Bhartiya Vidya Bhavan's
Sardar Patel Institute of Technology
(Autonomous Institute Affiliated to University of Mumbai)
Department of Computer Engineering

SCHOLARSHIP MANAGEMENT SYSTEM

By

Sejal Soni (2024300247)

Harsh Shinde (2024300241)

Pranav Solunke (2024300246)

Gangesh Dhangar (2025301006)

Guided by

Dr. Pramod Bide

Course Project

Database Management System (S.Y.2025-2026)

Abstract

The **Scholarship Portal Management System** is a comprehensive web-based platform designed to simplify and automate the process of scholarship management. This system provides an integrated environment where students can browse available scholarship schemes, apply through a structured multi-step process, upload necessary documents, and track their application status in real time.

Developed using the **MERN stack** — **MySQL**, **Express**, **React**, and **Node.js** — the system ensures efficiency, scalability, and security. The backend employs **mysql2/promise** for asynchronous operations, **bcryptjs** for password encryption, and **multer** for handling file uploads, while the frontend built in **React** offers a smooth and responsive single-page experience.

Administrators can create, update, and manage scholarship schemes, review student applications, and make approval decisions. The portal also features an integrated **grievance management system**, enabling administrators to request revisions and students to edit and resubmit their applications.

By implementing triggers, views, and data integrity constraints, this project demonstrates key **Database Management System (DBMS)** principles in a real-world setting. It ensures accuracy, transparency, and accountability, forming an ideal example of how modern web technologies integrate seamlessly with structured database management.

Table of Content

CHAPTER 1:
CREATION OF DATABASE

1. CREATION OF DATABASE

A database is the backbone of every management system.
In the **Scholarship Portal Management System**, the database named **scholarship_db** stores all core information such as user credentials, scholarship scheme details, application submissions, uploaded documents, and system notifications.
Its design ensures referential integrity, prevents redundancy, and allows smooth data retrieval through properly defined relationships and constraints.
The following sections explain each query in detail.

1.Setting Up the Database

1.1. Creating the Database

This query creates the main database and selects it for all subsequent operations.
It guarantees that all tables, triggers, and views are organized under a single logical unit.

```
CREATE DATABASE scholarship_db;
USE scholarship_db;
```

1.2 Creating the Users Table

This table maintains login information and personal details of both students and administrators.
It includes a **CHECK constraint** to enforce that only Gmail addresses are accepted and stores passwords in hashed form to protect user data.

```
CREATE TABLE users (
   id INT AUTO_INCREMENT PRIMARY KEY,
   name VARCHAR(100) NOT NULL,
   email VARCHAR(100) UNIQUE NOT NULL,
   password_hash VARCHAR(255) NOT NULL,
   role ENUM('user','admin') DEFAULT 'user',
   CHECK (LOWER(email) LIKE '%@gmail.com'));
```

1.3 Creating the Schemes Table

The following query defines the structure for all scholarship schemes available in the system.
It stores descriptive and eligibility information so that users can browse, filter, and apply to relevant programs.

```
CREATE TABLE schemes (
   id INT AUTO_INCREMENT PRIMARY KEY,
   scheme_name VARCHAR(150) NOT NULL,
   type VARCHAR(50),
   category VARCHAR(50),
   academic_year YEAR,
   amount DECIMAL(10,2),
   eligibility TEXT,
   income_limit INT,
   min_age INT,
   application_deadline DATE);
```

1.4 Creating the Applications Table

This query builds the most critical table that connects **users** and **schemes**.
It records every scholarship application submitted by students and captures its entire lifecycle status.

```
CREATE TABLE applications (
   id INT AUTO_INCREMENT PRIMARY KEY,
   user_id INT,
   scheme_id INT,
   data_json JSON,
   status ENUM('Pending','Approved','Rejected','Grievance') DEFAULT 'Pending',
   grievance_message TEXT,
   can_resubmit BOOLEAN DEFAULT 0,
   application_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   FOREIGN KEY (user_id) REFERENCES users(id),
   FOREIGN KEY (scheme_id) REFERENCES schemes(id));
```

1.5 Creating the Documents Table

Applicants must upload supporting files such as photographs or mark sheets.
The documents table manages these uploads and maintains a one-to-many link with applications.

```
CREATE TABLE documents (
   id INT AUTO_INCREMENT PRIMARY KEY,
   application_id INT,
   doc_type VARCHAR(50),
   filename VARCHAR(255),
   uploaded_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   FOREIGN KEY (application_id) REFERENCES applications(id) ON DELETE
CASCADE);
```

1.6 Creating the Notifications Table

The portal provides user alerts for events such as application approval, rejection, or grievance messages.
The following query defines the notifications table that logs these system updates.

```
CREATE TABLE notifications (
   id INT AUTO_INCREMENT PRIMARY KEY,
   user_id INT,
   message TEXT,
   type VARCHAR(50),
   related_id INT,
   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   FOREIGN KEY (user_id) REFERENCES users(id));
```

1.7 Inter-Table Relationships

The overall schema creates logical links among all entities:

- **users → applications** (one-to-many) – each user can submit multiple applications.
- **schemes → applications** (one-to-many) – each scheme can receive many applications.

- **applications → documents** (one-to-many) – each application can have multiple uploaded documents.
- **users → notifications** (one-to-many) – each user can receive many notifications.

These carefully defined foreign-key relationships ensure **data integrity**, simplify joins, and enable complex queries such as viewing all applications of a student or all applicants for a particular scheme.

CHAPTER 2:
DATA MANIPULATION

2. DATA MANIPULATION

The **Data Manipulation Language (DML)** operations form the functional core of the **Scholarship Portal Management System**.
These operations — including **INSERT**, **SELECT**, **UPDATE**, and **DELETE** — enable the system to dynamically manage records within the database through the Node.js backend.
This chapter explains how data is added, retrieved, modified, and removed across various modules such as registration, application submission, scheme management, and grievance handling.

2.1 User Registration

When a new user registers on the portal, their details such as name, email, and password are collected.
The password entered by the user is encrypted using **bcryptjs** before being stored, ensuring that even if the database is compromised, actual passwords remain secure.
The following SQL statement inserts the new user record into the users table.

```
INSERT INTO users (name, email, password_hash, role)
VALUES ('John Doe', 'john.doe@gmail.com', 'encrypted_password_here', 'user');
```

2.2 User Login and Verification

During login, the system must validate a user's credentials.
The backend fetches the user's record based on their email address and compares the entered password (after hashing) with the stored hash using bcrypt.compare().
The following query retrieves the necessary user details for verification.

```
SELECT id, name, email, password_hash, role
FROM users
WHERE email = 'john.doe@gmail.com';
```

2.3 Viewing and Filtering Scholarship Schemes

The scholarship browsing page displays a list of active schemes that users can apply for.
Filters such as scholarship name, academic year, type, and category are applied dynamically.

Additionally, expired schemes are excluded from the default list by checking the application_deadline.

```
SELECT *
FROM schemes
WHERE (scheme_name LIKE '%merit%' OR category = 'Merit-Based')
AND (application_deadline IS NULL OR application_deadline >= CURDATE());
```

2.4 Submitting a Scholarship Application

When a student submits their application, multiple operations occur within a **single database transaction** to maintain **atomicity**.

The application details, form data, and uploaded files are inserted into different tables, but the system ensures that all insertions either succeed together or fail together.

