# Technical Documentation

This document serves as comprehensive technical documentation for the project handover for the Education Management Information system for Karnal.

## 1. Project Overview

This project aims to enhance the management and monitoring of schools by providing insights into academic performance at different levels. District administrators will be able to view overall school performance, while schools can track the performance of their classes, and teachers can monitor individual student progress. The system will track daily attendance, student marks, and subject-level performance, helping educators make informed decisions to improve learning outcomes. The project is currently under development and will provide a structured platform for managing classes, teachers, students, and their academic data.

- Built as a part of Code for Gov tech - Dedicated Mentoring Program 2025
- Project Ticket: https://github.com/Code4GovTech/C4GT/issues/471
- Organization: ADC Karnal

## 2. Tech Stack

This project is built with a modern full-stack architecture combining FastAPI (Python) for the backend and React (Vite) for the frontend.

#### Frontend

- *React 18* – UI library for building dynamic user interfaces

- *Vite* – Next-gen frontend tooling for fast development & builds
- *TailwindCSS* – Utility-first CSS framework for styling
- *shadcn/ui + Radix UI* – Accessible, composable UI components
- *React Router v6* – Routing & navigation
- *React Hook Form + Zod* – Form validation & schema enforcement
- *TanStack Table* – Data table handling & complex data rendering
- *Recharts* – Data visualization and charts
- *PapaParse + xlsx + FileSaver* – CSV/XLSX import/export handling
- *Axios* – API requests

**Backend**

- *FastAPI* – High-performance Python web framework
- *SQLModel* – ORM + Pydantic + SQLAlchemy integration
- *PostgreSQL* – Relational database (SQLite supported for local dev)
- *Alembic* – Database migrations
- *Pydantic v2* – Data validation & settings management
- *Uvicorn + Gunicorn* – ASGI server for production & development
- *Auth stack*
    - JWT (PyJWT + python-jose)
    - Passlib + bcrypt for password hashing
    - OAuth2 for secure authentication flows

**Tooling and Developer Experience**

    - *TypeScript* – Type safety for frontend
    - *ESLint + TypeScript-ESLint* – Linting & code quality
    - *Prettier (optional)* – Code formatting
    - *Rich* – Better logging in backend
    - *dotenv / pydantic-settings* – Config & environment variable management

## 3. Key Features

The system is designed with role-based access control (RBAC). Each role has specific features tailored to their responsibilities.

🏛️ District Admin

- View overall district performance across all schools.
- Monitor and compare school-wise performance trends.
- Access high-level analytics to support decision-making.

🏫 School Admin / Principal

- View the overall performance of their school.
- Create and manage classes.
- Assign teachers to classes.
- Track class-level performance.
- Manage school-wide timetable and schedules.

👩Teacher

- Add and manage students in assigned classes.
- Record daily attendance for their classes.
- Upload and update student marks across subjects.
- Track student-level performance and progress.
- Upload and manage class timetables & schedules.

## 4. Installation Guide

**Prerequisite**

- Node.js >= 18
- Python >= 3.10
- PostgreSQL installed (SQLite works for local dev)

### Backend Setup

Follow these steps to set up the backend locally.

1. *Navigate to backend directory*

```
cd backend
```

2. *Create & activate a virtual environment*
   This isolates project dependencies from your system Python.

```
python -m venv venv

# Linux / macOS
source venv/bin/activate

# Windows
venv\Scripts\activate
```

3. *Install dependencies*

```
pip install -r requirements.txt
```

4. *Configure environment variables*
   Copy the example .env file and update it with your local settings.

```
cp .env.example .env
```

5. *Run database migrations*
   Apply the database schema and ensure tables are created.

```
alembic upgrade head
```

6. *Start the Development Server*
   Run the FastAPI backend locally. You have two options:

   *Option 1*: Using Uvicorn

```
python -m uvicorn src:app --reload --host 0.0.0.0 --port 8000
```

   Option 2: Using FastAPI CLI

```
fastapi run src
```

   💡 Tip: Both commands start the server with hot reload. Use FastAPI CLI if installed for convenience

### Frontend Setup

Follow these steps to set up the frontend locally. By default, the React frontend runs on `localhost:5173` (Vite dev server) and communicates with the FastAPI backend on `localhost:8000`.

