Salla Assesment

Task 1: Case Study: Build a Simple Billing App for a SaaS Platform

Note:

I had planned to build the SaaS billing app architecture using microservices with an event-driven approach, leveraging gRPC and Kafka. However, due to some office deadlines, I ended up going with a monolithic architecture instead.

Problem Statement:

Your task is to design and implement a billing app for a SaaS platform that supports multiple subscription tiers and handles recurring billing.

Schema Design:

1. Customer Table

- `id`: Primary Key, unique identifier for each customer. This is essential for uniquely identifying customer records across the database.
- `name`: Name of the customer. This is important for personalization and communication purposes.
- `email`: Customer's email address. This is critical for communication, login identification, and password resets.
- `created_at`: Timestamp of when the customer record was created. This helps in tracking when the customer joined the service.
- `updated_at`: Timestamp of the last update made to the customer's record. Useful for auditing changes.
- `subscriptionPlanId`: Foreign key linked to the SubscriptionPlan table. It determines which plan the customer is currently subscribed to.
- `subscription_status`: Status of the customer's subscription (e.g., active, canceled, pending). Useful for managing access to services and billing.

- `subscription_start_date`: This field represents the date when the customer's subscription begins.
- `subscription_end_date`: This field indicates the date when the customer's current subscription period will end
- `last_payment_date`: Date of the last payment. This helps in determining billing cycles and delinquency.

2. SubscriptionPlan Table

- `id`: Primary Key, unique identifier for each subscription plan.
- `name`: Descriptive name of the plan (e.g., Basic, Pro, Enterprise). Helps customers understand the tier of service.
- `price`: Monthly price of the subscription plan. Necessary for billing purposes.
- `duration`: Duration of the subscription plan (typically in months). Useful for determining when the subscription renews or expires.
- `billing_cycle`: This field defines how often the customer is billed for the subscription plan. This is crucial for setting up automated billing, invoice generation, and subscription management
- `status`: Status of the plan (active/inactive). Helps manage availability of the plan for new or upgrading customers.
- `features`: Description of features provided in the plan. Useful for sales and customer support to explain plan capabilities.

3. Invoice Table

- `id`: Primary Key, unique identifier for each invoice.
- `customerId`: Foreign key linked to the Customer table. Identifies the customer to whom the invoice is issued.
- `subscriptionPlanId`: Foreign key linked to the SubscriptionPlan table. Details the plan for which the invoice is issued.
- `amount`: Total amount charged on the invoice. Necessary for financial records and customer billing.

- `issue_date`: Date when the invoice is issued.
- `due_date`: Date by which the payment should be made. Important for reminding customers and managing collections.
- `payment_date`: Date when the payment was received. Useful for reconciling accounts and managing cash flow.
- `status`: Status of the invoice (e.g., paid, unpaid, overdue). Key for tracking the payment lifecycle.

4. Payment Table

- `id`: Primary Key, unique identifier for each payment.
- `invoiceId`: Foreign key linked to the Invoice table. Ensures that the payment is correctly associated with an invoice.
- `amount`: Amount paid. This should match the invoice amount or part of it if the payment is partial.
- `payment_date`: Timestamp of when the payment was made. Critical for financial records.
- `payment_method`: Method of payment (e.g., credit card, PayPal, bank transfer). Important for processing and reconciliation.
- `status`: Status of the payment (e.g., successful, failed, pending). Helps in managing payment processing and troubleshooting.

Core Technologies Powering Our SaaS Platform

Backend: Node Js

Database: PostgreSQL

Framework: Nest Js

Step-by-Step Guide to Setting Up Your SaaS Billing Platform

Step # 1 : Installing Docker and Docker Compose:

Start by downloading Docker, which includes Docker Compose as part of its desktop installation for Windows and Mac. For Linux users, Docker Compose must be installed separately. This setup will enable you to manage containerized applications smoothly. Visit the official Docker website to download the appropriate installer for your operating system and follow the provided installation instructions to set up both Docker and Docker Compose.