The first query records the main application details:

```
INSERT INTO applications (user_id, scheme_id, data_json, status)
VALUES (1, 101, '{"full_name":"John Doe","dob":"2003-01-
01","income":50000}', 'Pending');
```

After this, uploaded documents such as photographs or certificates are inserted into the documents table:

```
INSERT INTO documents (application_id, doc_type, filename)
VALUES (LAST_INSERT_ID(), 'photo', 'john_photo.jpg');
```

2.5 Administrator Management of Scholarship Schemes

Administrators have complete CRUD control over scholarship schemes. They can add new schemes, update details, or delete old ones.

When creating a new scheme, the admin inserts its details using the following query:

```
INSERT INTO schemes (scheme_name, type, category, amount, academic_year)
VALUES ('Merit Scholarship', 'Post-Matric', 'Merit-Based', 10000, 2024);
```

If an outdated scheme must be removed, the admin performs the following operation:

```
DELETE FROM schemes WHERE id = 101;
```

2.6 Reviewing and Updating Application Status

When administrators review submitted applications, they can approve, reject, or mark them for grievance correction.

Each decision updates the status field in the applications table.

For approving an application:

```
UPDATE applications
SET status = 'Approved'
WHERE id = 501;
```

For rejecting an application:

```
UPDATE applications
SET status = 'Rejected'
WHERE id = 502;
```

2.7 Grievance System Operations

The grievance system allows administrators to request changes or missing documents in a student's application.

When a grievance is raised, the status is updated and a message is recorded:

```
UPDATE applications
SET status = 'Grievance', grievance_message = 'Re-upload income certificate',
can_resubmit = 1
WHERE id = 503;
```

Once the student corrects the issue and resubmits, the application re-enters the review process:

```
UPDATE applications
SET status = 'Pending', grievance_message = NULL, can_resubmit = 0
WHERE id = 503;
```

2.8 Deleting Invalid or Test Applications

For maintenance or testing, invalid application entries can be removed manually by administrators.

```
DELETE FROM applications WHERE id = 504;
```

CHAPTER 3:
OPERATIONS ON DATABASE

3. OPERATIONS ON DATABASE

The operations performed on a database ensure that data remains valid, consistent, and meaningful.
This chapter focuses on essential **SQL operations** that implement the system's business logic, including **constraints, dynamic filtering, insertion, updates, and cascading deletions**.
Each operation contributes to maintaining the database's **integrity, security, and reliability** while supporting real-time portal functionality.

3.1 Enforcing Data Integrity – Gmail Address Constraint

To maintain uniformity and prevent invalid user registrations, the portal allows only **Gmail-based email addresses** for user accounts.
This restriction enhances authenticity and simplifies communication since most student users use Gmail accounts.
The following constraint is applied at the database level to enforce this rule:

```
ALTER TABLE users
ADD CONSTRAINT chk_gmail_only
CHECK (LOWER(email) LIKE '%@gmail.com');
```

3.2 Dynamic Filtering and Deadline Logic for Schemes

The Scholarship Portal allows students to search for active scholarships using filters like **category**, **type**, and **academic year**, while ensuring that expired scholarships are hidden by default.
The backend dynamically builds SQL queries based on user input.
A typical query used for displaying active scholarships is as follows:

```
SELECT *
FROM schemes
WHERE (scheme_name LIKE '%merit%' OR category = 'Merit-Based')
AND (application_deadline IS NULL OR application_deadline >= CURDATE());
```

3.3 Inserting Application and Associated Documents

Submitting a scholarship application involves inserting records into both the **applications** and **documents** tables within a **single transaction**.

This ensures that either both operations succeed together or none at all, maintaining atomicity.
The first query inserts the student's application details:

```
INSERT INTO applications (user_id, scheme_id, data_json, status)
VALUES (1, 101, '{"full_name":"John Doe","dob":"2003-01-
01","income":50000}', 'Pending');
```

Once the application is created, supporting documents are added:

```
INSERT INTO documents (application_id, doc_type, filename)
VALUES (LAST_INSERT_ID(), 'income_certificate',
'john_income_certificate.pdf');
```

3.4 Updating Application Status – Grievance Lifecycle

The **grievance feature** allows administrators to send applications back to users for correction or resubmission.
This involves updating both the status and grievance_message columns of the applications table.
Two main update queries are used in this lifecycle:
When an admin raises a grievance request:

```
UPDATE applications
SET status = 'Grievance',
   grievance_message = 'Please re-upload your income certificate',
   can_resubmit = 1
WHERE id = 503;
```

When the user corrects and resubmits the application:

```
UPDATE applications
SET status = 'Pending',
   grievance_message = NULL,
   can_resubmit = 0
WHERE id = 503;
```

3.5 Automatic Deletion using Cascading Foreign Keys

To prevent inconsistent or orphaned data, the system uses **ON DELETE CASCADE** for dependent tables such as applications and documents.

This ensures that when a parent record (like a scheme) is deleted, all related records are automatically removed by the DBMS.

```
ALTER TABLE applications
ADD CONSTRAINT fk_scheme_delete
FOREIGN KEY (scheme_id) REFERENCES schemes(id)
ON DELETE CASCADE;
```

3.6 Maintaining Notifications for Status Changes

Whenever an application's status changes, the system must notify the respective user.

Although this is automated using a trigger (explained in Chapter 5), the underlying SQL operation for logging a notification is shown below:

```
INSERT INTO notifications (user_id, message, type, related_id)
VALUES (
   1,
   'Your application for Merit Scholarship was updated to: Approved',
   'status_update',
   503);
```

3.7 Complex Join for Application Overview

For administrative review, a combined view of all applications, users, and schemes is necessary.

This requires a **JOIN operation** that retrieves complete application details in a single query:

```
SELECT
   a.id AS application_id,
   u.name AS applicant_name,
   u.email AS applicant_email,
   s.scheme_name,
   a.status,
   a.application_date
FROM applications a
JOIN users u ON a.user_id = u.id
JOIN schemes s ON a.scheme_id = s.id;
```

CHAPTER 4:
VIEWS

4. VIEWS

A **View** in SQL is a virtual table generated by executing a stored query.
It allows users to see combined or filtered data from multiple tables without directly accessing the base tables.
In the **Scholarship Portal Management System**, views are used to simplify complex joins between tables such as users, applications, and schemes, making it easier for administrators to monitor all student applications in one place.
Views also improve security by exposing only relevant fields while hiding sensitive data such as password hashes.

4.1 Purpose of Views in the Scholarship Portal

Views serve three key purposes in this project:

1. **Simplification of Complex Queries** – Instead of writing repetitive JOIN queries in multiple API endpoints, a single pre-defined view consolidates all related information.
2. **Enhanced Data Security** – By using views, sensitive fields (like password_hash) from base tables are hidden from unauthorized access.
3. **Data Abstraction for Admin Review** – Views provide a logical representation of essential information such as applicant name, scheme applied for, and current status, which the admin can directly query.

Thus, views act as a **data abstraction layer** between the raw tables and the presentation layer in the admin dashboard.

4.2 Creating the Admin Application Review View

The following view, named **v_application_details**, combines columns from the applications, users, and schemes tables.
It provides a single, easy-to-read record set for administrators, displaying applicant details, scheme information, and the current application status.

```
CREATE VIEW v_application_details AS
SELECT
    a.id AS application_id,
    a.application_date,
    a.status,
    a.grievance_message,
```

```
    u.name AS applicant_name,
    u.email AS applicant_email,
    s.scheme_name,
    s.type AS scheme_type,
    s.category AS scheme_category,
    s.academic_year
FROM
    applications a
LEFT JOIN
    users u ON a.user_id = u.id
LEFT JOIN
    schemes s ON a.scheme_id = s.id;
```

The alias v_application_details gives a meaningful name to this logical view, making it easy for admins to access through a single SQL command:

```
SELECT * FROM v_application_details;
```

This command retrieves a complete list of all applications along with associated user and scheme details — without the need for multiple joins or complex backend logic.