1. *Install dependencies*

```
npm install
```

2. *Start the development server*

```
npm run dev
```

## 🐳 **Docker Setup**

Use Docker to run both the backend and frontend in isolated containers. Make sure you have Docker and Docker Compose installed.

1. *Create and configure environment files.*
   Create `.env` files in both the `backend` and `frontend` directories by copying from their respective `.env.example` files.

2. *Run the services.*

   You have two options:

   *Option 1: Build locally (default)*

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

   This will build Docker images for both backend and frontend and start the containers.

   *Option 2: Use prebuilt images (faster)*
   If you prefer not to build locally, you can pull the prebuilt images from Docker Hub:

```
docker pull kajal2005/adc-backend:latest
docker pull kajal2005/adc-frontend:latest
docker compose up
```

   This will start the containers using the prebuilt images, making setup much faster and simpler.

3. *Access the application.*

The frontend will be accessible at http://localhost:5173. The backend API will be running at http://localhost:8000.
To stop the services, press Ctrl + C in the terminal and then run:

```
docker compose down
```

## 5. High level repository structure

This repository contains both frontend and backend code for the Education Management System for Karnal. Below is a high-level overview of the folder and file structure:

```
ADC_KARNAL_EMIMS/
├── backend/                 # FastAPI + SQLModel backend code
│  ├── src/                  # Application source code
│  ├── alembic/              # Database migrations
│  ├── requirements.txt      # Python dependencies
│  └── .env.example          # Example environment variables
| └── README.md              #APIs, models, migrations, and coding standards.
│
├── frontend/ # React (Vite) frontend code (lives at root level)
│  ├── src/               # React components, pages & utilities
│  ├── public/            # Static assets
│  ├── package.json       # Frontend dependencies
│  └── vite.config.ts     # Vite configuration
│  └── README.md          #components, styling conventions, and dev workflow.
│
├── docs/                 # Technical documentation & handover files
├── README.md             # Main project overview & setup guide
├── LICENSE               # Project license file
└── .gitignore            # Git ignore rules
```

## 6. Detailed Backend Architecture

The **EMIMS backend** is built with **FastAPI** and provides a robust, modular, and secure API layer to manage school and district-level educational data.

Key functionalities include:

