Task: Build a School Management System API

Requirements:

1. **Database Schema:**
    - Use **PostgreSQL** as the database.
    - Design and manage the database using **TypeORM**.
    - Define entities with proper relationships:
        - **`Users`**:
            - Roles: `admin`, `teacher`, `student`, and `bursar` (use an ENUM or a separate `Roles` table).
            - Attributes: `id`, `name`, `email`, `password`, `role`, and timestamps.
            - Relationships:
                - A `teacher` can be assigned to multiple `classes` and `subjects`.
                - A `student` belongs to a single `class`.
                - An `admin` oversees the system.
                - A `bursar` manages `payments`.
        - **`Classes`**:
            - Attributes: `id`, `name`, `description`, and timestamps.
            - Relationships:
                - A `class` has many `students`.
                - A `class` can have multiple `teachers` (subject-specific).
        - **`Subjects`**:
            - Attributes: `id`, `name`, `description`, and timestamps.
            - Relationships:
                - A `subject` is taught by one or more `teachers`.
                - A `subject` is associated with a `class`.
        - **`Payments`**:
            - Attributes: `id`, `studentId`, `amount`, `status` (pending/paid), `paymentDate`, and timestamps.
            - Relationships:
                - A `payment` is linked to a `student`.
    - Ensure proper **foreign key constraints**, cascading deletions, and indexes for optimized performance.

2. **API Features:**
    - **Authentication and Authorization:**
        - Implement user authentication using **JWT**.
        - Role-based access control (e.g., only admins can create classes, only bursars can manage payments).
    - **CRUD Operations:**
        - **Users:** Create, update, delete, and fetch users by role.
        - **Classes:** Create, update, delete, and fetch classes with enrolled students and assigned teachers.
        - **Subjects:** Create, update, delete, and fetch subjects by class or teacher.
        - **Payments:** Record payments, check payment history, and fetch unpaid dues.
    - **Search and Filtering:**
        - Retrieve students by class or payment status.
        - List teachers teaching a particular subject or class.
    - **Reporting:**
        - Total payments collected by month/year.
        - Outstanding fees per class.
3. **API Documentation:**
    - Use **Swagger** to document the API.
    - Follow **OpenAPI Guidelines** for structuring the documentation.
    - Include:
        - Clear descriptions for each endpoint.
        - Examples of request and response payloads for each operation.
        - Response status codes and their meanings (`200`, `201`, `400`, `401`, `404`, `500`).
        - Proper tagging of endpoints (e.g., `Users`, `Classes`, `Subjects`, `Payments`).
        - Default and error responses for all endpoints.
4. **TypeORM Implementation:**
    - Use **TypeORM decorators** to define entity schemas.
        - Example: Use `@Entity`, `@PrimaryGeneratedColumn`, `@Column`, `@ManyToOne`, `@OneToMany`, `@ManyToMany`, and `@JoinTable` as appropriate.
    - Create migration files using **TypeORM CLI** for schema updates.
    - Utilize **repositories** for database queries and ensure data consistency.
5. **Additional Tasks:**
    - **Pagination:** Implement pagination for fetching large lists (e.g., students, payments).
    - **Error Handling:** Add comprehensive error handling for invalid inputs, unauthorized access, and database issues.
    - **Validation:** Validate incoming data using DTOs with **`class-validator`**.
6. **Extras for Advanced Students:**

- Implement an email notification system to alert students or parents about pending payments.
- Use **WebSocket** or **GraphQL subscriptions** to notify the admin or bursar about real-time payment updates.
- Add a search endpoint that uses full-text search for querying users or classes.

7. **Testing:**

   - Write unit tests and integration tests for major endpoints.

8. **Swagger Implementation Guide:**

   - Use `@nestjs/swagger` module to integrate Swagger in the NestJS project.
   - Include the Swagger UI accessible at a specific endpoint (e.g., `/api-docs`).
   - Generate OpenAPI specifications using decorators such as `@ApiTags`, `@ApiProperty`, `@ApiResponse`, and `@ApiOperation`.
   - Set up Swagger security to handle JWT authentication (e.g., using `Bearer` token).