImages', maxCount: 5 },

    { name: 'gymProfileImage', maxCount: 1 }

]);

export const changeGymInfo = async (req, res) => {

    try {

        const { gymId } = req.params

        const { phone, email } = req.body

        const updateGym = await Gym.update({

            gymPhone: phone,

            gymEmail: email,

        }, { where: { id: gymId } })

        res.json(updateGym)

    } catch (err) {

        console.log(err)

        res.json(err)

    }

}

export const countData = async(req, res)=>{

    try{

        const {gymId} = req.params

        const profits = await Transaction.sum('amount',{where: {receiverId: gymId, status: 'success'}})

        const membershipPlans = await Plan.count({where: {gymId}})

        const activeMembers = await PlanMapping.count({where: {gymId}})

        res.json({profits, membershipPlans, activeMembers})

    }catch(err){

        console.log(err)

        res.json(err)

    }

}

export const setGymVerified = async (req, res) => {

    try {

        const { gymId } = req.body;

        const updateGym = await Gym.update({ status: 'verified' }, { where: { id: gymId } })

        res.json(updateGym)

    } catch (err) {

        console.log(err)

        res.json(err)

    }

}

export const updateGymAdminWorkout = async (req, res) => {

    try {

        const { gymWorkoutId, gymId } = req.params

        const check = await GymWorkoutMapping.findOne({ where: { gymWorkoutId } })

        if (check) {

            const delWorkout = GymWorkoutMapping.destroy({ where: { gymWorkoutId } })

            return res.json(delWorkout)

        }

        const addWorkout = await  GymWorkoutMapping.create({

            gymId,

            gymWorkoutId

        })

        res.json(addWorkout)

    }catch(err){

        console.log(err)

        res.json(err)

    }

}

export const addMembershipPlan = async(req, res)=>{

    try{

        const {planName, planDescription, price, version} = req.body

        const {gymId} = req.params

        const add = await Plan.create({

            planName,

            planDescription,

            planType: 'membership',

            price,

            version,

            gymId,

            state: 'active'  })

        res.json(add)

    }catch(err){

        console.log(err)

        res.json(err)

    }

}

export const updateGymAdminFeature = async (req, res) => {

    try {

        const { gymFeatureId, gymId } = req.params

        const check = await GymFeatureMapping.findOne({ where: { gymFeatureId } })

        if (check) {

            const delWorkout = GymFeatureMapping.destroy({ where: { gymFeatureId } })

            return res.json(delWorkout)

        }

        const addFeature = await  GymFeatureMapping.create({

            gymId,

            gymFeatureId

        })

        res.json(addFeature)

    }catch(err){

        console.log(err)

        res.json(err)

    }

}

export const changeGymPp = async (req, res) => {

    const { gymId } = req.params;

    const pp = req.files.pp || [];

    const ppPath = pp.map(img => `/uploads/${img.filename}`);

    console.log("Image Paths: ", ppPath);

    const results = [];

    for (const img of ppPath) {

        try {

            console.log(img);

            await Gym.update(

                { profileImage: img },

                { where: { id: gymId } }

            );

            console.log(`PP ${img} added successfully.`);

            results.push({ img, status: 'success' });

        } catch (error) {

            console.error(`Error adding feature ${img}:`, error);

            results.push({ img, status: 'error', error });

        }

    }

    res.json({ message: 'Process completed', results });

};

export const deleteGymImage = async (req, res) => {

    try {

        const { imageId, gymId } = req.params

        const checkImage = await GymImages.count({ where: { gymId } })

        console.log("check image: ", checkImage)

        if (checkImage === 1) {

            return res.status(400).json({ messageDel: 'Cannot Delete!! There should be atleast 1 image' })

        }

        const delImage = await GymImages.destroy({ where: { id: imageId } })

        res.json(delImage)

    } catch (err) {

        console.log(err)

        res.json(err)

    }

}

export const addGymImage = async (req, res) => {

    try {

        const { gymId } = req.body

        const gymImages = req.files.gymImages || [];

        const gymImagePaths = gymImages.map(img => `/uploads/${img.filename}`);

        console.log("Image Paths: ", gymImagePaths, " ");

        for (const img of gymImagePaths) {

            try {

                console.log(img)

                await GymImages.create({

                    gymId,

                    gymImgPath: img

                });

                console.log(`Workout ${img} added successfully.`);

            } catch (error) {

                console.error(`Error adding feature ${img}:`, error);

            }

        }

        res.json({ message: 'successfull' })

    } catch (err) {

        console.log(err)

        res.json(err)

    }

}

export const insertGym = async (req, res) => {

    try {

               console.log("hello")

        const gymData = await JSON.parse(req.body.gymData);

        console.log("PLAN ID: ", req.body.planId)

        console.log(gymData)

        const gymImages = req.files.gymImages || [];

        const gymProfileImage = req.files.gymProfileImage ? req.files.gymProfileImage[0] : null;

        const gymLicense = req.files.gymLicense ? req.files.gymLicense[0] : null;

        const gymImagePaths = gymImages.map(img => `/uploads/${img.filename}`);  // Store paths of gym images

        const gymProfileImagePath = gymProfileImage ? `/uploads/${gymProfileImage.filename}` : null;  // Store profile image path

        const gymLicensePath = gymLicense ? `/uploads/${gymLicense.filename}` : null;

        console.log("Image Paths: ", gymImagePaths, " ", gymProfileImagePath);

        console.log("gym license path: ", gymLicensePath)

        const newGym = await Gym.create({

            gymName: gymData.name,

            gymPhone: gymData.contact,

            gymEmail: gymData.email,

            gymLicense: gymLicensePath,

            ownerId: req.body.ownerId,

            profileImage: gymProfileImagePath,

            status: 'unverified'

        })

        if (!newGym) {

            res.status(400).json({ message: 'error creating gym' })

        }

        const gymId = newGym.id;

        await insertPlanMappingDraft(req.body.ownerId, req.body.planId, gymId)

        const newPlan1 = await Plan.create({

            planName: gymData.membershipPlans.plan1.title,

            planDescription: gymData.membershipPlans.plan1.description,

            duration: gymData.membershipPlans.plan1.duration,

            price: gymData.membershipPlans.plan1.price,

            planType: 'membership',

            gymId: gymId,

            state: 'active'

        })

        if (!newPlan1) {

            res.status(400).json({ message: 'error creating plan 1' })

        }

        const plan2 = gymData.membershipPlans.plan2;

        if (!plan2.title || !plan2.description || !plan2.duration || !plan2.price) {

            // return res.json({ message: 'fill the others' })

        } else {

            const newPlan2 = await Plan.create({

                planName: plan2.title,

                planDescription: plan2.description,

                duration: plan2.duration,

                price: plan2.price,

                planType: 'membership',

                gymId: gymId,

                state: 'active'

            })

        }

        const plan3 = gymData.membershipPlans.plan3;

        if (!plan3.title || !plan3.description || !plan3.duration || !plan3.price) {

            // return res.json({ message: 'fill the others' })

        }

        else {

            const newPlan3 = await Plan.create({

                planName: plan3.title,

                planDescription: plan3.description,

                duration: plan3.duration,

                price: plan3.price,

                planType: 'membership',

                gymId: gymId,

                state: 'active'

            })

        }

        const features = gymData.features

        for (const feature of features) {

            try {

                console.log(feature)

                await GymFeatureMapping.create({

                    gymId: gymId,

                    gymFeatureId: feature

                });

                console.log(`Feature ${feature} added successfully.`);

            } catch (error) {

                console.error(`Error adding feature ${feature}:`, error);

            }

        }

        const workouts = gymData.workouts

        for (const workout of workouts) {

            try {

                console.log(workout)

                await GymWorkoutMapping.create({

                    gymId: gymId,

                    gymWorkoutId: workout

                });

                console.log(`Workout ${workout} added successfully.`);

            } catch (error) {

                console.error(`Error adding feature ${workout}:`, error);

            }

        }

        const morning = `${gymData.openingHours.morning.start}am - ${gymData.openingHours.morning.end}am`

        const evening = `${gymData.openingHours.evening.start}pm - ${gymData.openingHours.evening.end}pm`

        const newOpeningHour = await GymOpeningHours.create({

            gymId,

            morning,

            evening

        })

        const location = gymData.location

        const newLocation = await GymLocation.create({

            buildingNo: location.buildingNo,

            latitude: location.lat,

            longitude: location.lng,

            area: location.area,

            city: location.city,

            landmark: location.landmark,

            gymId

        })

        for (const img of gymImagePaths) {

            try {

                console.log(img)

                await GymImages.create({

                    gymId,

                    gymImgPath: img

                });

                console.log(`Workout ${img} added successfully.`);

            } catch (error) {

                console.error(`Error adding feature ${img}:`, error);

            }

        }

        const enApi = encrypt(gymData.publicKey, gymData.secretKey)

        insertApiKey(enApi.encryptedData1, enApi.encryptedData2, enApi.key, enApi.iv, gymId)

        res.json({ message: 'successful operation' })

    } catch (err) {

        console.error(err);

        res.status(500).json({ error: "Failed to register gym", details: err.message });

    }

};

export const getGyms = async (req, res) => {

    try {

        console.log("get gyms: ", req.query)

        const page = parseInt(req.query.page, 10) || 1;

        const limit = parseInt(req.query.limit, 10) || 5;

        const offset = (page - 1) \* limit;

        const location = req.query.location

        const totalGyms = await Gym.count({

            where: { status: 'verified' },

            include: [

                {

                    model: GymLocation,

                    where: { city: location, }

                },

                {

                    model: PlanMapping,

                    required: true, // optional: allow gyms without plans

                    where: { status: 'active' },

                    order: [['createdAt', 'DESC']],

                },

                { model: Plan }

            ]

        });

        const gyms = await Gym.findAll({

            limit,

            offset,

            where: { status: 'verified' },

            include: [

                {

                    model: GymLocation,

                    where: { city: location }

                },

                {

                    model: PlanMapping,

                    required: true, // optional: allow gyms without plans

                    where: { status: 'active' },

                    order: [['createdAt', 'DESC']],

                },

                { model: Plan }

            ]

        });

        console.log("gyms: ", gyms)

        const totalPages = Math.ceil(totalGyms / limit);

        const currentPage = page;

        res.status(200).json({ gyms, totalPages, currentPage, totalGyms });

    } catch (err) {

        console.log(err)

        res.status(500).json(err);

    }

};

export const getNotif = async (req, res) => {

    try {

        console.log("hello fro get notif", req.params)

        const { userId } = req.params

        const notif = await Notification.findAll({

            where: { userId }

        })

        console.log(notif)

        res.json(notif)

    } catch (err) {

        console.log(err)

        res.status(500).json(err)

