Full Stack Developer Assessment

PART 1 – BACKEND

Objective

Develop a **microservice** for a **School Payment and Dashboard Application**. The focus of this assessment is to create a **REST API** for managing **Transactions and Payments**.

🛠️ Project Setup

1. Initialize Project

- Use **Node.js** with **NestJS** (or a similar framework like Express, Fastify, etc.)
- Connect the application to **MongoDB Atlas**

🛢️ Database Schemas

Use MongoDB atlas for DB URI

I. Order Schema

Stores order-related information.

Fields	**types**
_id	Object_id
school_id	Object_id /string
trustee_id	Object_id /string

student_info	{
	name:string,
	id:string,
	email:string
	}

gateway_name	String

II. Order Status Schema

Stores payment transaction information.

Fields	**types**
collect_id	ObjectId (Reference to Order schema (_id))
order_amount	number
transaction_amount	number
payment_mode	string
payment_details	string
bank_reference	string
payment_message	string
status	string
error_message	string
payment_time	Date and time

III. Webhook Logs Schema

Custom schema to store webhook-related logs.

🔐 User Authentication (JWT)

- Create a **User Schema** to store login credentials.
- Secure all API endpoints using **JWT Authentication**.

TASKS

🔁 Payment Gateway Integration

API Documentation:

Refer to this document:

📄 [Payment API Docs]

Integration Flow:

- **Implement a POST /create-payment route.**
- **Accept payment details from the user.**
- **Forward data to the payment API using create-collect-request**
- **Generate JWT-signed payloads as required.**
- **Redirect the user to the payment page from the API response.**
- **hint: read document carefully and implement create-collect-request/ create transaction and redirect user to payment page(link will be in response of API) , use jsonwebtoken of sign**

Credentials for Payment API

- pg_key : edvtest01
- API KEY : eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0cnVzdGVlSWQiOiI2NWIwZTU1MmRkMzE5NTBhOWI0MWM1YmEiLCJJbmRleE9mQXBpS2V5Ijo2LCJpYXQiOjE3MTE2MjIyNzAsImV4cCI6MTc0MzE3OTg3MH0.Rye77Dp59GGxwCmwWekJHRj6edXWJnff9finjMhxKuw
- school_id : 65b0e6293e9f76a9694d84b4

🌐 Webhook Integration

Create a POST route to update transactions details in DB with the given payload

Endpoint:

POST /webhook

Payload Format:

```
{
  "status": 200,
  "order_info": {
    "order_id": "collect_id/transaction_id",
```

```
  "order_amount": 2000,
  "transaction_amount": 2200,
  "gateway": "PhonePe",
  "bank_reference": "YESBNK222",
  "status": "success",
  "payment_mode": "upi",
  "payemnt_details": "success@ybl",
  "Payment_message": "payment success",
  "payment_time": "2025-04-23T08:14:21.945+00:00",
  "error_message": "NA"
}
```


Actions:

- **Parse the webhook payload.**
- **Update the corresponding Order Status entry in MongoDB.**

Tip: Use Postman to simulate webhook calls.

API Endpoints

1. Fetch All Transactions

- **GET /transactions**
- **Use MongoDB aggregation pipeline to combine order and order_status schemas.**

- **Fields to return:**
    - **collect_id**
    - **school_id**
    - **gateway**
    - **order_amount**
    - **transaction_amount**
    - **status**
    - **custom_order_id**

Note: Populate dummy data in both schemas for testing.

this API will return data by combining to schemas order and order status use mongodb pipeline to join these two schemas

2. Fetch Transactions by School

- **GET /transactions/school/:schoolId**
- **Returns all transactions related to a specific school.**

---

3. Check Transaction Status

- **GET /transaction-status/:custom_order_id**
- **Returns the current status of the transaction.**

✅ Additional Notes

- **Data Validation & Error Handling**
  Ensure proper validation of all incoming data using validation libraries (e.g., class-validator for

NestJS). Implement consistent and informative error responses across all endpoints.

- **Environment Configuration**
  Use .env files and the ConfigModule in NestJS (or similar) to manage environment variables. This includes sensitive keys like:
    - MongoDB Atlas connection string
    - Payment API credentials (API key, PG key)
    - JWT secret and expiry time
- **README & Documentation**
  Include a comprehensive README.md file with:
    - Setup and installation instructions
    - API usage examples
    - Environment variable configuration
    - Postman collection for testing
- **Scalability & Performance**
    - **Pagination**: Implement pagination for all list endpoints (e.g., /transactions) using limit and page query parameters.
    - **Sorting**: Support sorting by fields such as payment_time, status, or transaction_amount via query parameters (e.g., sort=payment_time&order=desc).
    - **Indexing**: Ensure important fields like school_id, custom_order_id, and collect_id are indexed in MongoDB to speed up queries.
- **Security Best Practices**
    - Use **JWT Authentication** for all protected routes.
    - Sanitize and validate incoming requests to prevent injection attacks.
    - Use HTTPS in production and set appropriate CORS policies.
- **Robust Logging**

- Log incoming webhook events and failed transactions for audit and debugging.