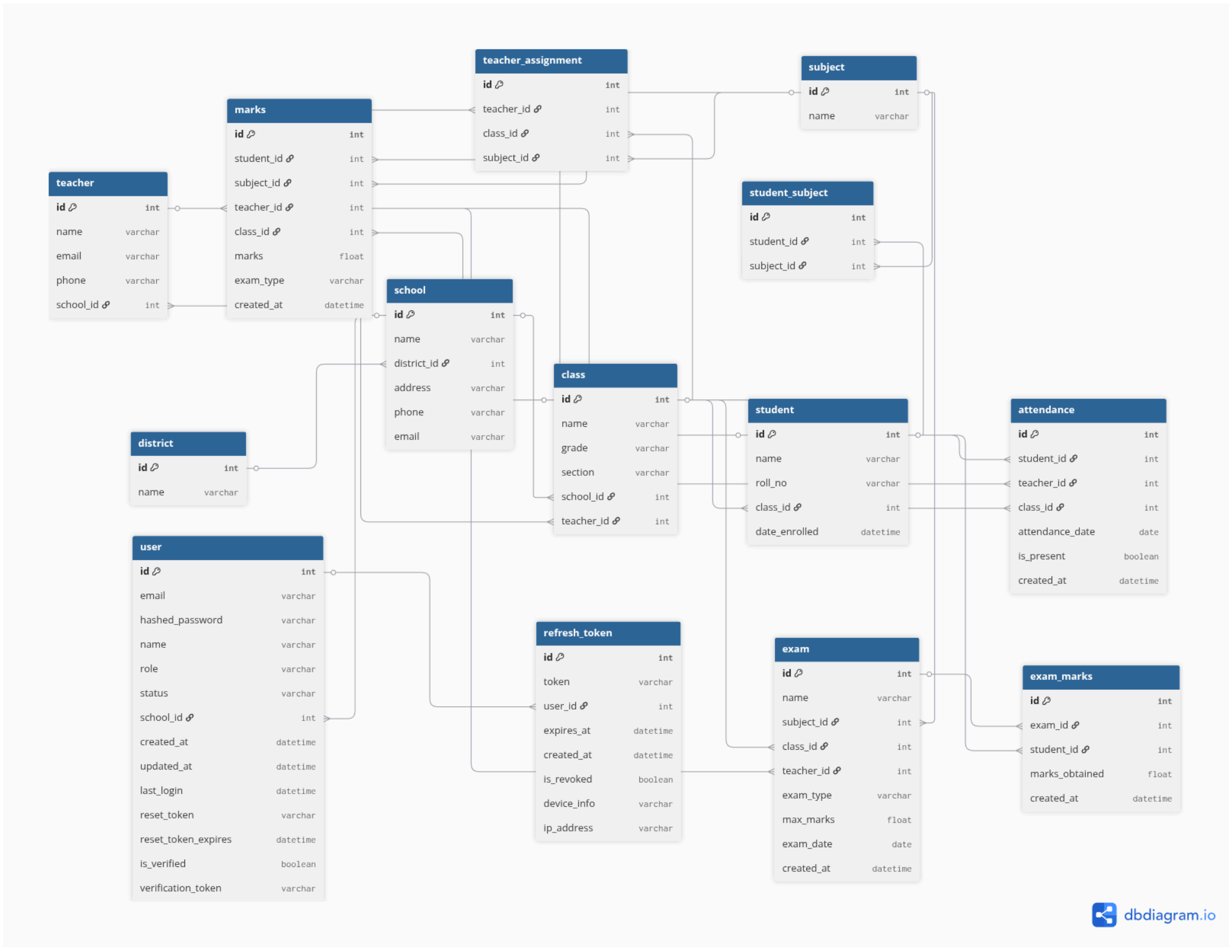
- Role-Based Access Control (RBAC)
- Student attendance and performance tracking
- Teacher and school management
- Dashboard analytics for district administrators
- Secure authentication and session management

The system is designed with **async I/O** for high-performance operations and modular design for easy extensibility.

1. **Folder Structure**

```
ADC_KARNAL_EMIMS/Backend
├── alembic/               # Database migration folder (using Alembic for versioning)
│   ├── versions/          # Individual migration scripts
│   └── env.py             # Alembic environment configuration
├── src/                   # Main application source code
│   ├── auth/              # Authentication & authorization module
│   │   ├── routes.py      # API endpoints for login, signup, token refresh, etc.
│   │   ├── models.py      # Database models for users, roles, tokens, etc.
│   │   ├── dependencies.py  # Dependencies for auth routes (like token verification)
│   │   ├── security.py    # Security utilities (password hashing, JWT handling)
│   │   └── services.py    # Business logic for authentication
│   ├── db/                # Database related logic
│   │   ├── main.py        # Database initialization and connection setup
│   ├── models/            # Models for main application (students, teachers, etc.)
│   │   ├── main.py
│   ├── routes/            # API endpoints for app functionality
│   │   ├── analytics.py   # Analytics-related API endpoints
│   │   ├── attendance.py    # Attendance management endpoints
│   │   ├── classes.py          # Class management endpoints
│   │   ├── dashboard.py        # Dashboard-related endpoints
│   │   ├── exams.py            # Exam and assessment endpoints
│   │   ├── schools.py          # School management endpoints
│   │   ├── subjects.py         # Subjects management endpoints
│   │   ├── students.py         # Student-related endpoints
│   │   └── teachers.py         # Teacher-related endpoints
│   ├── __init__.py             # Makes src a Python package
│   ├── config.py               # Application configuration (settings, environment variables)
│   ├── middleware.py           # Custom middleware (e.g., logging, authentication checks)
│   ├── seed_auth_data.py       # Script to populate initial auth-related data (roles, admin
user)
│   ├── seed_demo_data.py       # Script to populate demo/sample data for testing
├── requirements.txt            # Python dependencies for the project
├── alembic.ini                 # Alembic configuration file for database migrations
├── .env.example                # Example environment variables file
├── Dockerfile                  # Docker configuration to containerize the backend
└── README.md                   # Project documentation and setup instructions
```