    }

}

export const getSaved = async (req, res) => {

    try {

        console.log(req.params)

        const { userId } = req.params;

        const userSaved = await Saved.findAll(

            {

                where: { userId }

            }

        )

        res.json(userSaved)

    } catch (err) {

        res.status(500).json(err)

    }

}

export const getUserSavedGyms = async (req, res) => {

    try {

        const { userId } = req.params;

        const savedGyms = await Gym.findAll({

            include: [

                { model: GymLocation },

                {

                    model: Saved,

                    where: { userId }

                },

                { model: Plan }

            ]

        });

        res.json(savedGyms)

    } catch (err) {

        res.status(500).json(err)

    }

}

export const saveGym = async (req, res) => {

    const { gymId, userId } = req.body;

    console.log("from save gym: ", gymId, userId);

    try {

        const check = await Saved.findOne({

            where: {

                userId,

                gymId

            }

        });

        if (check) {

            await Saved.destroy({

                where: {

                    id: check.id

                }

            });

            res.json({ message: 'deleted' });

        } else {

            const newSaved = await Saved.create({

                userId,

                gymId

            });

            res.json({ newSaved, message: 'saved' });

        }

    } catch (err) {

        console.log(err);

        res.status(500).json({ message: 'An error occurred' });

    }

};

export const getFeatures = async (req, res) => {

    try {

        const features = await GymFeature.findAll();

        if (!features) {

            res.status(400).json({ message: 'Features not available' })

        }

        res.status(200).json(features)

    } catch (error) {

        res.status(400).json(error)

    }

}

export const getWorkouts = async (req, res) => {

    try {

        const workouts = await GymWorkout.findAll();

        if (!workouts) {

            res.status(400).json({ message: 'Workouts not available' })

        }

        res.status(200).json(workouts)

    } catch (error) {

        res.status(400).json(error)

    }

}

* 1. **Web**

**tailwind.config.js**

/\*\* @type {import('tailwindcss').Config} \*/

export default {

  content: [

    "./index.html",

    "./src/\*\*/\*.{vue,js,ts,jsx,tsx}",

  ],

  theme: {

    extend: {

      screens:{

        'smartphone': '320px',

        'smartphone-md': '375px',

        'smartphone-lg': '425px',

        'smartphone-landscape': '480px',

        'smartphone-xl': '600px',

        'tablet': '768px',

        'laptop': '1024px',

        'desktop': '992px'

      },

      colors: {

        'college-blue': '#016DBC',

        'admin-pink': '#FFC6D3',

        'admin-light-pink': '#fff3f9',

        'college-white': "#FFFFFF",

        'hover-blue': "#0d7ecf",

        'college-black': "#000000",

        'college-grey': "rgb(202, 202, 202)"

      },

      boxShadow: {

        'right': '0px 0 3px rgba(0, 0, 0, 0.1)'

      }

    },

  },

  plugins: [],

}

**main.js**

import './assets/style.css'

import { createPinia } from 'pinia';

import axios from 'axios';

import { createApp } from 'vue'

import App from './App.vue'

import router from "./router";

import { MotionPlugin } from '@vueuse/motion';

axios.defaults.baseURL = 'http://localhost:3000';

const app = createApp(App);

const pinia = createPinia();

app.use(router);

app.use(pinia)

app.use(MotionPlugin);

app.mount('#app')

* + 1. **Views**

**HomeView.vue**

<script setup>

import { ref, onMounted, watch } from 'vue';

import Hero from '../../components/Hero.vue'

import Services from '../../components/Services.vue'

import Action from '../../components/Action.vue'

import Mobile from '../../components/Mobile.vue'

import { useGymStore } from '../../stores/gyms'

import { useHomeStore } from '../../stores/home'

import { useRoute } from 'vue-router';

import { storeToRefs } from 'pinia'

const route = useRoute();

const home = useHomeStore()

const gymStore = useGymStore();

const { gyms } = storeToRefs(gymStore)

const { getGyms } = gymStore;

const { isOpenLogin } = storeToRefs(home)

watch(

    () => route.query.showLoginModal,

    (newVal) => {

        if (newVal === 'true') {

            isOpenLogin.value = true;

        }}

);

onMounted(async () => {

    if (route.query.showLoginModal === 'true') {

        isOpenLogin.value = true;

    }

});

</script>

<template>

    <section class="bg-white">

        <Hero />

        <Services />

        <Action />

        <Mobile />

    </section>

</template>

<style scoped>

.hero {

    background: url(../../assets/images/wallpaper1.webp);

    background-size: cover;

    background-position: 50% 20%;

    background-repeat: no-repeat;

}

</style>

**RegisterGymView.vue**

<script setup>

import Hero from '../../components/Hero.vue';

import { storeToRefs } from 'pinia';

import NavbarSecond from '../../components/Navbars/NavbarSecond.vue';

import { reactive, ref, onMounted } from 'vue';

import L from 'leaflet'; // Import Leaflet

import Button from '../../components/Button.vue';

import ButtonLink from '../../components/ButtonLink.vue';

import { usePlanStore } from '../../stores/plans'

import { useGymRegStore } from '../../stores/gymReg'

import { useTokenStore } from '../../stores/token'

// import GMap from '@/components/GMap.vue'

const token = useTokenStore()

const { decodeToken, fetchUser } = token

const gymReg = useGymRegStore()

const { formData, handleRegister, handleGymImages, handleGymProfileImage, handleGymLicense, toggleFeature, toggleWorkout, fetchFeatures, fetchWorkouts } = gymReg;

const { message, isFormValid, allFeatures, allWorkouts } = storeToRefs(gymReg)

const plan = usePlanStore()

const { getSubscriptionPlanById } = plan;

const { subPlan } = storeToRefs(plan)

const props = defineProps({

    planId: {

        type: String,

        required: true

    },

    planName: {

        type: String,

        required: true

    }

})

const map = ref(null);

const mapContainerId = 'map'; let marker;

</script>

<template>

    <NavbarSecond />

    <section class="w-full flex  flex-col items-center justify-center px-4 laptop:px-10 pb-30 reg bg-white">

        <div class="h-auto w-[100%] md:w-[70%] lg:w-[55%] py-5 ">

            <form class="mx-auto flex flex-col space-y-5" @submit.prevent="handleRegister(props.planId)">

                <div class=" flex flex-col space-y-5" v-show="step === 2">

                    <h1 class="font-semibold mb-3 text-black text-2xl " v-motion-fade-visible-once>Other Gym Details</h1>

                    <div class="bg-white rounded-xl p-6 shadow-6" v-motion-fade-visible-once>

                        <h1 class="font-semibold  text-black text-2xl ">Add Gym Images\*</h1>

                        <p class="text-sm mb-2">Upload atleast one image of your gym</p>

                        <span class="text-red text-sm mb-3" v-if="message.gymImages">{{ message.gymImages }}</span>

                        <div class="flex items-center w-full bg-forth p-4">

                                <input id="dropzone-file-gym-images" type="file" class="border-none" multiple name="gym-images"

                                    @change="handleGymImages" required/>

                        </div>

                    </div>

                    <div class="bg-white rounded-xl p-6 shadow-6" v-motion-fade-visible-once>

                        <h1 class="font-semibold  text-black text-2xl ">Add Gym Profile Image\*</h1>

                        <p class="text-sm mb-3">Upload atleast one image of your gym</p>

                        <span class="text-red text-sm mb-3" v-if="message.gymProfileImage">{{ message.gymProfileImage

                        }}</span>

                        <div class="flex items-center w-full bg-forth p-4">

                                <input id="dropzone-file-profile-image" type="file" class="border-none"  name="gym-profile-image"

                                    @change="handleGymProfileImage" required/>

                        </div>

                    </div>

                    <div class="bg-white rounded-xl p-6 shadow-6" v-motion-fade-visible-once>

                        <h1 class="font-semibold  text-black text-2xl ">Add Gym License\*</h1>

                        <p class="text-sm mb-3">Upload gym license(Only PDF is accepted)</p>

                        <span class="text-red text-sm mb-3" v-if="message.gymLicense">{{ message.gymLicense

                        }}</span>

                        <div class="flex items-center w-full bg-forth p-4">

                                <input id="dropzone-file-profile-image" type="file" class="border-none"  name="gym-profile-image"        @change="handleGymLicense" required/>

                        </div>

                    </div>

                    <div class="bg-white rounded-xl p-6 shadow-6" v-motion-fade-visible-once>

                        <h1 class="font-semibold  text-black text-2xl ">Opening Hours\*</h1>

                        <p class="text-sm mb-3">Add opening hours of your gym in during morning and eving</p>

                        <h1 class="font-semibold  text-black text-xl mb-3">Morning</h1>

                        <div class="grid md:grid-cols-2 md:gap-6">

                            <div class="relative z-0 w-full mb-5 group">

                                <input v-model="formData.openingHours.morning.start" type="time" name="floating\_first\_name"

                                    id="morning-start"

                                    placeholder=" " required />

                                <label for="morning-start"

                                 Start time\*</label>

                            </div>

                    </div>

                    <div class="bg-white rounded-xl p-6 shadow-6" v-motion-fade-visible-once>

                        <h1 class="font-semibold  text-black text-2xl ">Razorpay API Key\*</h1>

                        <p class="text-sm mb-3">Enter Razorpay API keys to receive payment(Don't worry, your API keys are

                            safe with us)</p>

                        <div class="grid md:grid-cols-2 md:gap-6">

                            <div class="relative z-0 w-full mb-5 group">

                                <input v-model="formData.publicKey" type="password" name="ownerName" id="ownerName"

                                    placeholder=" " required />

                                <label for="ownerName"

                                   API

                                    Public Key\*</label>

                            </div>

                            <div class="relative z-0 w-full  group">

                                <input v-model="formData.secretKey" type="password" name="email" id="email"

                                    placeholder=" " required />

                                <label for="email"

                                    class=API

                                    Secret Key\*</label>

                                <span class="text-red text-sm ">{{ message.email }}</span>

                            </div>

                        </div>

                    </div>

                    <div class="bg-white rounded-xl p-6 shadow-6" v-motion-fade-visible-once>

                        <h1 class="font-semibold  text-black text-2xl ">Membership Plan</h1>

                        <p class="text-sm mb-3">Add Membership plans offered by your gym(More Plans can be added later)</p>

                        <h1 class="font-semibold  text-black text-xl mb-3">Plan-1</h1>

                        <div class="grid md:grid-cols-2 md:gap-6">

                            <div class="relative z-0 w-full mb-5 group">

                                <input v-model="formData.membershipPlans.plan1.title" type="text" name="plan1-title" id="plan1-title"

                                    placeholder=" " required />

                                <label for="plan1-title"

                                   Plan

                                    Title\*</label>

                            </div>

                            <div class="relative z-0 w-full mb-5 group">

                                <input v-model="formData.membershipPlans.plan1.price" type="number" name="plan1-price"