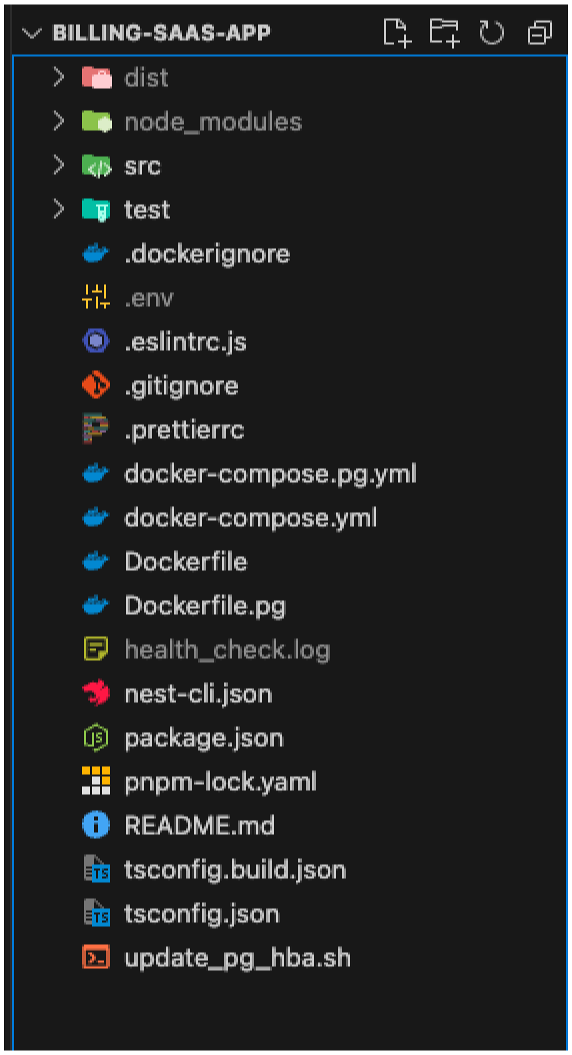
Step # 2 : Clone the Billing SaaS Application Repository:

This step involves cloning your SaaS billing application from its GitHub repository.

```
https://github.com/tahakhan-dev/billing-saas-app.git
```

Step # 3: Navigate to the Project's Root Directory

After cloning the repository, switch to the project's root folder to begin configuration and setup.



```
BILLING-SAAS-APP
  > dist
  > node_modules
  > src
  > test
    .dockerignore
    .env
    .eslintrc.js
    .gitignore
    .prettierrc
    docker-compose.pg.yml
    docker-compose.yml
    Dockerfile
    Dockerfile.pg
    health_check.log
    nest-cli.json
    package.json
    pnpm-lock.yaml
    README.md
    tsconfig.build.json
    tsconfig.json
    update_pg_hba.sh
```

Step # 4: Set Environment Variables for Docker Compose

Configure the required environment variables to ensure the Docker Compose setup runs correctly.

```
PORT=3000
CLUSTER_ENV=dev

#----- Database Credential------------------
DB_HOST=host.docker.internal
DB_PORT=5433
DB_USER=tahakhan
DB_PASSWORD=
DB_DATABASE=postgres
DB_TYPE=postgres
ENABLE_AUTOMATIC_CREATION=true
AUTO_LOAD_ENTITIES=true

JWT_SECRET=6502f2502a8b22bbbd724cd4efedcbe7fbdf47410cbb385e69c6


# ----------- EMAIL CREDENTIAL --------------------

SMTP_HOST=smtp.example.com
EMAIL_USER=user
EMAIL_PASSWORD=your-email-password
SENDER_ADDRESS=send-email-address
```