## 2. Database Schema



## Relationships

- **District → School:** A district contains many schools.
- **School → Class:** A school has many classes.
- **School → User / Teacher:** A school employs many users and teachers.
- **Class → Student:** A class contains many students.
- **Student → Marks / Attendance / Exam Marks:** A student has many records for marks, attendance, and exam results.
- **Teacher → Teacher Assignment:** A teacher has multiple class and subject assignments.

- **Subject → Teacher Assignment:** A subject can be assigned to multiple teachers.

## 3. Authentication & Security

The system uses JWT-based authentication with refresh tokens and role-based access control (RBAC).

**Features:**

- Password hashing using **bcrypt**
- Short-lived access tokens, long-lived refresh tokens
- Role enforcement: ADMIN, PRINCIPAL, TEACHER
- Device/IP logging for auditing
- Token revocation & unique `jti` identifiers

### Authentication Flow

#### *User Registration*

- Admin/Principal creates user via `/auth/register`
- Passwords hashed & stored securely
- Roles assigned (teacher, principal, admin)

#### *Login*

- Credentials verified via `/auth/login`
- Access & Refresh tokens generated
- Device/IP logged

#### Token Refresh

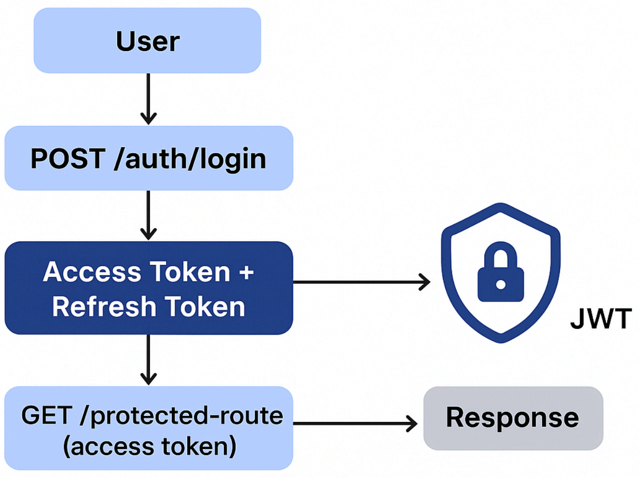
- `/auth/refresh` endpoint generates new access token

### Logout

- `/auth/logout` revokes refresh token

### Password Management

- `/auth/change-password` for active users
- Future: email-based password reset


User
POST /auth/login
Access Token + Refresh Token
JWT
GET /protected-route (access token)
Response

**Authentication & RBAC Workflow**

## 4. API Design & Endpoints

### Base URL : /api/{version}/

### 1. Attendance Endpoints (/attendance.py)

- *POST /: Mark Attendance* 📅
  - Marks attendance for a list of students. It can create new records or update existing ones if attendance for the same student and date already exists.
- *GET /class/{class_id}/date/{attendance_date}:* Get Class Attendance
  - Retrieves attendance records for all students in a specific class on a given date.