                                    id="plan1-price"

                                    placeholder=" " required />

                                <label for="plan1-price"

                                   Price\*</label>

                            </div> </div>

                        <div class="relative z-0 w-full mb-5 group">

                            <input v-model="formData.membershipPlans.plan1.duration" type="number" name="plan1-duration"

                                id="plan1-duration"

                                placeholder=" " />

                        </div><div class="bg-white rounded-xl p-6 shadow-6" v-motion-fade-visible-once><h1 class="font-semibold  text-black text-2xl ">Workouts</h1>

<p class="text-sm mb-3">Select workouts available in your gym</p><div class="flex flex-wrap space-x-2 space-y-2">  <span class="border p-2 py-1 rounded-xl hover:cursor-pointer transition-all duration-100 ease-in"

  v-for="w in allWorkouts" :key="w.id" @click="toggleWorkout(w.id)" :class="{

                                    'bg-first text-white': formData.workouts.includes(w.id), 'text-black': !formData.workouts.includes(w.id) //

                                }">

                                {{ w.workoutName }}

                            </span</div></div><div class="bg-white rounded-xl p-6 shadow-6" v-motion-fade-visible-once><Button content="Submit" buttonType="submit" /></div>

**GymAdminDashboard.vue**

<script setup>

import Button from '@/components/Button.vue'

import {ref, onMounted} from 'vue'

import { storeToRefs } from 'pinia'

import { useGymAdminStore } from '@/stores/gymAdmin'

import GymAdminMyGymPopups from '@/components/GymAdminMyGymPopups.vue'

const gymAdmin = useGymAdminStore()

const { gymIdRef, count } = storeToRefs(gymAdmin)

const {countData, formatAmount} = gymAdmin

onMounted(()=>{

    countData(gymIdRef.value)

})

</script>

<template>

    <div class="grid grid-cols-1 gap-4 md:grid-cols-2 md:gap-6 xl:grid-cols-4 2xl:gap-7.5">

        <div

            class="rounded-sm border border-stroke bg-white py-6 px-7.5 shadow-default dark:border-strokedark dark:bg-boxdark">

            <div class="flex h-11.5 w-11.5 items-center justify-center rounded-full bg-meta-2 dark:bg-meta-4"><i class="fa-solid fa-money-bill text-white"></i>

            </div>

            <div class="mt-4 flex items-end justify-between">

                <div>

                    <h4 class="text-title-md font-bold text-black dark:text-white">₹{{  formatAmount(count.profits) }}</h4>

                    <span class="text-sm font-medium">Total Profits</span>

                </div>

                <!-- <span class="flex items-center gap-1 text-sm font-medium">

                    sdgsdg

                </span> -->

            </div>

        </div>

        <div

            class="rounded-sm border border-stroke bg-white py-6 px-7.5 shadow-default dark:border-strokedark dark:bg-boxdark">

            <div class="flex h-11.5 w-11.5 items-center justify-center rounded-full bg-meta-2 dark:bg-meta-4">

                <i class="fa-solid fa-money-check-dollar"></i>

            </div>

            <div class="mt-4 flex items-end justify-between">

                <div>

                    <h4 class="text-title-md font-bold text-black dark:text-white">{{  formatAmount(count.membershipPlans) }}</h4>

                    <span class="text-sm font-medium">Total Membership Plans</span>

                </div>

            </div>

        </div>

        <div

            class="rounded-sm border border-stroke bg-white py-6 px-7.5 shadow-default dark:border-strokedark dark:bg-boxdark">

            <div class="flex h-11.5 w-11.5 items-center justify-center rounded-full bg-meta-2 dark:bg-meta-4"><i class="fa-solid fa-users "></i>

            </div>

            <div class="mt-4 flex items-end justify-between">

                <div>

                    <h4 class="text-title-md font-bold text-black dark:text-white">{{  formatAmount(count.activeMembers) }}</h4>

                    <span class="text-sm font-medium">Total Active Members</span>

                </div>

            </div>

        </div>

    </div>

</template>

**AdminGymView.vue**

<script setup >

import { ref, watch } from 'vue'

import { storeToRefs } from 'pinia'

import AdminGymPopups from '@/components/AdminGymPopups.vue'

import { useGymStore } from '@/stores/gyms'

import { useAdminStore } from '@/stores/admin'

const admin = useAdminStore();

const { isOpen } = storeToRefs(admin)

const { closeModals, activateGymDetails } = admin

const gymStore = useGymStore()

const { getAllGyms } = gymStore;

const { allGyms, gymSearch, totalPages, currentPage } = storeToRefs(gymStore)

getAllGyms()

watch(gymSearch, (newVal) => {

    getAllGyms(newVal)

})

</script>

<template>

    <section class="flex flex-col">

        <AdminGymPopups />

        <div

            class="p-5 bg-gray-100 rounded-sm border border-stroke bg-white px-5 pt-6 pb-2.5 shadow-default dark:border-strokedark dark:bg-boxdark sm:px-7.5 xl:pb-1">

            <h1 class="text-xl mb-2">Gyms</h1>

            <div class="flex justify-between">

                <div>

                    <i class="fa-solid fa-magnifying-glass"></i>

                    <input type="text" name="" id="" class="bg-transparent p-2 outline-none" placeholder="Search Gyms"

                        v-model="gymSearch" />

                </div>

                <span>

                    <span></span>

                </span>

            </div>

            <div class="overflow-auto rounded-lg shadow hidden md:block">

                <table class="w-full">

                    <thead class="bg-gray-50 border-b-2 border-gray-200">

                        <tr>

                            <th class="w-20 p-3 text-sm font-semibold tracking-wide text-left">ID</th>

                            <th class="p-3 w-24 text-sm font-semibold tracking-wide text-left">Name</th>

                            <th class="w-24 p-3 text-sm font-semibold tracking-wide text-left">Email</th>

                            <th class="w-24 p-3 text-sm font-semibold tracking-wide text-left">Phone</th>

                            <th class="w-24 p-3 text-sm font-semibold tracking-wide text-left">Status</th>

                            <th class=" p-3 text-sm font-semibold tracking-wide text-left">Owner</th>

                            <th class="w-12 p-3 text-sm font-semibold tracking-wide text-left"></th>

                            <th class="w-12 p-3 text-sm font-semibold tracking-wide text-left"></th>

                            <th class="w-0 p-3 text-sm font-semibold tracking-wide text-left"></th>

                        </tr>

                    </thead>

                    <tbody class="divide-y divide-gray-100">

                        <tr class="" v-for="gym in allGyms" :key="gym.id">

                            <td class="p-3 text-sm text-gray-700 whitespace-nowrap">

                                <a href="#" class="font-bold text-blue-500 hover:underline">{{ gym.id }}</a>

                            </td>

                            <td class="p-3 text-sm text-gray-700 whitespace-nowrap">

                                {{ gym.gymName }}

                            </td>

                            <td class="p-3 text-sm text-gray-700 whitespace-nowrap">{{ gym.gymEmail }}</td>

                            <td class="p-3 text-sm text-gray-700 whitespace-nowrap">{{ gym.gymPhone }}</td>

                            <td class="p-3 text-sm text-gray-700 whitespace-nowrap">

                                <span

                                    class="p-1.5 text-xs font-medium uppercase tracking-wider text-grey  rounded-lg bg-opacity-50"

                                    :class="gym.status == 'unverified' ? 'bg-orange-600' : 'bg-green-400'">{{ gym.status

                                    }}</span>

                            </td>

                            <td class="p-3 text-sm text-gray-700 whitespace-nowrap">{{ gym.User.firstName }} {{

                                gym.User.lastName }}</td>

                            <td class="p-3 text-sm text-gray-700 whitespace-nowrap"><i

                                    @click="activateGymDetails(gym, gym.ownerId)"

                                    class="fa-solid fa-circle-info text-lg cursor-pointer"></i></td>

                            <!-- <td class="p-3 text-sm text-gray-700 whitespace-nowrap"><i class="fa-solid fa-circle-info text-lg"></i></td> -->

                            <td class="p-3 text-sm text-gray-700 whitespace-nowrap"><i

                                    class="fa-solid fa-delete-left text-lg cursor-pointer"></i></td>

                        </tr>

                    </tbody>

                </table>

            </div>

            <div class="grid grid-cols-1 sm:grid-cols-2 md:hidden gap-2">

                <div class=" space-y-3 p-4 rounded-lg shadow" v-for="gym in allGyms" :key="gym.id">

                    <div class="flex items-center space-x-2 text-sm">

                        <div>

                            <a href="#" class="text-blue-500 font-bold hover:underline">{{ gym.id }}</a>

                        </div>

                        <div class="text-gray-500">{{ gym.gymName }}</div>

                        <div>

                            <span

                                class="p-1.5 text-xs font-medium uppercase tracking-wider text-grey  rounded-lg bg-opacity-50"

                                :class="gym.status == 'unverified' ? 'bg-orange-500' : 'bg-green-200'">{{ gym.status

                                }}</span>

                        </div>

                    </div>

                    <div class="text-sm text-gray-700">

                        {{ gym.gymEmail }}

                    </div>

                    <div class="flex justify-between">

                        <div class="text-sm font-medium text-gray-700">

                            {{ gym.gymPhone }}

                        </div>

                        <div class="text-sm font-medium text-gray-700 flex items-center space-x-3">

                            <i class="fa-solid fa-circle-info text-lg cursor-pointer"></i>

                            <i class="fa-solid fa-delete-left text-lg cursor-pointer"></i>

                        </div>

                    </div>

                </div>

            </div>

            <div class="flex justify-center">

                <nav aria-label="Page navigation example" class="my-2">

                    <ul class="flex items-center -space-x-px h-8 text-sm">

                        <li>

                            <button :disabled="currentPage === 1"

                                class="flex items-center justify-center px-3 h-8 leading-tight text-gray-500 bg-white border border-gray-300 rounded-s-lg hover:bg-gray-100 hover:text-gray-700 "

                                @click="getAllGyms('', currentPage - 1, 10, true)">

                                <span class="sr-only">Next</span>

                                <svg class="w-2.5 h-2.5 rtl:rotate-180" aria-hidden="true"

                                    xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 6 10">

                                    <path stroke="currentColor" stroke-linecap="round" stroke-linejoin="round"

                                        stroke-width="2" d="M5 1 1 5l4 4" />

                                </svg>

                            </button>

                        </li>

                        <li v-for="page in totalPages" @click="getAllGyms('', page, 10, true)" :key="page">

                            <a href="#"

                                class="flex items-center justify-center px-3 h-8 leading-tight text-gray-500  border border-gray-300 hover:bg-gray-100 hover:text-gray-700 "