4.3 Benefits of Using the View

The introduction of v_application_details provides the following major advantages:

- **Simplified Reporting:** The admin can directly fetch structured data for analysis, export, or visualization.
- **Improved Security:** The view omits sensitive fields such as password_hash, ensuring only relevant data is exposed.
- **Reusability:** The same view is used by multiple backend routes (like /api/admin/applications) without rewriting SQL joins.
- **Performance Optimization:** Since MySQL internally optimizes view execution, frequent dashboard queries execute faster and more consistently.

CHAPTER 5:
TRIGGERS

5. TRIGGERS

A **Trigger** in SQL is a special type of stored procedure that is automatically executed, or "fired," by the database whenever a specified event occurs — such as an **INSERT**, **UPDATE**, or **DELETE** operation.
Triggers are extremely valuable for maintaining data consistency, enforcing business rules, and automating backend tasks.
In the **Scholarship Portal Management System**, a trigger is used to automatically record or notify users whenever the status of an application changes.
This ensures that no manual step is needed to log important updates — maintaining **real-time synchronization** between database activities and user notifications.

5.1 Purpose of Using Triggers

Before exploring the trigger definition, it's important to understand why triggers are useful in this system:

1. **Automation:** They automatically perform tasks after certain actions, reducing manual effort.
2. **Data Integrity:** They ensure that whenever sensitive data (like application status) changes, it is properly logged.
3. **Transparency:** Students are automatically notified when admins approve, reject, or mark their applications for grievance correction.
4. **Audit Trail Creation:** Triggers help in maintaining an audit record for accountability and future reference.

Thus, triggers form a crucial link between the **application workflow** and the **database layer**.

5.2 Designing the Trigger for Application Status Update

Whenever an admin updates the status column in the applications table (for example, marking an application as Approved, Rejected, or Grievance), the system should automatically log this change in the notifications table.
The following SQL trigger definition automates that process.
Explanation (before the code):
This trigger, named **application_status_update**, executes **after an UPDATE** operation on the applications table.
It compares the previous and new status values. If they differ, it inserts a message into the notifications table, informing the respective user of the update.
This automated behavior ensures **event-driven data logging** within the system.

```
DELIMITER $$

CREATE TRIGGER application_status_update
AFTER UPDATE ON applications
FOR EACH ROW
BEGIN
   IF OLD.status != NEW.status THEN
      INSERT INTO notifications (user_id, message, type, related_id)
      VALUES (
         NEW.user_id,
         CONCAT('Your application for ',
               (SELECT scheme_name FROM schemes WHERE id =
NEW.scheme_id),
               ' was updated to: ', NEW.status),
         'status_update',
         NEW.id
      );
   END IF;
END$$

DELIMITER ;
```

5.3Trigger in Action

To understand how this trigger works, consider the following update query executed by the admin:

```
UPDATE applications
SET status = 'Approved'
WHERE id = 503;
```

5.4 Advantages of Triggers in the System

Implementing triggers in the Scholarship Portal offers multiple benefits:

- **Automation of repetitive tasks:** No need for manual notification insertion in the backend code.

- **Improved consistency:** Ensures that every status change is logged uniformly.
- **Enhanced reliability:** Reduces the chance of human error or forgotten updates.
- **Better transparency:** Users are promptly informed about the progress of their applications.
- **Foundation for future features:** This logic can easily be extended for real-time notifications (via WebSockets or push notifications).

5.5 Future Enhancement – Audit Log Extension

While the current trigger focuses on notifications, the same logic can be extended to maintain a full **audit trail** of every application change.
For example, a separate table named audit_log can store the details of who made the change and when it occurred.

For e.g.:

```
INSERT INTO audit_log (application_id, old_status, new_status, changed_by, changed_at)
VALUES (NEW.id, OLD.status, NEW.status, CURRENT_USER(), NOW());
```

CHAPTER 6:
DATABASE CONNECTIVITY

6. DATABASE CONNECTIVITY

Database connectivity forms the crucial bridge between the **Node.js backend** and the **MySQL database** in the Scholarship Portal Management System.
A robust, efficient, and secure connection layer ensures smooth data flow for every operation — from user registration to application management.
In this system, the backend communicates asynchronously with MySQL using the mysql2/promise library, which provides high-performance, non-blocking connections.

6.1 Objective of Database Connectivity

The objective of this layer is to:

1. Establish a secure connection between the backend server and MySQL database.
2. Use **connection pooling** for optimal performance under multiple concurrent requests.
3. Prevent the exposure of sensitive credentials by using environment variables.
4. Handle database queries efficiently using reusable helper functions.
5. Support transactional operations (for atomic multi-step actions like applying for scholarships).

6.2 Installing and Importing Database Drivers

Before interacting with MySQL, the project must install the driver package that allows Node.js to execute SQL commands.
Here, the mysql2 library (specifically its promise-based version) is chosen for its asynchronous design and ease of use.

```
npm install mysql2
```

Next, it is imported into the project:

```
const mysql = require('mysql2/promise');
```

This line imports the promise-based version of mysql2, which allows the use of modern async/await syntax for database operations. The promise model ensures non-blocking behavior — meaning the server remains responsive even while waiting for the database to complete a query.

6.3 Configuring Connection Pool

Instead of creating a new connection for every request, the system uses a connection pool.
A pool maintains multiple pre-established connections that can be reused, significantly reducing latency and improving performance under load.

```
const dbConfig = {
   host: process.env.DB_HOST,
   user: process.env.DB_USER,
   password: process.env.DB_PASSWORD,
   database: process.env.DB_DATABASE
};

const pool = mysql.createPool(dbConfig);
```

6.4 Using Environment Variables for Security

To prevent exposing confidential data such as the database username, password, or host URL, environment variables are stored in a .env file.
This file is read using the dotenv library:
require('dotenv').config();

Example .env file contents:
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=your_password
DB_DATABASE=scholarship_db

6.5 Executing Queries Using Helper Functions

To streamline and standardize all database interactions, a set of reusable **helper functions** are defined.
These functions abstract away repetitive code and make database operations cleaner and more maintainable.

```
// Executes a SELECT query expected to return multiple rows
async function allSql(sql, params = []) {
```

```
    const [rows] = await pool.query(sql, params);
    return rows;
}

// Executes a SELECT query expected to return a single row
async function getSql(sql, params = []) {
    const [rows] = await pool.query(sql, params);
    return rows[0];
}

// Executes INSERT, UPDATE, or DELETE queries
async function runSql(sql, params = []) {
    const [result] = await pool.query(sql, params);
    return result;
}
```

6.6 Transaction Management

Certain actions, such as submitting a scholarship application (which includes inserting application data and uploading multiple documents), must either **fully succeed or fail entirely**.