- *GET /student/{student_id}/summary:* Get Student Attendance Summary
    - Provides a summary of a student's attendance, including total days, present days, absent days, and attendance percentage.

---

## 2. Classes Endpoints (/classes.py)

- *POST /:* Create Class 🏫
    - Creates a new class with details like name, grade, and section.
- *GET /:* List Classes
    - Fetches a list of all classes. Can be filtered by school_id.
- *GET /{class_id}*: Get Class Details
    - Retrieves detailed information for a specific class, including assigned subjects and teachers.
- *DELETE /{class_id}*: Delete Class
    - Deletes a class record by its ID.

---

## 3. Exams Endpoints (/exams.py)

- *POST /:* Create Exam 📝
    - Creates a new exam record, specifying the subject, class, and maximum marks.
- *GET /teacher/{teacher_id}:* Get Teacher's Exams
    - Lists all exams created by a specific teacher.
- *POST /marks:* Submit Exam Marks
    - Submits or updates exam marks for a list of students for a given exam.
- *GET /{exam_id}/marks:* Get Exam Marks
    - Retrieves all marks submitted for a specific exam.

- *GET /student/{student_id}/performance:* Get Student Exam Performance
  - Provides a detailed performance summary for a student across all their exams, including marks, percentages, and subjects.

---

## 4. Schools Endpoints (/schools.py)

- *POST /:* Create School 🏢
  - Adds a new school record to the database.
- *GET /:* List Schools
  - Fetches a list of all schools. Can be filtered by district_id.
- *GET /{school_id}:* Get School Details
  - Retrieves details for a specific school.

---

## 5. Students Endpoints (/students.py)

- *POST /:* Create Student 🧑‍🎓
  - Adds a new student record to a class.
- *GET /:* List Students
  - Fetches a list of all students. Can be filtered by class_id.
- *GET /{student_id}:* Get Student Details
  - Retrieves information for a specific student.
- *PUT /{student_id}*: Update Student
  - Updates a student's information.
- *DELETE /{student_id}:* Delete Student
  - Deletes a student record.
- *POST /subjects:* Enroll Student in Subject
  - Enrolls a student in a specific subject.
- *POST /marks:* Record Marks
  - Records marks for a student in a specific subject and exam type.

- *GET /{student_id}/marks:* Get Student Marks
    - Retrieves all marks for a student, with an optional filter for subject_id.

---

## 6. Subjects Endpoints (/subjects.py)

- *POST /:* Create Subject 📚
    - Adds a new subject to the master list.
- *GET /:* List Subjects
    - Fetches a list of all subjects.
- *GET /{subject_id}:* Get Subject Details
    - Retrieves details for a specific subject.
- *PUT /{subject_id}:* Update Subject
    - Updates a subject's information.
- *DELETE /{subject_id}:* Delete Subject
    - Deletes a subject record.

---

## 7. Teachers Endpoints (/teachers.py)

- *GET /:* List Teachers 🧑‍🏫
    - Fetches a list of all teachers. Can be filtered by school_id.
- *GET /{teacher_id}/classes:* Get Teacher's Classes
    - Retrieves a list of all classes and subjects assigned to a specific teacher.
- *POST /assignments:* Assign Teacher to Class/Subject
    - Creates or updates a TeacherAssignment to link a teacher to a specific class and subject.
- *DELETE /assignments/{assignment_id}:* Remove Assignment
    - Deletes a teacher's class/subject assignment.

**Swagger UI:** `/docs`
**ReDoc:** `/redoc`

## 5. Database setup and Migrations

This app uses SQLAlchemy's async engine and requires an async PostgreSQL driver (`asyncpg`).