                                :class="currentPage == page ? 'bg-gray-200' : 'bg-white'">{{

                                    page }} </a>

                        </li>

                        <li>

                            <button :disabled="currentPage === totalPages"

                                class="flex items-center justify-center px-3 h-8 leading-tight text-gray-500 bg-white border border-gray-300 rounded-e-lg hover:bg-gray-100 hover:text-gray-700 "

                                @click="getAllGyms('', currentPage + 1, 10, true)">

                                <span class="sr-only">Next</span>

                                <svg class="w-2.5 h-2.5 rtl:rotate-180" aria-hidden="true"

                                    xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 6 10">

                                    <path stroke="currentColor" stroke-linecap="round" stroke-linejoin="round"

                                        stroke-width="2" d="m1 9 4-4-4-4" />

                                </svg>

                            </button>

                        </li>

                    </ul>

                </nav>

            </div>

        </div>

    </section></template>

**AdminLoginView.vue**

<script setup>

import { storeToRefs } from 'pinia';

import { RouterLink } from 'vue-router';

import LoaderSpinner from '../components/LoaderSpinner.vue'

import { useAuthStore } from '../stores/auth';

import { ref } from 'vue';

import ErrorPage from '../components/ErrorPage.vue'

import { useErrorPage } from '../stores/errorPage'

const errorPage = useErrorPage();

const { isError } = storeToRefs(errorPage);

let flag = ref(false);

let passwordType = ref("password");

const authStore = useAuthStore();

const { loginEmail,

    loginPassword,

    isLoggedIn, errMessage,

    token, } = storeToRefs(authStore);

const { login, logout } = authStore;

const togglePasswordType = () => {

    flag.value = !flag.value;

    if (flag.value) { passwordType.value = "text"; }

    else {

        passwordType.value = "password";

    }

}

</script>

<template>

    <section v-motion-fade-visible-once

        class="px-4 sm:px-6 md:px-8 lg:px-16 xl:px-24 py-10 w-full min-h-screen flex justify-center items-center main-container"

        v-if="!isError">

        <form @submit.prevent="login" class="w-full max-w-md" v-motion-fade-visible-once>

            <div class="bg-white shadow-lg  p-8 flex flex-col justify-center items-center">

                <img src="../images/logo 2.png" alt="Logo" class="w-40 mb-6" />

                <h1 class="text-2xl font-bold mb-6">ADMIN LOGIN</h1>

  <div class="w-full space-y-4">

                    <div>

                        <label for="email" class="block text-sm font-bold mb-1">Email</label>

                        <input type="email" name="email" id="email"

                            class="w-full p-3 border border-gray-300  outline-none focus:ring-2 focus:ring-blue-500"

                            placeholder="Enter Email Address" v-model="loginEmail" />

                    </div>

                    <div>

                        <label for="password" class="block text-sm font-bold mb-1">Password</label>

                        <input :type="passwordType" name="password" id="password"

                            class="w-full p-3 border border-gray-300  outline-none focus:ring-2 focus:ring-blue-500"

                            placeholder="Enter Password" v-model="loginPassword" />

                    </div>

                    <span class="text-red-600 text-xs">{{ errMessage }}</span>

                    <button type="submit"

                        class="w-full bg-college-blue text-white py-3  mt-4 font-bold hover:bg-hover-blue transition duration-200">

                        Login

                    </button>

                </div>

            </div>

        </form>

        <LoaderSpinner />

    </section>

    <ErrorPage />

</template>

<style scoped >

.main-container {

    background: linear-gradient(rgba(255, 255, 255, 0.4), rgba(255, 255, 255, 0.4)), url(../images/background-5.png);

    background-size: cover; }

</style>

**AdminLayoutView.vue**

<script setup>

import { RouterView } from 'vue-router';

import AdminNav from "../components/AdminNav.vue";

import AdminSidebarMainView from "../components/AdminSidebarMainView.vue";

import ErrorPage from '../components/ErrorPage.vue'

import { storeToRefs } from 'pinia'

import { useErrorPage } from '../stores/errorPage'

const errorPage = useErrorPage();

const { isError } = storeToRefs(errorPage);

</script>

<template>

  <main class="bg-admin-light-pink relative h-[100vh] overflow-none" v-if="!isError">

    <AdminNav class="z-[9999]" />

    <div class="flex laptop:h-[88vh] desktop:h-[90vh] bg-admin-light-pink">

      <AdminSidebarMainView class="z-[9999]" />

      <div class="flex-grow smartphone:justify-center w-[100%] py-4 smartphone:px-2 tablet:px-5 desktop:px-4 z-[10] bg-admin-light-pink">