To ensure data integrity in such cases, **transactions** are used.

```
const connection = await pool.getConnection();
try {
    await connection.beginTransaction();

    await connection.query('INSERT INTO applications (user_id, scheme_id,
data_json, status) VALUES (?, ?, ?, "Pending")', [userId, schemeId, formData]);

    const [appResult] = await connection.query('SELECT LAST_INSERT_ID() AS
appId');
    const appId = appResult[0].appId;

    await connection.query('INSERT INTO documents (application_id, doc_type,
filename) VALUES (?, ?, ?)', [appId, 'photo', photoName]);
```

```
    await connection.query('INSERT INTO documents (application_id, doc_type,
filename) VALUES (?, ?, ?)', [appId, 'mark10', markSheetName]);

    await connection.commit();
} catch (error) {
    await connection.rollback();
    console.error('Transaction failed:', error);
} finally {
    connection.release();
}
```

6.7 Error Handling and Stability

The backend includes robust error handling to prevent silent database failures. If the database connection fails during startup, the server automatically terminates to prevent running in a broken state.

For e.g.:

```
pool.getConnection()
    .then(() => console.log('Database connected successfully!'))
    .catch(err => {
        console.error('Database connection failed:', err);
        process.exit(1);
    });
```

CHAPTER 7: FRONT-END

7. FRONT-END

The front-end is the visual and interactive part of the Scholarship Portal Management System through which users (students and administrators) interact with the application. It is built using React.js, a powerful JavaScript library designed for building modern, dynamic, and responsive user interfaces.
In this project, the front-end connects seamlessly to the Node.js backend through API calls, allowing users to register, browse scholarships, apply for them, and track their status — all without reloading the page.
This approach creates a Single Page Application (SPA) experience that is smooth, fast, and user-friendly.

7.1 Objective of the Front-End Design

The purpose of the front-end layer is to:

1. Provide an **intuitive and responsive interface** for both students and administrators.
2. Enable smooth navigation between multiple pages using **client-side routing**.
3. Connect dynamically to the backend APIs for real-time operations.
4. Implement clear separation between user roles (admin vs student).
5. Enhance accessibility and usability through consistent styling and feedback mechanisms.

7.2 Technology Stack Used

The front-end uses a modern technology stack from the MERN ecosystem, optimized for modularity and speed.

Technology	Purpose
React.js	Core library for building the user interface using reusable components.
React Router (react-router-dom)	Handles client-side routing between pages without full reloads.
JavaScript (ES6+)	Provides application logic, API calls, and state management.
CSS	Controls the styling, layout, and responsiveness of all pages and components.
Fetch API / Axios	Used for sending and receiving data from the backend asynchronously.

This combination provides a modern web experience with minimal reloads and fast page rendering.

7.3 Application Structure

The entire React front-end is organized into modular components for clarity and scalability.

Each page (Home, Browse, Apply, Admin Dashboard, etc.) is placed under the src/pages/ directory, and reusable UI parts (like Navbar and Footer) are placed under src/components/.

```
src/
|
├── App.js
├── index.js
├── components/
|   ├── Navbar.js
|   └── Footer.js
├── pages/
|   ├── Home.js
|   ├── Browse.js
|   ├── Apply.js
|   ├── Applications.js
|   ├── AdminDashboard.js
|   ├── AdminSchemes.js
|   ├── AdminApplicationDetails.js
|   └── EditApplication.js
└── styles.css
```

This structure ensures that each functionality (viewing schemes, applying, managing grievances, etc.) is handled within its own dedicated React component.

7.4 Core Components and Their Roles

1. App.js (Main Application Component)

App.js serves as the entry point of the front-end application.

It defines all routes using **React Router**, manages global state (like currentUser), and conditionally renders pages based on whether the logged-in user is a student or an admin.

```javascript
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import Home from './pages/Home';
import Browse from './pages/Browse';
import Apply from './pages/Apply';
import Applications from './pages/Applications';
import AdminDashboard from './pages/AdminDashboard';
import AdminSchemes from './pages/AdminSchemes';
import AdminApplicationDetails from './pages/AdminApplicationDetails';
import EditApplication from './pages/EditApplication';
import Navbar from './components/Navbar';

function App() {
  const [currentUser, setCurrentUser] = useState(null);

  return (
    <Router>
      <Navbar currentUser={currentUser} setCurrentUser={setCurrentUser} />
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/browse" element={<Browse />} />
        <Route path="/apply/:id" element={<Apply />} />
        <Route path="/applications" element={<Applications />} />
        <Route path="/admin" element={<AdminDashboard />} />
        <Route path="/admin/schemes" element={<AdminSchemes />} />
        <Route path="/admin/application/:id" element={<AdminApplicationDetails />} />
        <Route path="/application/:id/edit" element={<EditApplication />} />
      </Routes>
    </Router>
  );
}
```

2. Navbar.js (Navigation Component)

The Navbar provides links to all major sections of the system, dynamically updating its links based on the user's role.

For example, only admins can see "Manage Schemes" and "Admin Dashboard" links.

Key features:

- Uses **React Router's Link** for smooth transitions.

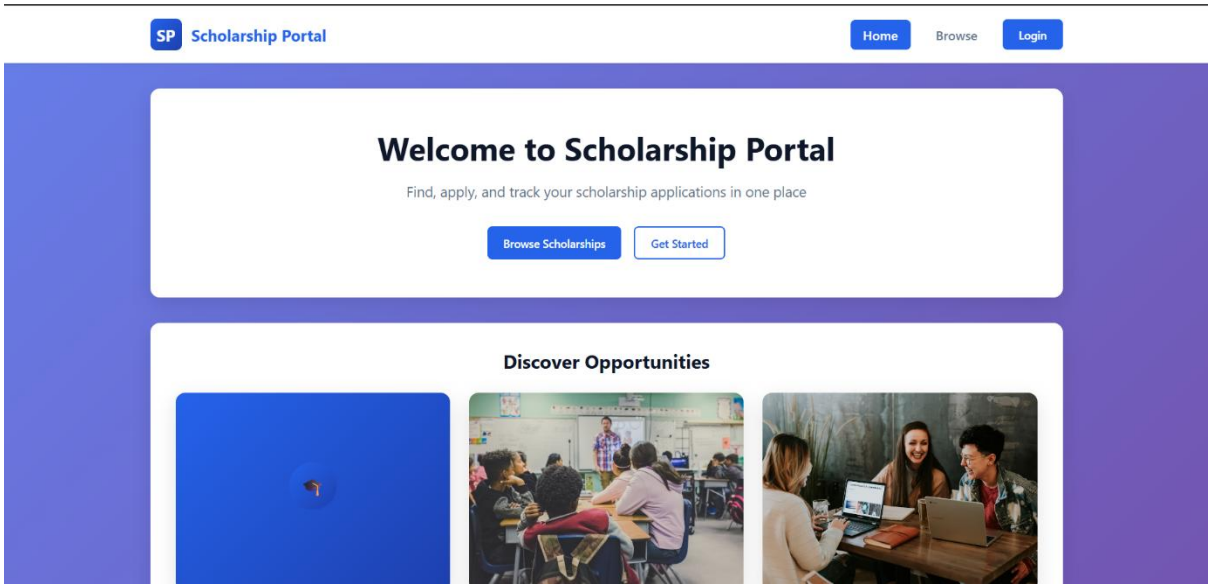
- Automatically changes links after login/logout.
- Adds styling and responsiveness for mobile view.

Github link: https://github.com/solunkepranav/project-react-based/
Demo link:
https://drive.google.com/file/d/145t9Gb4uTYbejgasyjguIxt5Q3OGsCFl/view?usp=drivesdk

3. Home.js (Landing Page)

The Home page introduces users to the portal's purpose and features.
It contains a **hero section**, short feature highlights, and quick navigation buttons to "Browse Scholarships" and "Login/Register."
It may include:

- Static banners or icons describing transparency, speed, and reliability.
- Call-to-action buttons that navigate users deeper into the portal.