*Your `.env` should have:*

```
DATABASE_URL=postgresql+asyncpg://username:password
@host/dbname
```

**Alembic Commands:**

```
# Generate new migration

alembic revision --autogenerate -m "Add marks table"

# Apply migrations

alembic upgrade head
```

## 5. Deployment

- Backend containerized with Docker
- Production: gunicorn -k uvicorn.workers.UvicornWorker
- .env used for environment-specific configs
- Future: CI/CD pipeline integration recommended

### 6. Security Considerations

- Password hashing with bcrypt
- Expiring JWT tokens & refresh mechanism
- RBAC for API access
- Input validation with Pydantic
- HTTPS required in production
- Logging of failed login attempts

### 7. Performance & Optimizations

- Async endpoints for high throughput
- PostgreSQL connection pooling
- Query optimization using `selectinload` for relations
- Caching planned with Redis for future improvements

### 8. Future Improvements

- Real-time WebSocket notifications (attendance, marks updates)
- Admin dashboard for token & session management
- Password reset & email verification flows
- Security analytics & monitoring tools
- Integration with Government EMIS APIs

## 7. Detailed Frontend Architecture

The frontend of EMIMS is built using **React 18** with **Vite** as the build tool.

It offers a modular and responsive UI for multiple roles — District Admin, School Admin, and Teachers — integrating directly with the FastAPI backend.

The UI uses modern component libraries, robust form handling, and real-time data visualization.

1. **Folder Structure**

```
ADC_KARNAL_EMIMS
src/
├── App.tsx              # Root app component
├── main.tsx             # React entry point
├── index.css            # Global styles
├── vite-env.d.ts         # Vite TypeScript environment declarations

├── components/           # Reusable UI components
│   ├── auth/             # Authentication components
│   │   ├── LoginForm.tsx    # Login form
│   │   ├── ProtectedRoute.tsx  # Wraps routes to enforce auth
│   │   └── RoleBasedComponent.tsx  # Conditional rendering by role
│   │
│   ├── dashboard/         # Dashboard widgets & charts
│   │   ├── AlertsWidget.tsx
│   │   ├── PerformanceChart.tsx
│   │   ├── StatCard.tsx
│   │   └── TeacherWorkload.tsx
│   │
│   ├── layout/            # Layout components
│   │   ├── Footer.tsx
│   │   ├── Header.tsx
│   │   ├── Layout.tsx      # Main layout wrapper (Sidebar + Header + Content)
│   │   └── Sidebar.tsx
│   │
│   ├── shared/            # Shared utility components
│   │   ├── DataTable.tsx
│   │   ├── EmptyState.tsx
│   │   ├── ExportButton.tsx
│   │   ├── FileUpload.tsx
│   │   ├── FilterBar.tsx
│   │   └── LoadingSpinner.tsx
│   │
│   ├── ui/              # Low-level UI primitives
│   │   ├── dialog.tsx
│   │   └── tabs.tsx
│   │
│   └── visualizations/     # Charts specific to performance & progress
│       ├── SchoolPerformanceChart.tsx
│       ├── StudentProgressChart.tsx
│       └── SubjectScoresByClass.tsx
│
├── contexts/             # React Context API for global state
│   ├── AuthContext.tsx     # Authentication & roles
│   ├── DataContext.tsx     # Table/filter states
│   └── SchoolContext.tsx    # School-related global data
│
├── lib/                  # Utility libraries & helpers
│   └── utils.ts            # Generic helper functions
│
├── pages/                # Route-level page components
│   ├── Analytics.tsx
│   ├── ClassManagement.tsx
│   ├── Dashboard.tsx
│   ├── DataManagement.tsx
│   ├── NotFound.tsx
│   ├── Reports.tsx
│   ├── SchoolDetail.tsx
│   ├── Schools.tsx
│   ├── Settings.tsx
│   ├── Students.tsx
│   ├── Subjects.tsx
│   ├── TeacherDashboard.tsx
│   └── Teachers.tsx
│
├── services/             # API integration & business logic
│   ├── api.ts            # Axios instance & API helpers
│   └── auth.ts          # Auth-specific API calls

└── types/              # TypeScript type definitions
    └── auth.ts
```