Submission Guidelines

1. **Hosting**
   - Host your project on a cloud platform (e.g., Heroku, AWS, or similar) to make it accessible via a public URL.
2. **GitHub Repository**
   - Push your code to a public GitHub repository.
   - Include comprehensive API documentation in the repository's README file, explaining how to use each endpoint.
   - Provide env file
3. **Submission Links**
   - Share the hosted project link along with the GitHub repository URL.

PART 2- FRONTEND

Objective:
Develop a responsive and user-friendly interface for the School Payments and Dashboard application. The focus is on creating a React-based frontend that integrates with the backend APIs to display and manage data dynamically.

Requirements

1. Project Setup

- Use **React.js**
- Set up a new project using **Vite** or **Create React App**.
- Style the application with **Tailwind CSS** or any CSS framework of your choice (e.g., Material-UI, Bootstrap).
- Use **Axios** for API calls and **React Router** for navigation.

2. Core Features

a) Dashboard Pages

1. **Transactions Overview**
    - Display a paginated and searchable list of all transactions fetched from the /transactions API.
    - Columns:
        - collect_id
        - school_id
        - gateway
        - order_amount
        - transaction_amount
        - status
        - custom_order_id
    - Include a filter dropdown for status (e.g., "Success", "Pending", "Failed").
    - Multi-select filters for status and school IDs. And date-wise transactions
    - Column sorting **(asc/desc).**
    - Persist filters in URL so users can share specific views or reload without losing filter states.
2. **Transaction Details by School**
    - Create a page that fetches and displays transactions for a specific school_id.

- Provide a dropdown or search bar to select a school_id.

3. **Transaction Status Check**
   - A dedicated page or modal where users can input custom_order_id and retrieve the current status of a transaction using the check-status API.

3. Optional Features (Extra Credit)

a) Real-time Data Visualization

b) Dark Mode

- Implement a toggle for light/dark themes.

Additional Requirements

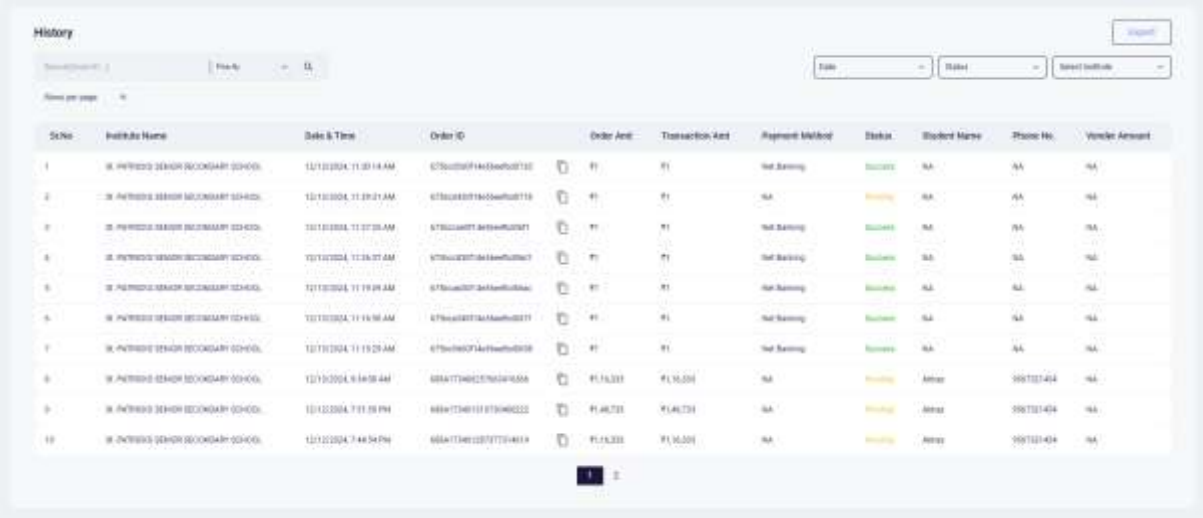
1. **Hosting**
   - Deploy the frontend app to a cloud hosting service like **Netlify**, **Vercel**, or **AWS Amplify**.
2. **GitHub Repository**
   - Push the frontend code to a public GitHub repository.
   - Include a comprehensive README file:
     - Project setup instructions.
     - Detailed documentation for each page and its functionality.
     - Screenshots of the app.

Submission Guidelines

- Hosted App URL.
- GitHub Repository URL with clear documentation.

Reference Image

1) Below is an example design for the transaction table layout:



2) Open the following link, watch the video, and try to create a similar hover effect in a table

[Video Link]