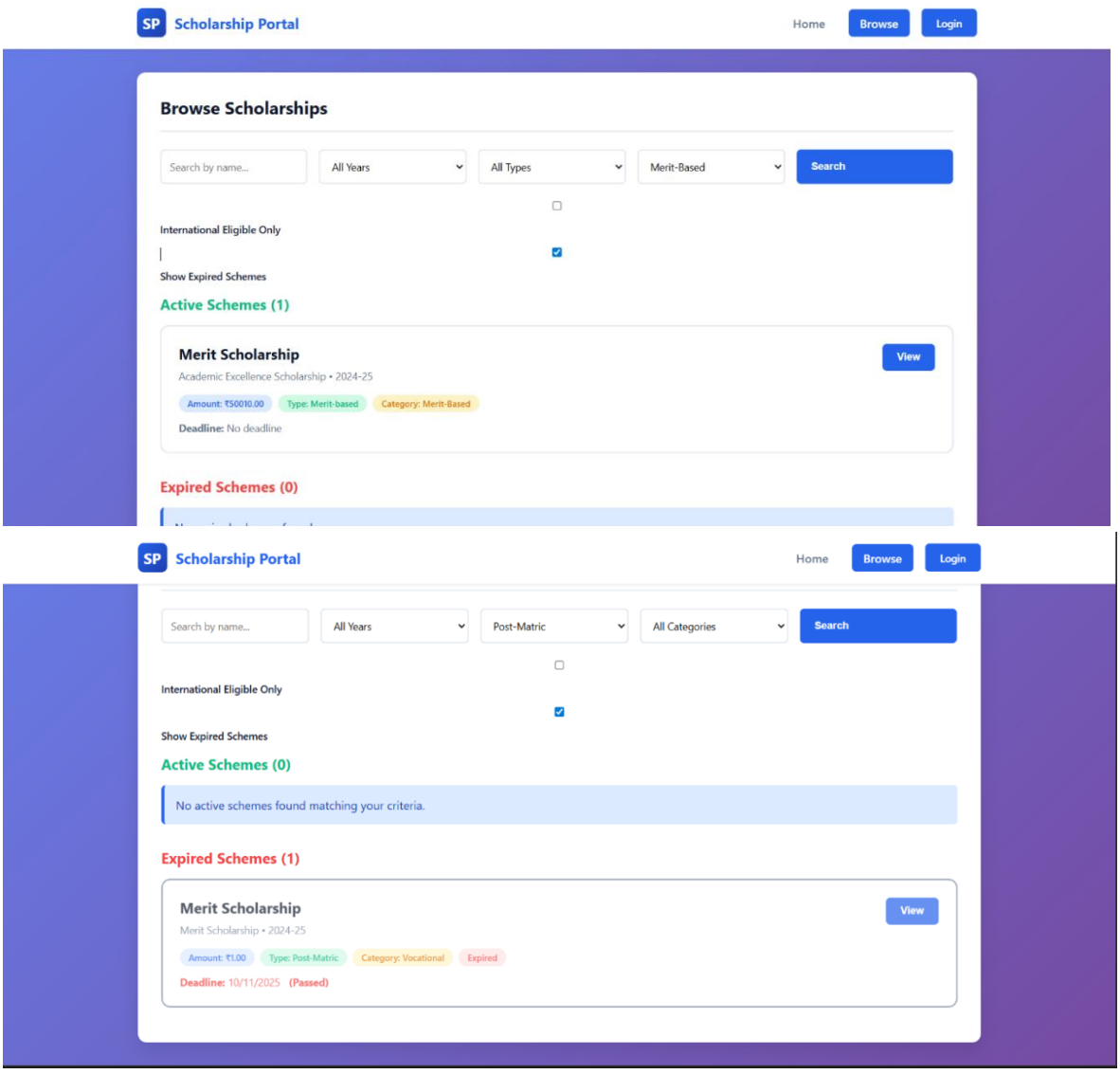
4. Browse.js (Scholarship Browsing Page)

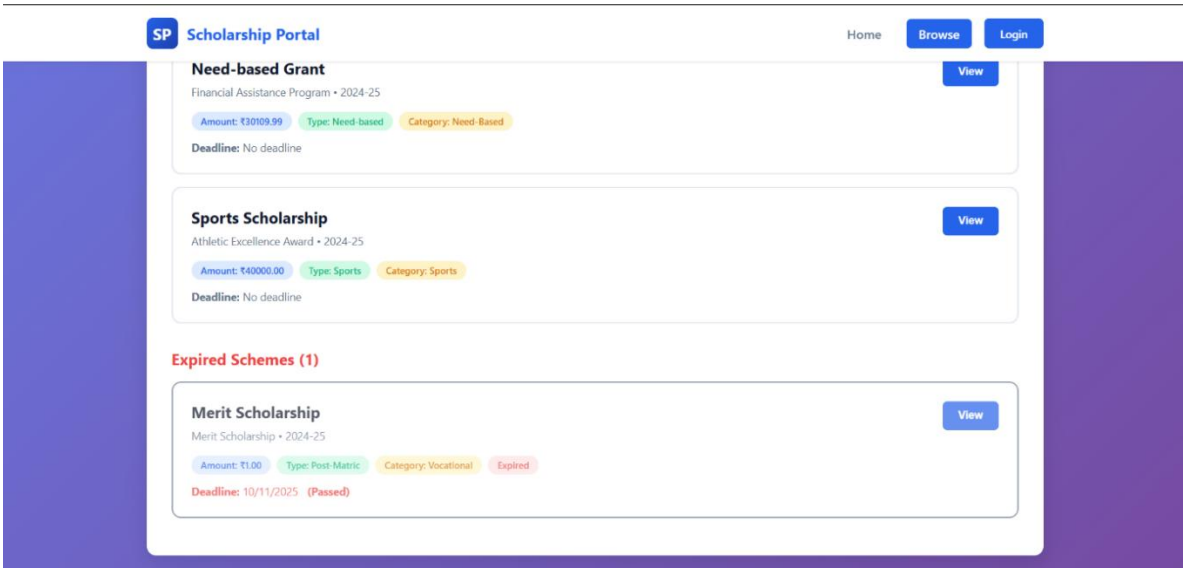
This is one of the most interactive parts of the portal.
It fetches all active scholarships from the backend API (/api/schemes) and allows users to filter them based on keywords, academic year, type, or category.
Dynamic features:

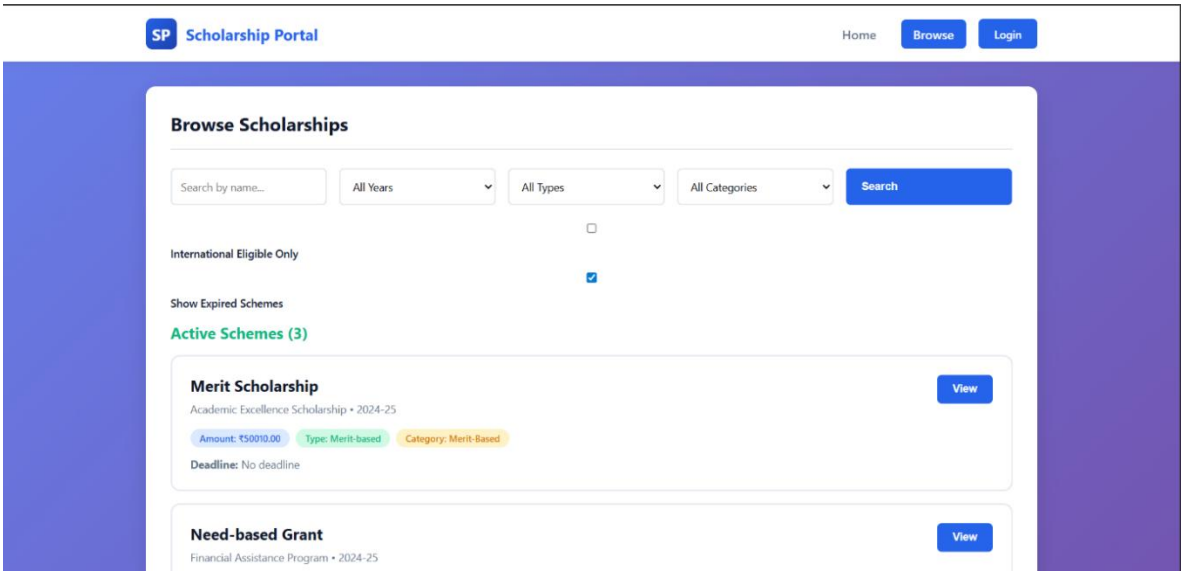
- **Search box** to look for specific scholarships by name.
- **Dropdown filters** for type (Merit-based, Need-based, etc.).

- **Separate "Expired Schemes" view** showing closed opportunities (fetched from /api/schemes/expired).

Each scholarship appears as a **card** with details like name, amount, eligibility, and a button linking to the **Apply page**.







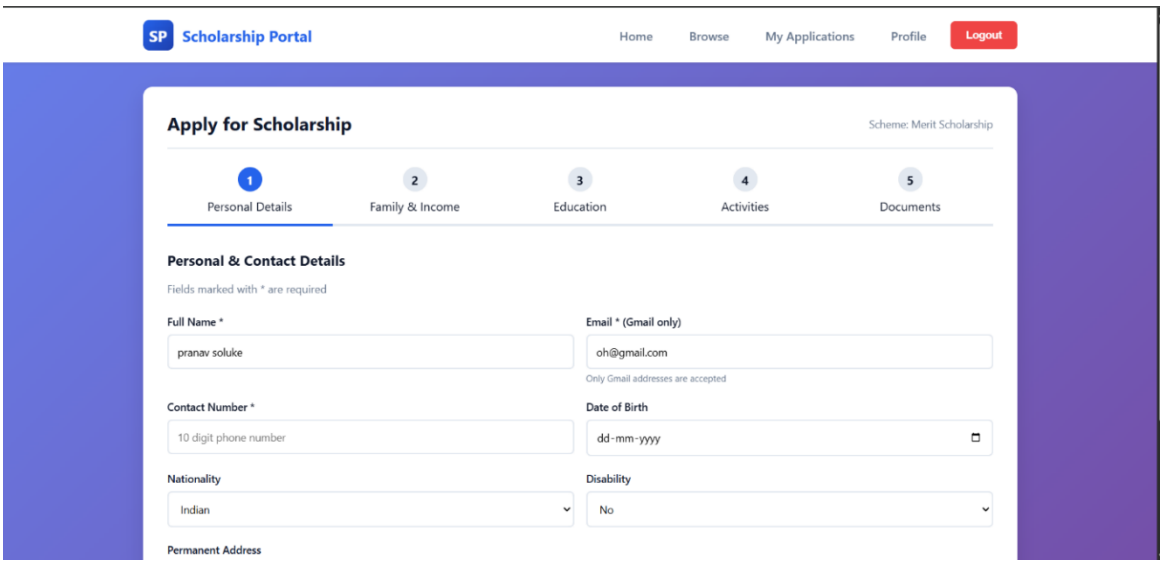
5. Apply.js (Scholarship Application Page)

The Apply page implements a **multi-step form** that collects detailed applicant data such as personal, educational, and financial details — and allows file uploads for documents like photos or mark sheets.

Key features:

- Divided into 5 logical sections (step 1 to step 5).
- Each step validates input before proceeding to the next.
- Uses **React's useState** hook for tracking progress and **FormData API** to handle file uploads.
- On submission, data is sent via a **POST request** to /api/apply.

This page provides users with real-time feedback (loading spinners, success/error messages) to make the submission process transparent and reliable.

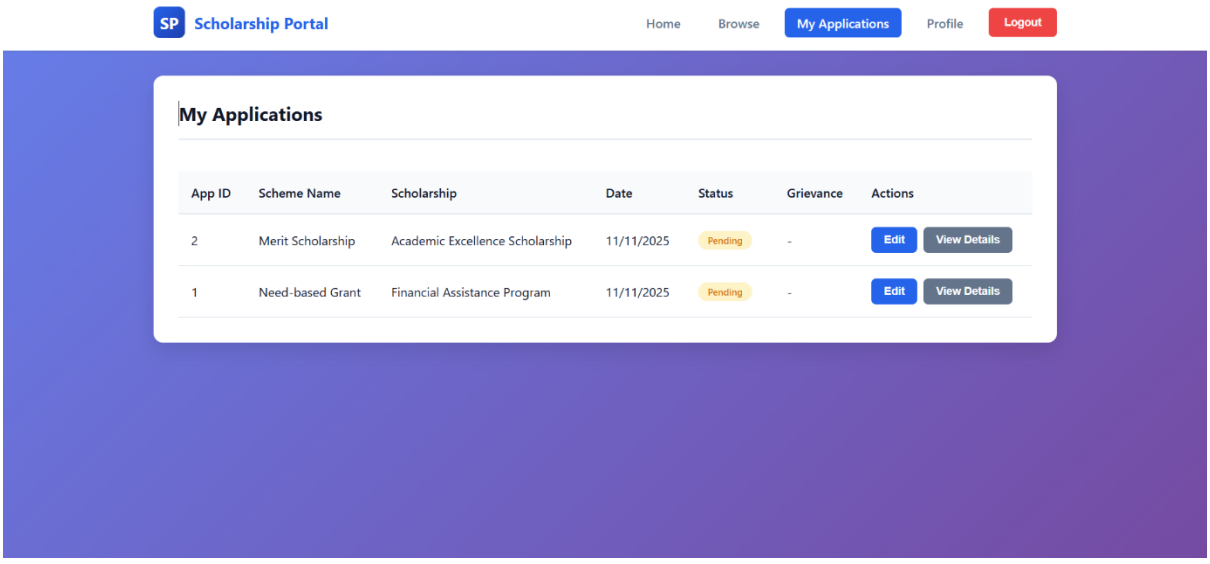


6. Applications.js (User Dashboard Page)

This component shows all applications submitted by the current user.
It sends a GET request to /api/my-applications/:userId and displays results in a structured table.
The table includes:

- Application ID
- Scholarship Name
- Application Date
- Current Status (Pending / Approved / Rejected / Grievance)

Each status is color-coded using styled **badges**.
If an application is in "Grievance" state, the table includes an **Edit** button, redirecting the user to the **EditApplication** page.