**2. Authentication Flow**

***Process:***

- *User Login*: A user submits credentials via LoginForm.tsx, which triggers an API call to the backend's /auth/login endpoint.
- *Token Storage*: Upon successful authentication, the backend returns a JWT token. This token is securely stored in the browser's localStorage for persistence.
- *Global State:* The AuthContext consumes the token, updating the global state to reflect the user and isAuthenticated status, making this information accessible to any component.
- *Route Protection:* The ProtectedRoute.tsx component is used to wrap routes that require authentication, automatically redirecting unauthenticated users to the login page.
- *Role-Based Access*: The RoleBasesComponent.tsx component, along with AuthContext, handles conditional rendering and navigation based on the user's role (e.g., District Admin, School Admin, Teacher).

**Role-based navigation:**

*Roles Default Dashboard*

- District Admin --> /district-dashboard
- School Admin --> /school-dashboard
- Teacher --> /teacher-dashboard

**Security:**

- JWT auto-attached to all API calls
- Role-based UI & route protection
- Secure logout clears all tokens

**3. State Management**

The application uses a lightweight React Context API for state management, avoiding the overhead of libraries like Redux.

- *AuthContext*: Manages the authentication state, user roles, and related data.
- *DataContext*: Provides global access to core application data, such as a list of schools or classes, preventing prop drilling.
- *SchoolContext*: Manages state specific to the currently selected school.

**4. UI & Styling**

The user interface is built using a modern, component-based approach.

- *Frameworks*: Styling is handled with TailwindCSS for utility-first class management, and shadcn/ui provides pre-styled, accessible UI components.
- *Design*: The design is responsive, ensuring a consistent and user-friendly experience on various devices, from desktops to tablets.
- *Theming*: The application maintains a consistent visual identity through a defined color palette, typography, and spacing conventions.

**5. Data Visualization**

Data is presented in a clear and intuitive way using charts and graphs powered by the Recharts library.

Performance Charts:

- *SchoolPerformanceChart.tsx*: Displays high-level school performance metrics.
- *StudentProgressChart.tsx*: Visualizes individual student progress over time.
- *SubjectScoresByClass.tsx*: Compares student scores across different subjects within a class.

**6. Data Import & Export**

The system provides functionality for efficient data handling.

- **Export**: Users can export data, such as student marks, into standard formats like .csv or .xlsx using ExportButton.tsx and libraries like xlsx and FileSaver.
- **Import**: The FileUpload.tsx component allows teachers to upload updated marks or other data via CSV files, which are parsed by the PapaParse library.

**7. API Integration**

The application communicates with the backend via a centralized API service.