        <RouterView class="bg-admin-light-pink" />

      </div>

    </div>

    <div id="modal"></div>

  </main>

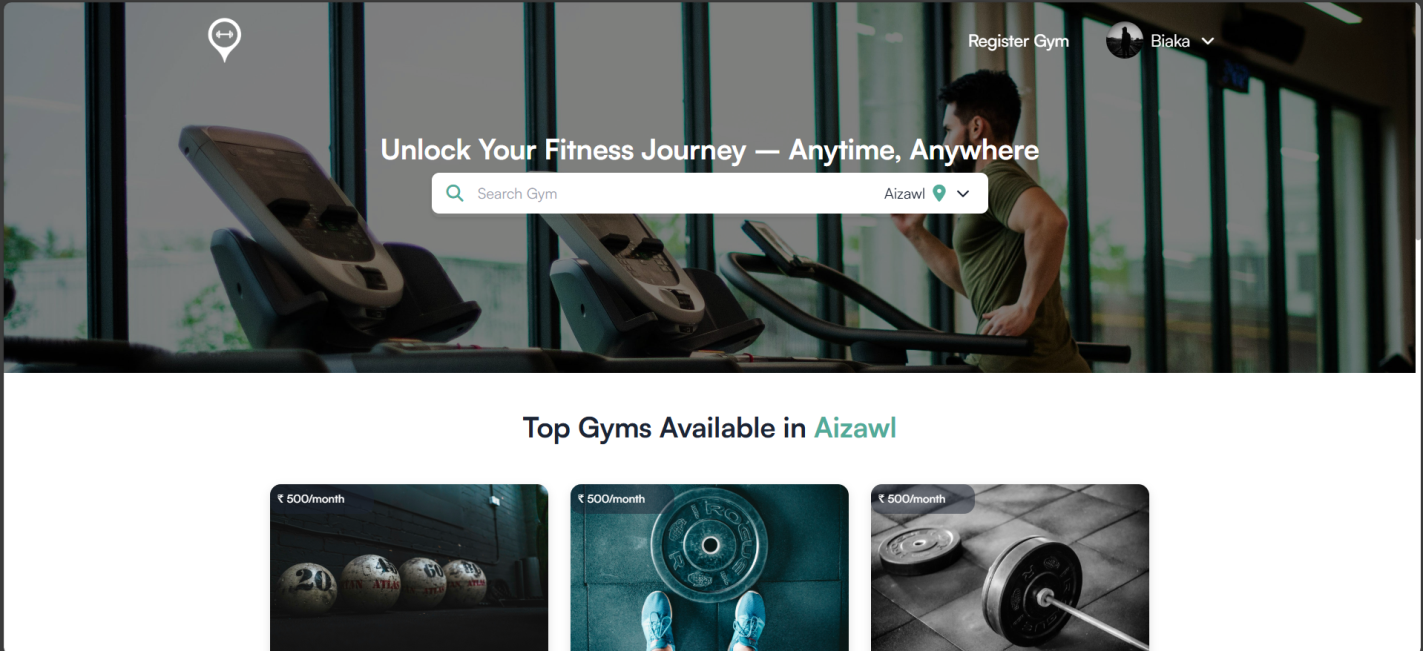
  <ErrorPage />

</template>

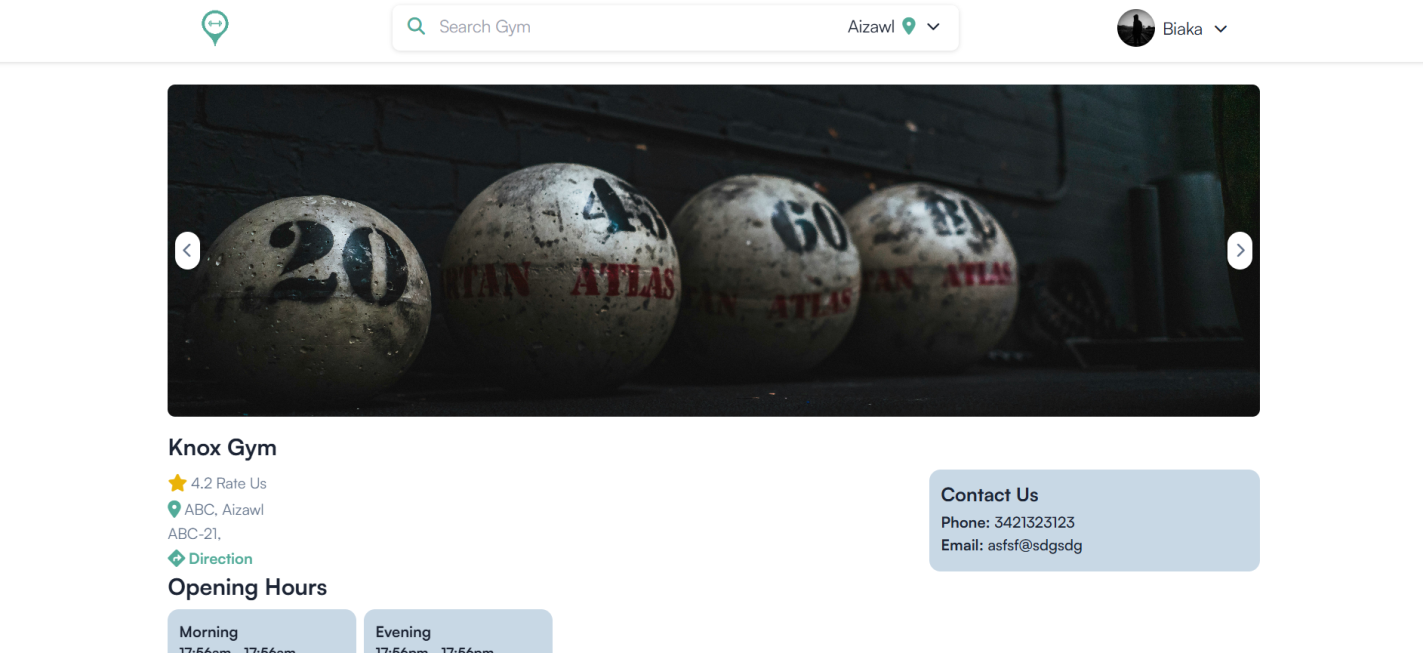
# Output Screens

# Web

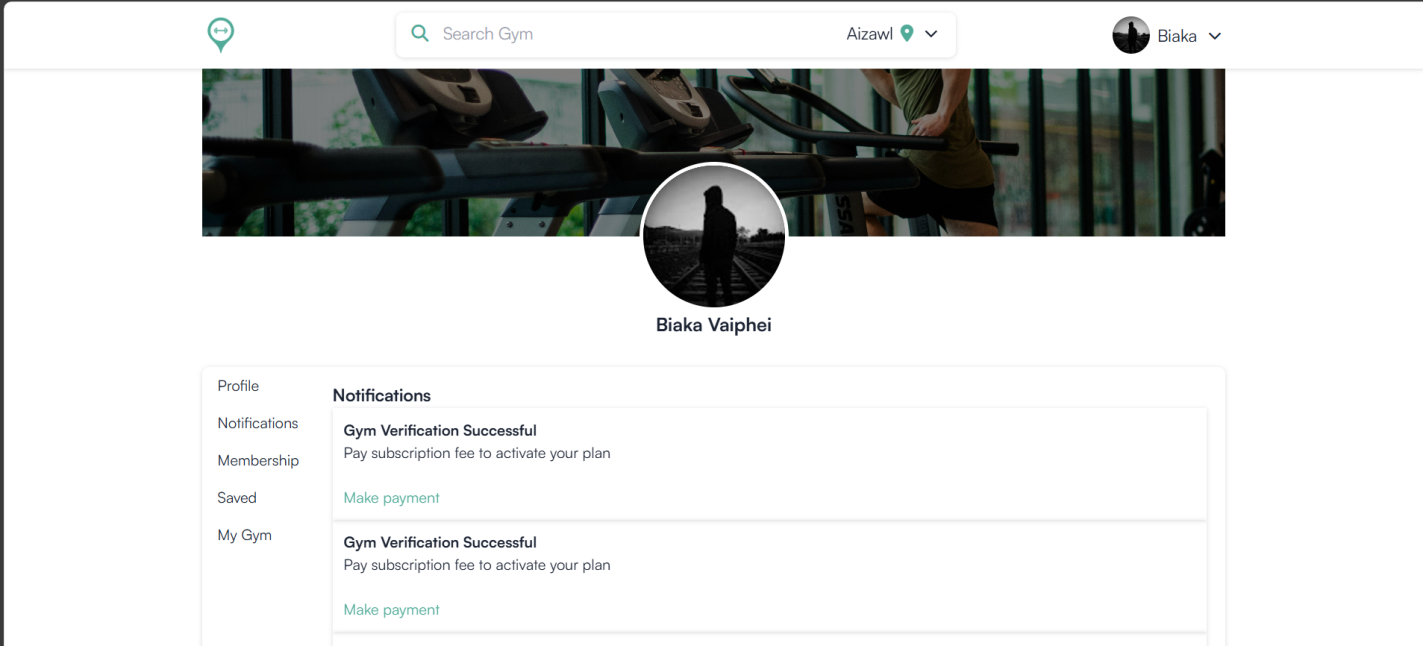
Home Page

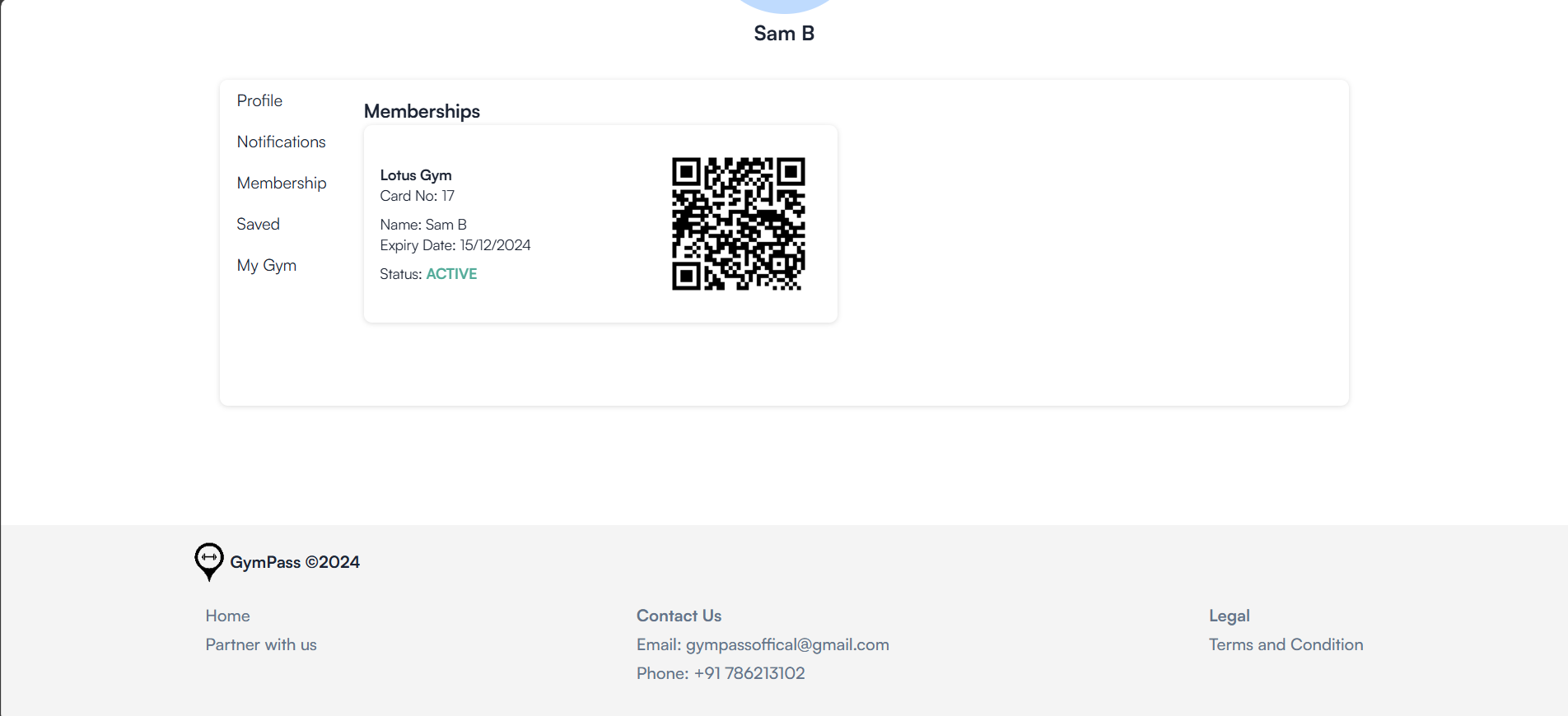


Gym Page

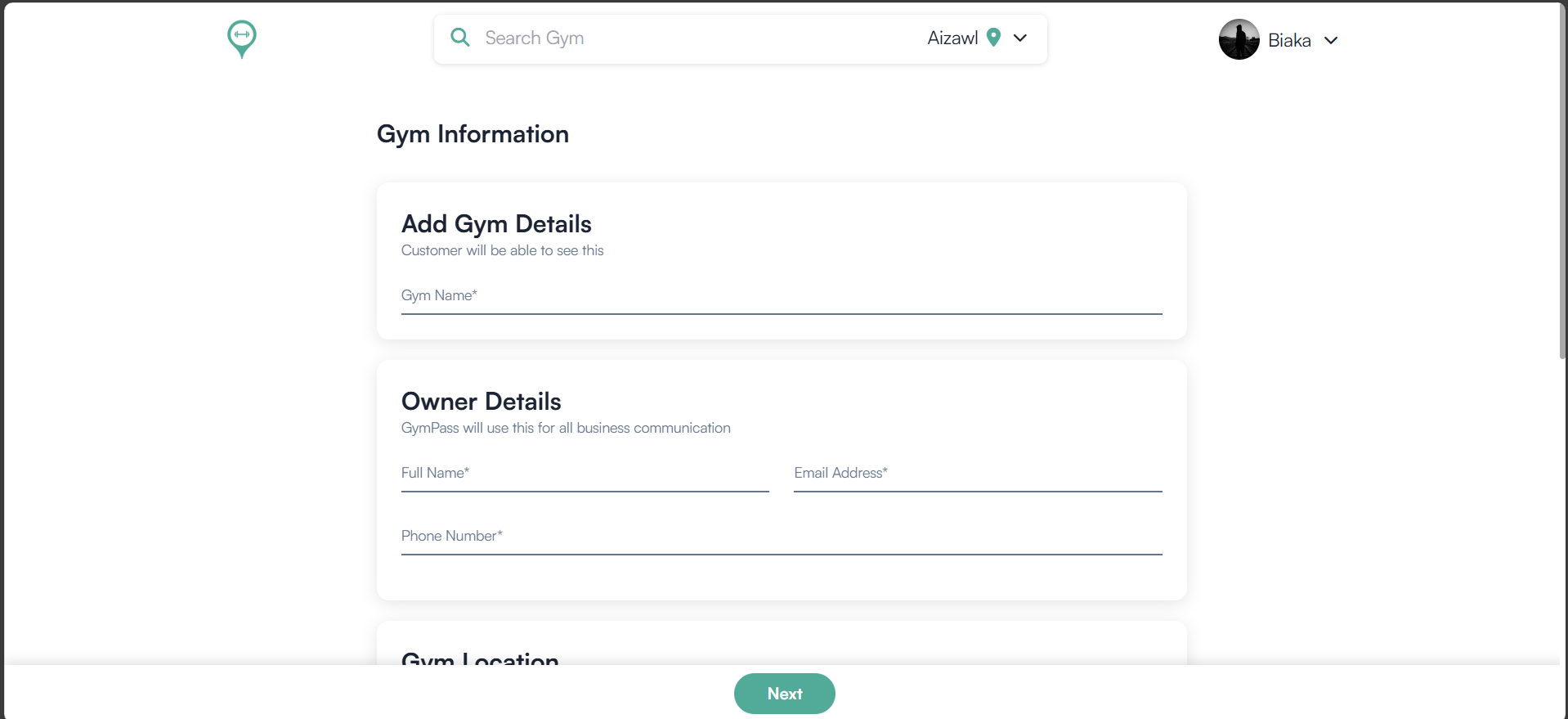


Profile Page

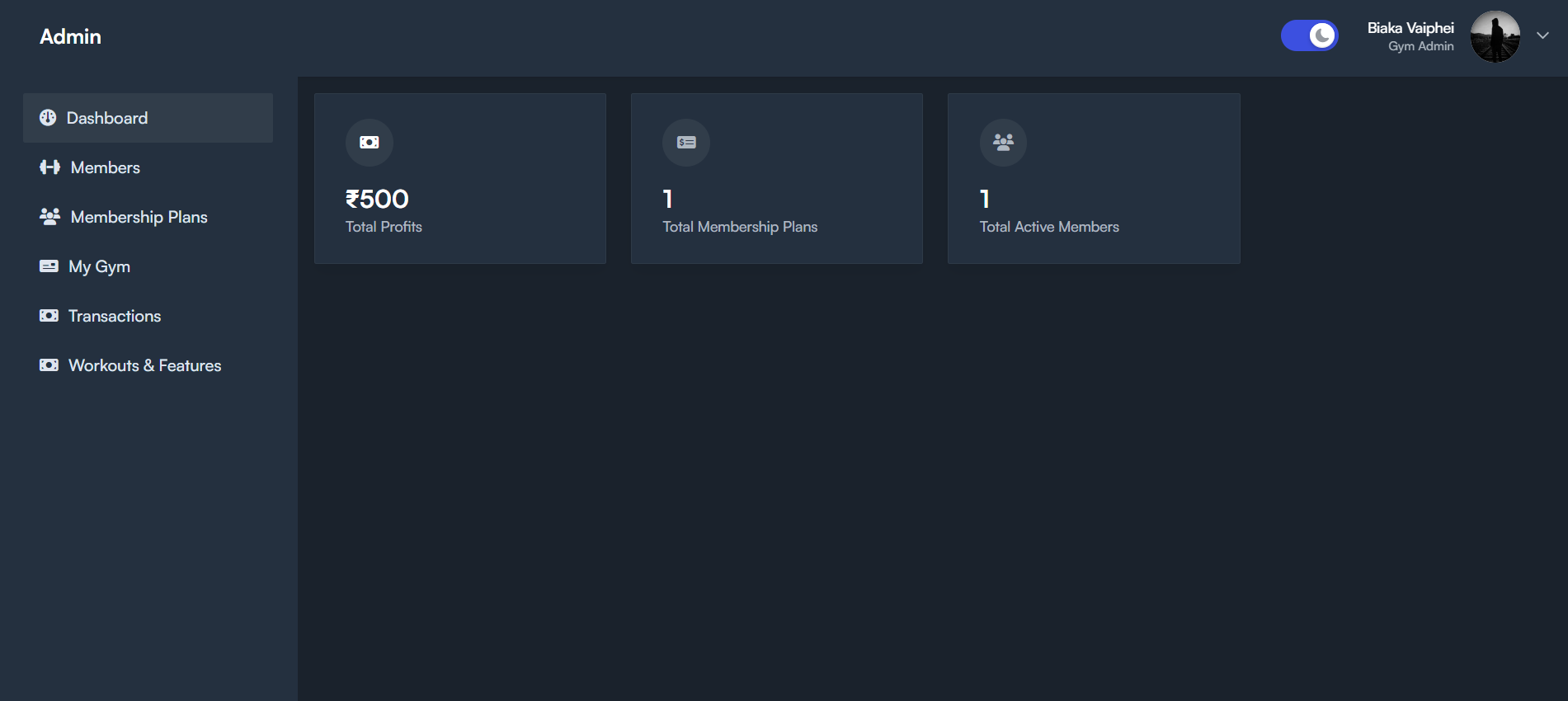


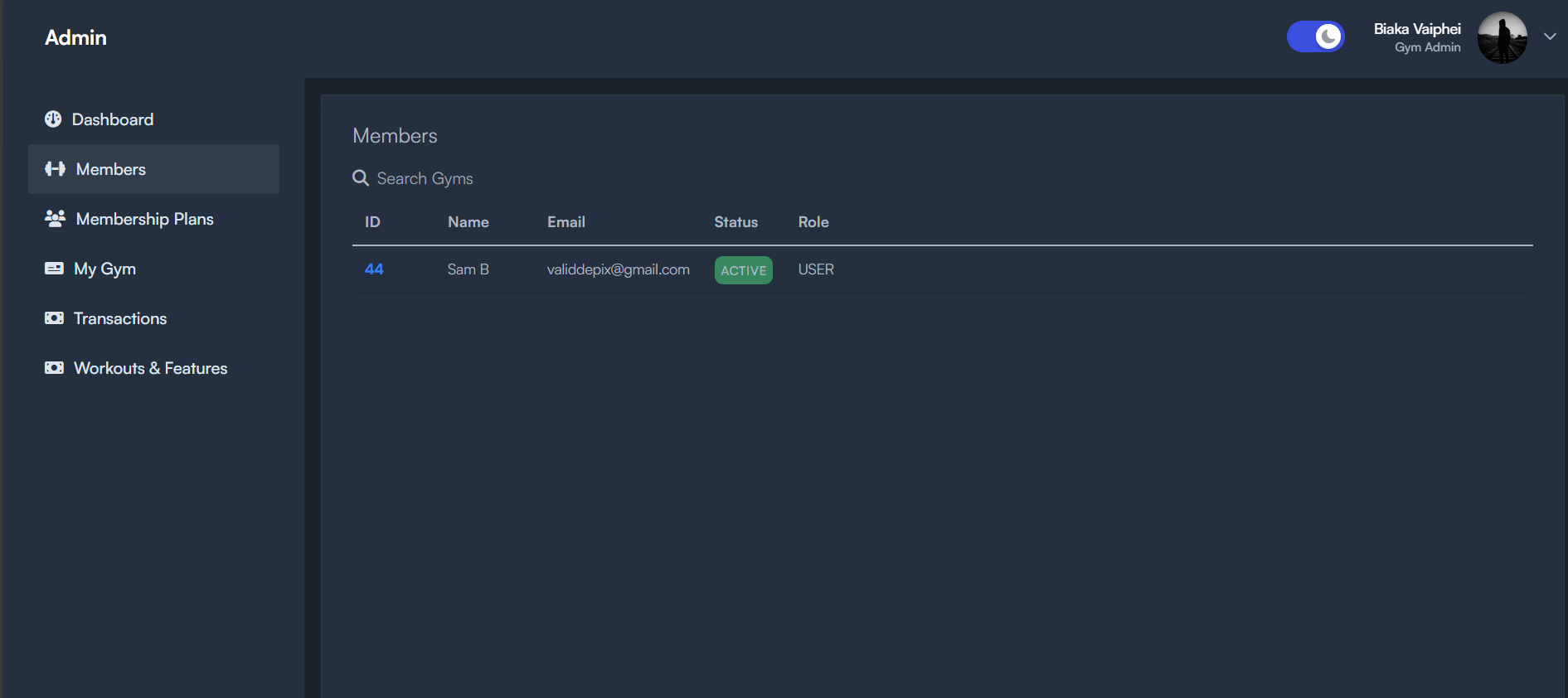


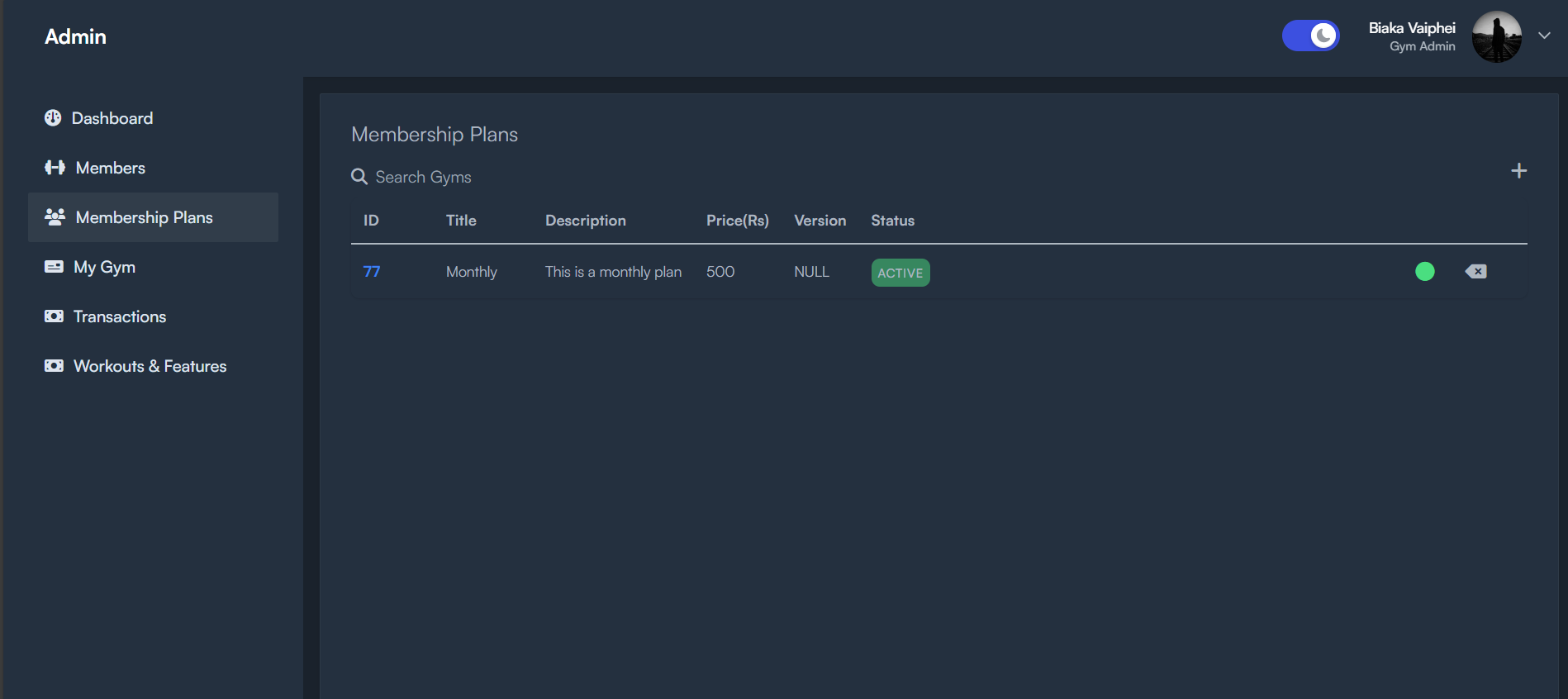
Gym Reg Page

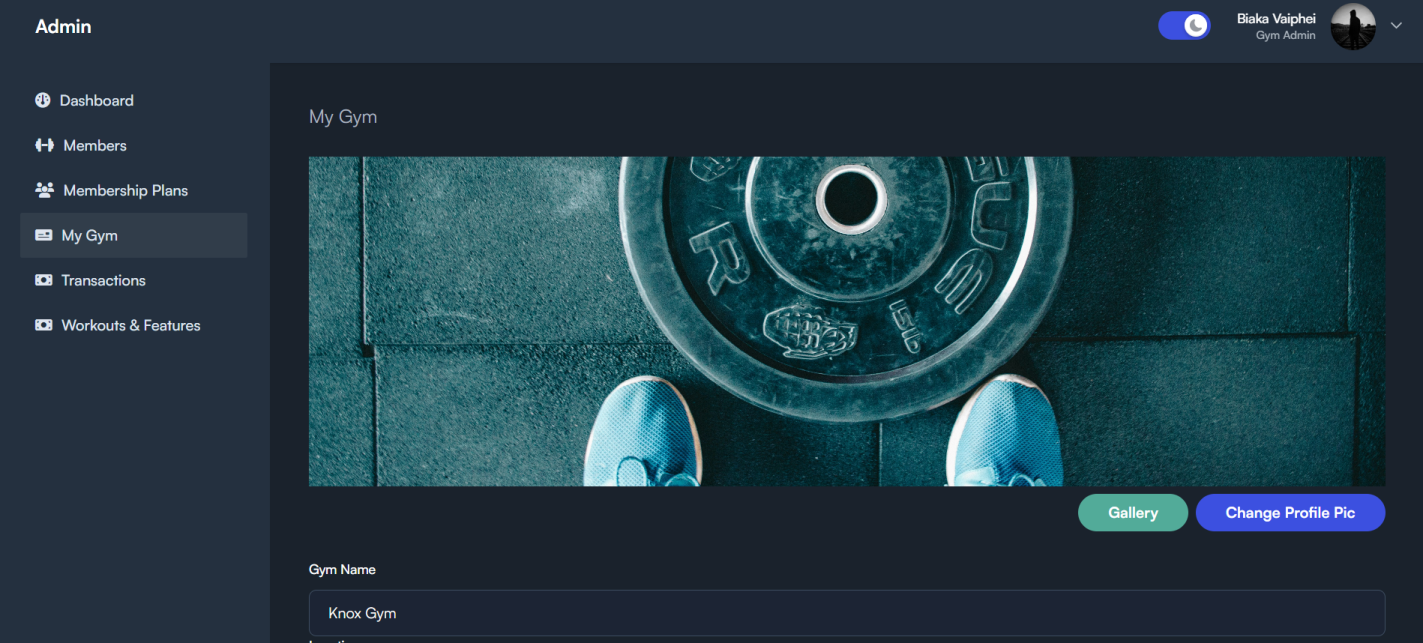


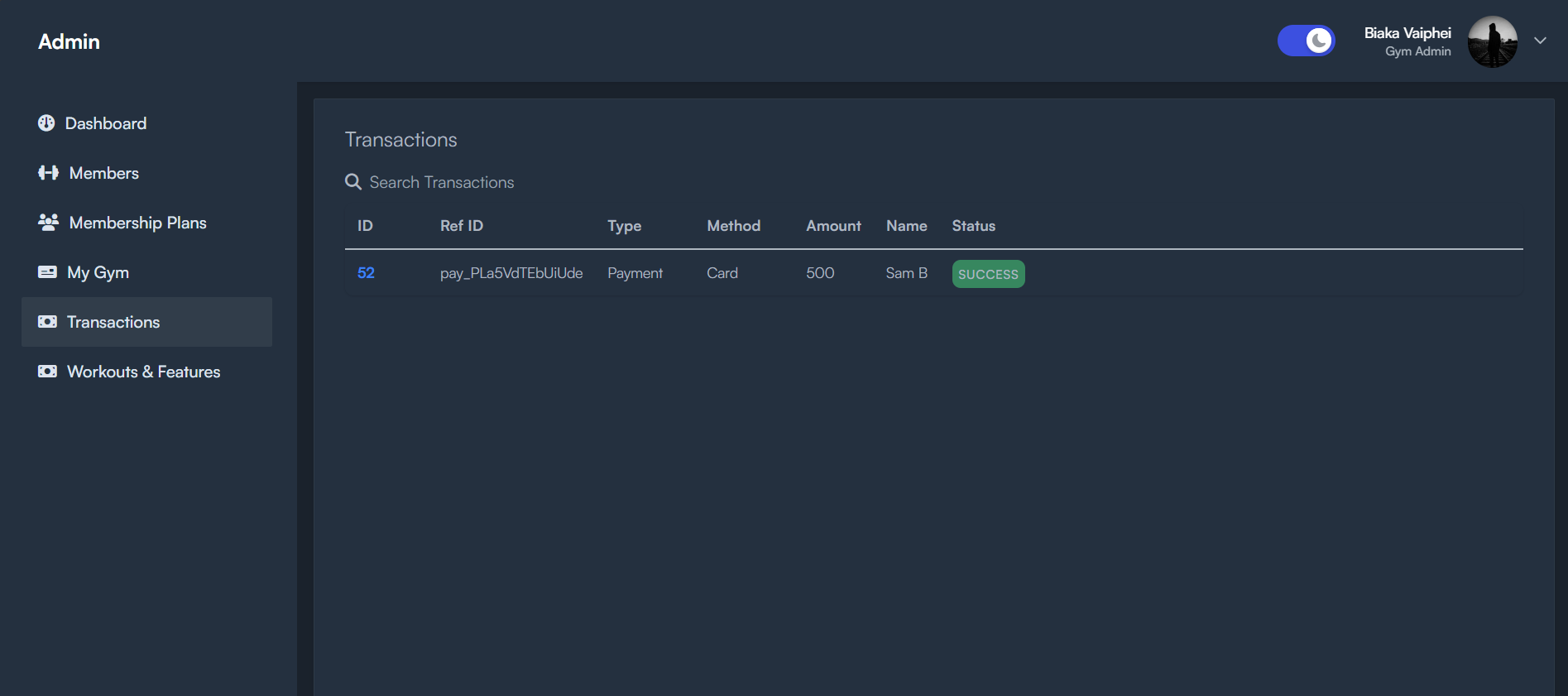
Gym Admin

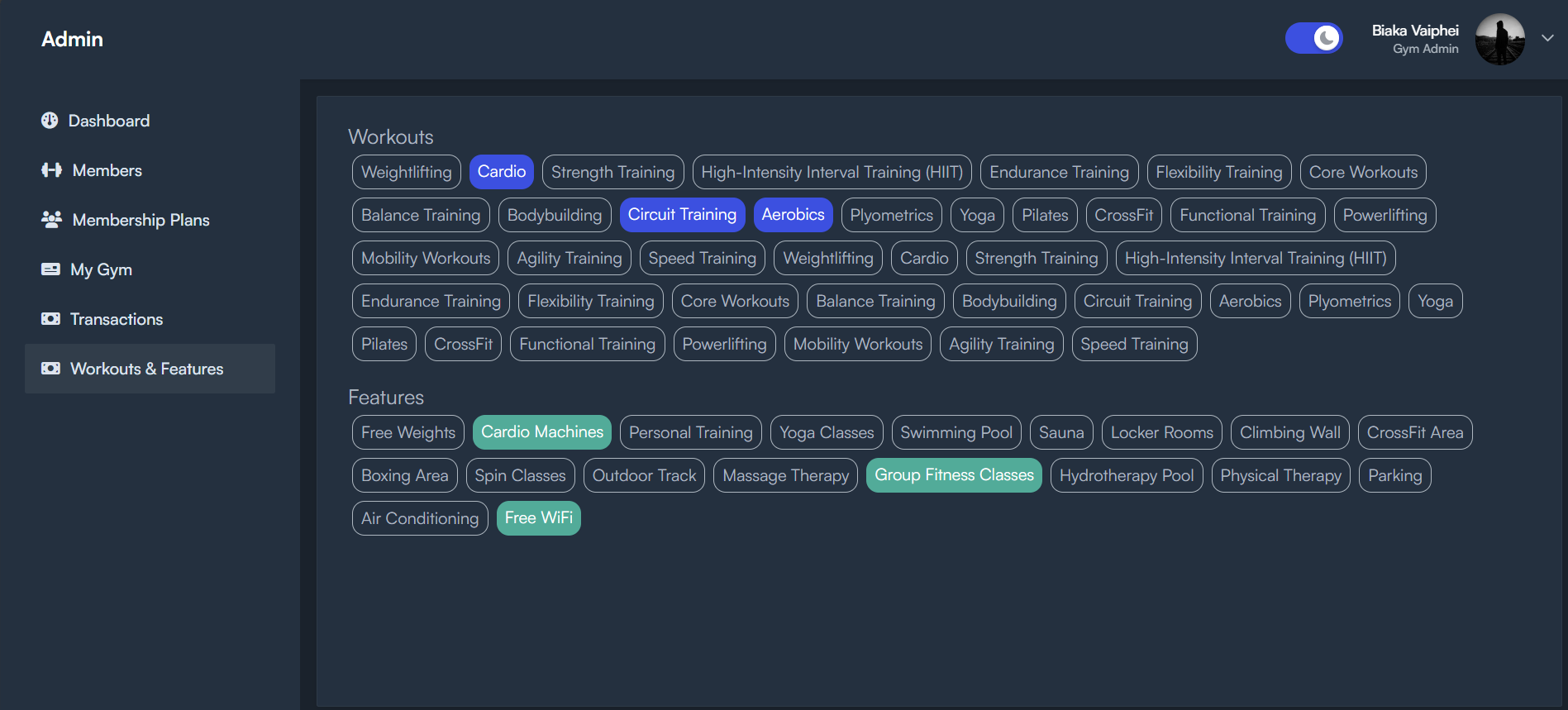




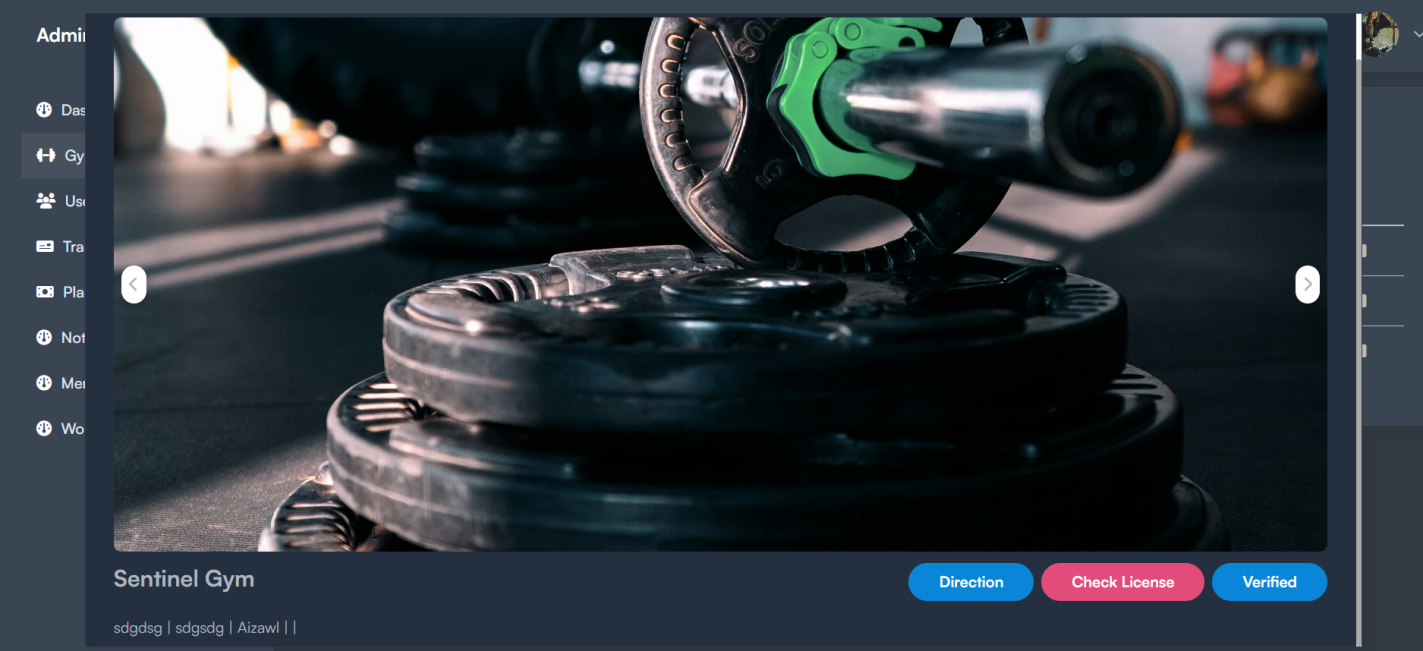


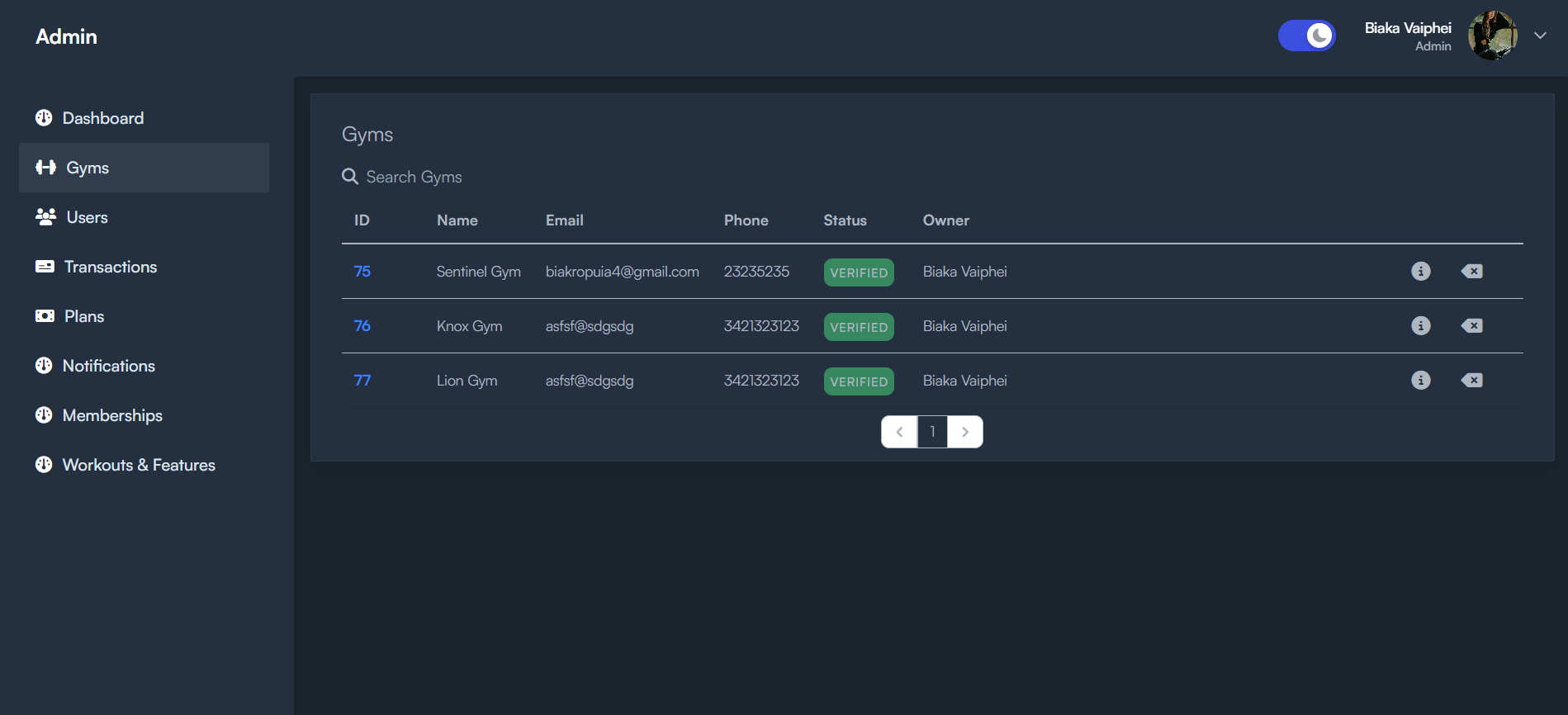


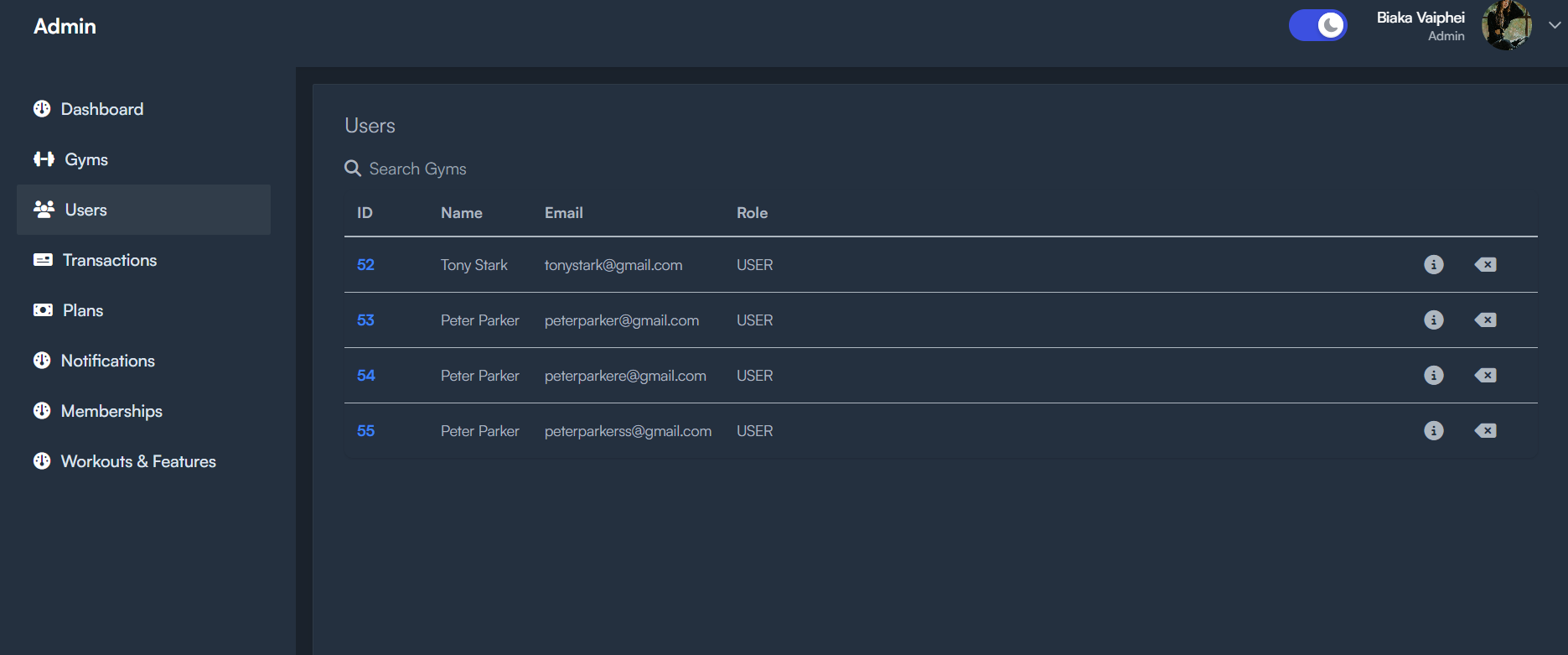




Platform Admin







# Testing

* 1. **Testing Theory**