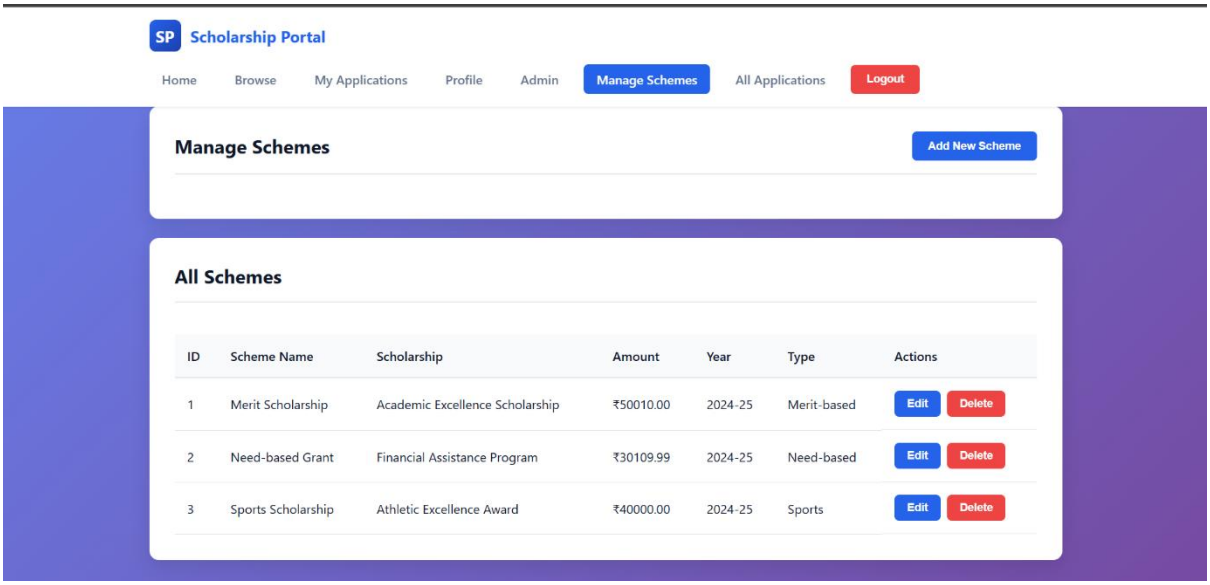
7. EditApplication.js (Resubmission Page)

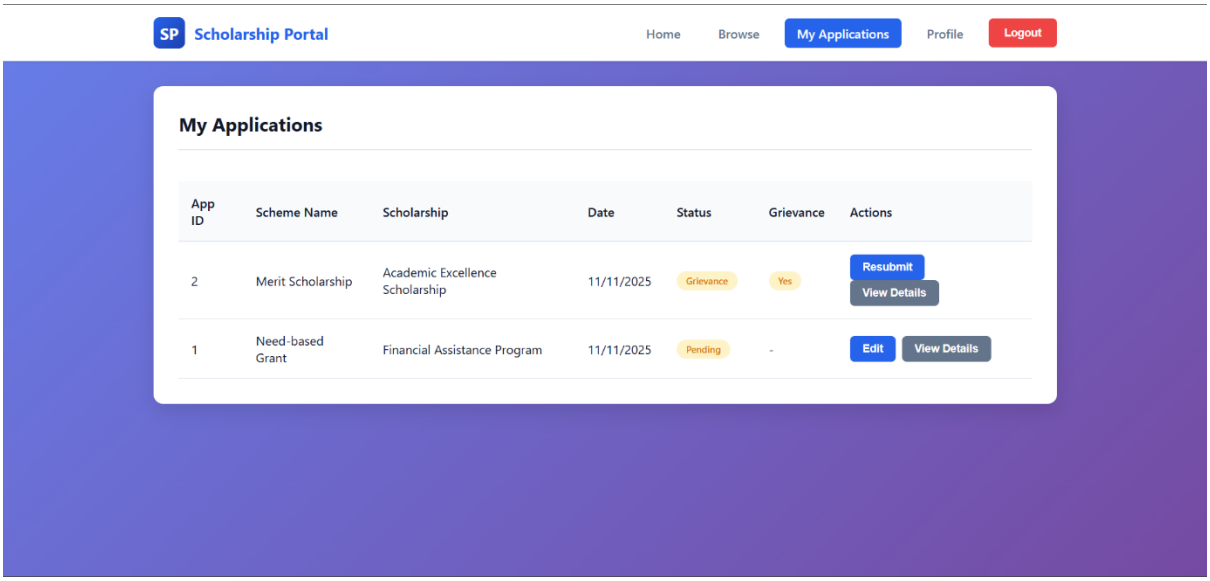
This page supports the **grievance system**, allowing students to edit and resubmit applications after admin feedback.

When an admin marks an application with a grievance message, this page displays that message and unlocks the editable fields.

On submission, a **PUT request** is made to /api/application/:id/edit which updates the record and resets its status to "Pending."

This functionality promotes fairness and transparency, giving users the opportunity to correct and resubmit their applications.





8. AdminDashboard.js

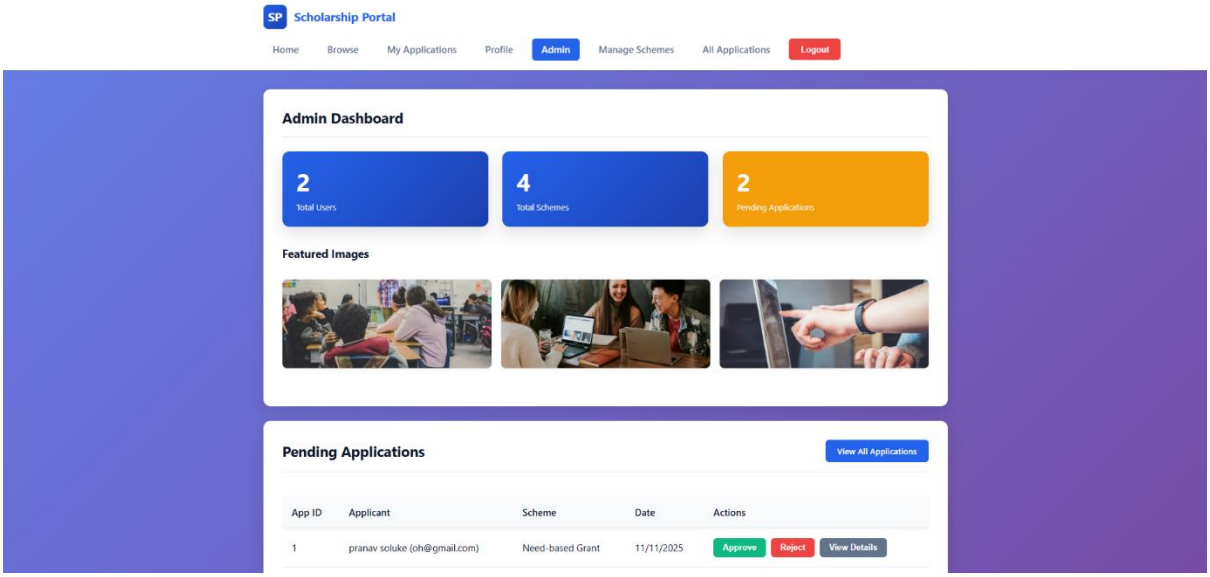
The Admin Dashboard provides a summarized view of system activity using aggregate queries.

It fetches counts for total users, total schemes, and pending applications using /api/admin/stats.

Displayed elements include:

- **Stat cards** showing summary counts.
- A table of **Pending Applications** awaiting review.
- Action buttons (Approve / Reject / Send Grievance) for each application.

This component provides admins with an at-a-glance operational view of the system.