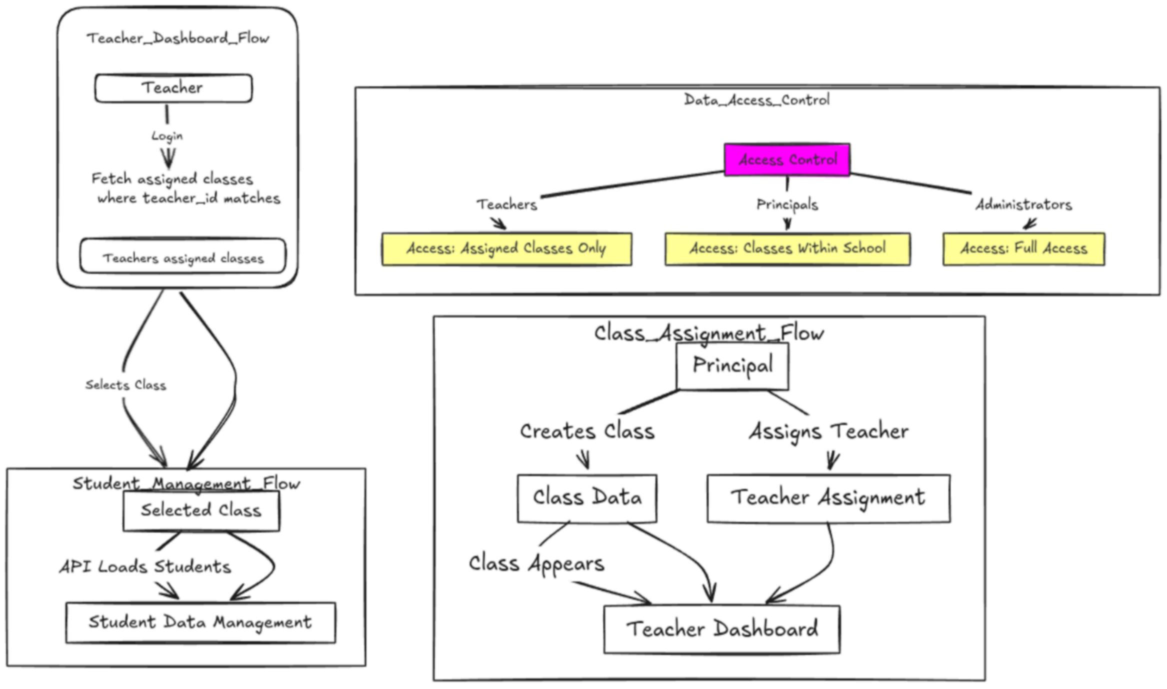
- **Service Layer**: All API calls are defined in /services/api.ts and /services/auth.ts, creating a clean separation of concerns.
- **Axios**: An Axios instance is configured to automatically attach the JWT token to the Authorization header of every request, simplifying authenticated API calls.

**8. Routing Architecture**

Centralized Routing: Routing is managed in App.tsx using react-router-dom.

- *Protected Routes*: The ProtectedRoute.tsx component is used to secure specific routes, ensuring only authenticated users can access them.
- *Role-Based Routing*: Different roles are directed to their respective dashboards (e.g., /teacher-dashboard, /school-dashboard).

## 9. Workflow diagram



## 10. Build and Deployment

```
npm run dev → http://localhost:5173
```

```
# Production

npm run build
```

Production build served via Nginx or Node

Dockerized setup supported

**11. Performance Optimizations**

- Lazy Loading: Routes are lazy-loaded to reduce the initial bundle size.
- Code Splitting: The build process automatically splits code into smaller chunks.
- Memoization: React.memo and useCallback are used to memoize components and functions, preventing unnecessary re-renders.

**12. Future Enhancements**

- Advanced Data Visualization: Integrate heatmaps and drill-down charts for more detailed performance analysis.
- Real-time Dashboard: Implement WebSockets for live data updates on dashboards.
- Offline Support: Enable offline caching with Service Workers for data access without an internet connection.
- Progressive Web App (PWA): Make the app installable with push notifications and other PWA features.
- Multilingual Support: Add i18n support for multiple languages, starting with Hindi.

**13. Code Standards**

To ensure maintainability and scalability, the following standards are enforced:

- TypeScript: Strict typing is used throughout the codebase to catch errors early.
- Linting & Formatting: ESLint and Prettier are configured to maintain consistent code style and enforce best practices.
- Component Structure: Components are organized in a folder-based structure, grouping related files together.
- Hooks: The codebase follows modern React practices, using functional components and React Hooks extensively.
- Validation: Zod is used for robust runtime schema validation, ensuring data integrity.

**Note on Code Style**

For consistency with the rest of the project's existing code, I have used React.FC for the new components.

I am aware that React.FC is considered deprecated in modern React and TypeScript best practices. In future we plan to refactor the entire codebase to use the simpler function component syntax once the core features are stable and finalized.

## Contributing

We welcome contributions! Please follow these steps:

1. Fork the repository.
2. Create a new feature branch (`git checkout -b feature/my-feature`).
3. Commit your changes (`git commit -m "Add new feature"`).
4. Push to your branch (`git push origin feature/my-feature`).
5. Open a Pull Request.