Software testing is a critical element of software quality assurance and represents the ultimate reviews of specification, design and coding. Testing represents and interesting anomaly for the software. During earlier definition and development phases, it was attempted to build software from and abstract concept to a tangible implementation. No system is error free because it is so till the next error crops up during any phase of the development or usage of the product. A sincere effort however needs to be put to bring out a product that is satisfactory.

The testing phase involves the testing of development system using various data. Preparation of the test data plays a vital role in system testing. After preparing the test data, the system under study was tested using those data. While testing the system, by using the test data, errors were found and corrected by using the following testing steps and corrections were also noted for future use. Thus, a series of testing is performed on the proposed system before the system is ready for implementation.

The various types of testing done on the system are:

1. Integration Testing
2. Validation Testing
3. Unit Testing
4. Output Testing
   * 1. **Unit Testing**

Unit testing focuses on verification effort on the smallest unit of software design module. Using the unit test plans prepared in the design phase of the system development as a guide, important control paths are tested to uncover errors within the boundary of the modules. At the end of this testing phase each module is found to have and adverse effect working satisfactorily, as regard to the expected output from the module.

* + 1. **Integration Testing**

Data can be lost across an interface, one module can on another; sub-functions when combined may not produce the desired major function: global data structures can present problems. Integration testing is a systematic technique for the program structure while at the same time concluding tests to uncover errors associated with interface. All modules are combined in this testing step.

* + 1. **Validation Testing**

At the culmination of the integration testing, the software is completely assembled as a package, interfacing errors have been uncovered and corrected, and a final series of software validation testing began. Here we test if the system functions in a manner that can be reasonably expected by the customer. The system is tested against the system requirement specification.

* + 1. **Output Testing**

After performing validation testing, the next phase is output testing of the proposed system, since no system can be useful if it does not produce the desired output in the specified format. The output generated or displayed by the system under consideration is tested by asking the user about the format required by them, here the output format is considered in two ways.

# Conclusion

This project successfully streamlines the gym registration and management process, making it more efficient and user-friendly for both gym goers and gym owners. By automating membership sign-ups, payments, and record-keeping, it reduces manual effort, enhances accuracy, and ensures a seamless experience for all users.