9. AdminSchemes.js

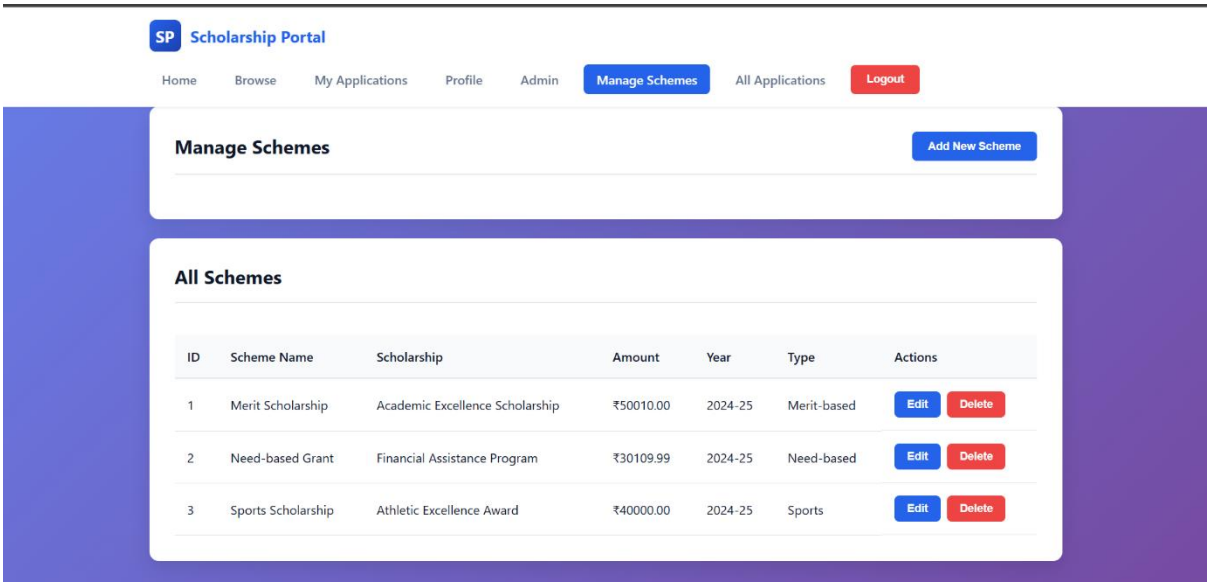
This is a full **CRUD interface** where admins can:

- Create new scholarships
- Update existing ones
- Delete outdated schemes

Each action is handled through modal forms and confirmed via backend routes:

- POST /api/admin/schemes → Add new scheme
- PUT /api/admin/schemes/:id → Update
- DELETE /api/admin/schemes/:id → Delete (with ON DELETE CASCADE automatically removing associated applications)

This page enhances admin efficiency in managing the scholarship lifecycle.



10. AdminApplicationDetails.js

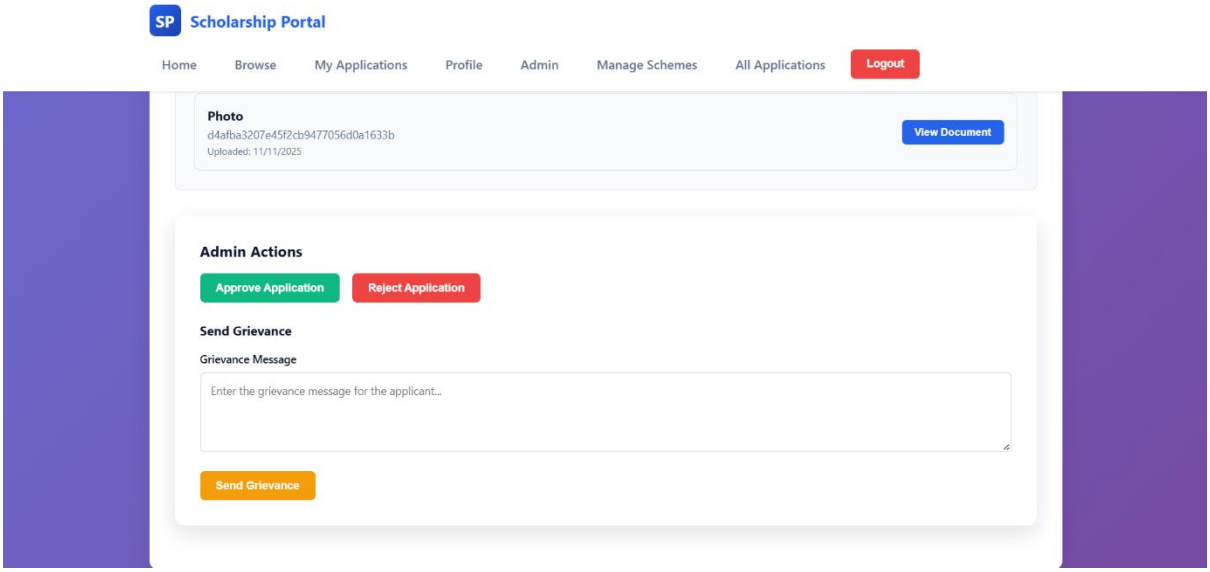
Displays a full, detailed view of a single application.

Admin users can see all user-submitted information (pulled from the data_json column) and uploaded documents.

Actions available:

- Approve the application (/api/admin/application/:id/approve)
- Reject the application (/api/admin/application/:id/reject)
- Send a grievance request (/api/admin/application/:id/grievance)

This module ensures clear and traceable administrative actions.



7.5 Styling with CSS

The styling is defined in styles.css, applying a modern and consistent UI theme throughout the site.

- **Color scheme:** Soft, professional palette with accent colors for buttons and headings.
- **Layout:** Grid and flexbox used for responsive designs.
- **Forms:** Styled with clear input boxes, rounded corners, and focus effects.
- **Media Queries:** Ensure full mobile and tablet compatibility.

Each page maintains a clean layout, ensuring accessibility and visual harmony.

CONCLUSION

The **Scholarship Management System** has been successfully developed and implemented as a comprehensive, database-driven web application designed to streamline the management of scholarship programs for both students and administrators.

This project effectively demonstrates the practical implementation of key **Database Management System (DBMS)** concepts such as data definition, data manipulation, transactions, views, triggers, and database connectivity within a real-world, full-stack environment.

The system's database has been structured with well-defined tables including **users**, **schemes**, **applications**, **documents**, and **notifications**, ensuring clear relationships between entities through the use of **primary and foreign keys**. This relational design upholds **referential integrity** and **data consistency**, forming a robust backbone for the entire application.

Comprehensive **DML operations** — including INSERT, SELECT, UPDATE, and DELETE — are utilized to perform key functionalities such as user registration, scholarship browsing, application submissions, grievance handling, and administrative approvals or rejections.
Transaction Control Language (TCL) commands ensure atomicity during multi-step operations like application submissions, which involve inserting records into multiple related tables simultaneously.

In addition, a **trigger (application_status_update)** has been implemented to automate the insertion of notifications whenever an application's status changes, ensuring real-time communication, transparency, and accountability.
A **view (v_application_details)** has also been created to simplify complex administrative queries by combining data from multiple tables into a single, consolidated virtual table for easy access and analysis.

The backend, developed using **Node.js with Express.js**, connects seamlessly with **MySQL** through the **mysql2/promise** library, providing asynchronous, efficient, and secure database operations using connection pooling and environment-based configurations.

On the other hand, the **front-end**, built using **React.js, CSS, and JavaScript**, offers an intuitive, responsive, and dynamic interface. It enables students to browse active scholarships, apply through a multi-step form, upload documents, and track their application status, while administrators efficiently manage schemes, applications, and grievances through a dedicated dashboard.

In conclusion, this project successfully integrates **DBMS principles with modern web technologies**, delivering a complete and scalable **Scholarship Management System** that enhances transparency, reduces manual work, and ensures accuracy, efficiency, and fairness in scholarship processing.