Step # 5: Launch PostgreSQL with Docker Compose

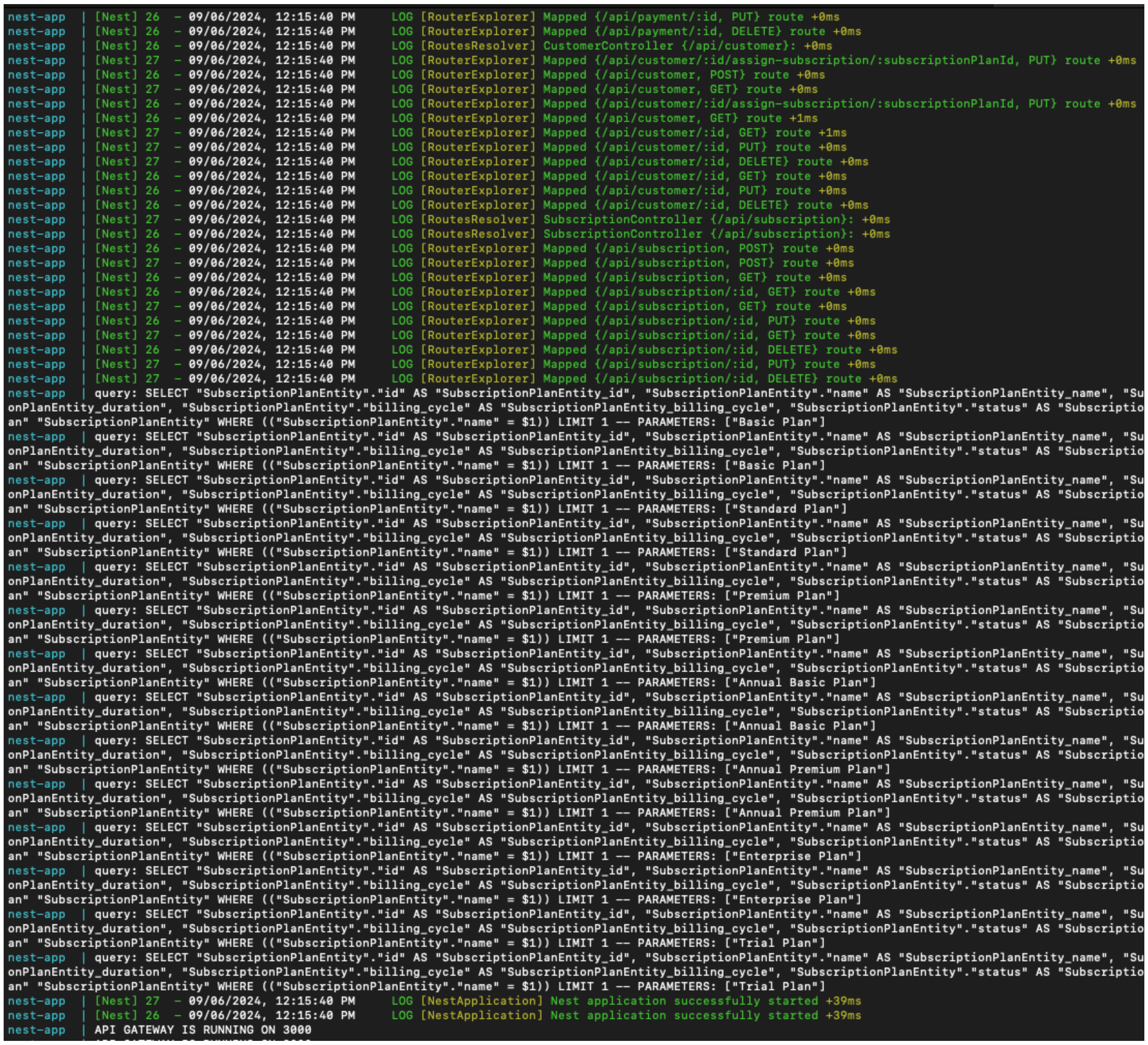
This docker-compose.pg.yml file will set up a PostgreSQL container and automatically create the necessary databases. To automate this process, I've implemented a shell script.

```
docker-compose -f docker-compose.pg.yml up -d
```

Step # 6: Execute Docker Compose to Start the SaaS Billing Application

Use Docker Compose to launch your SaaS billing application, initializing all necessary services defined in the `docker-compose.yml` file.

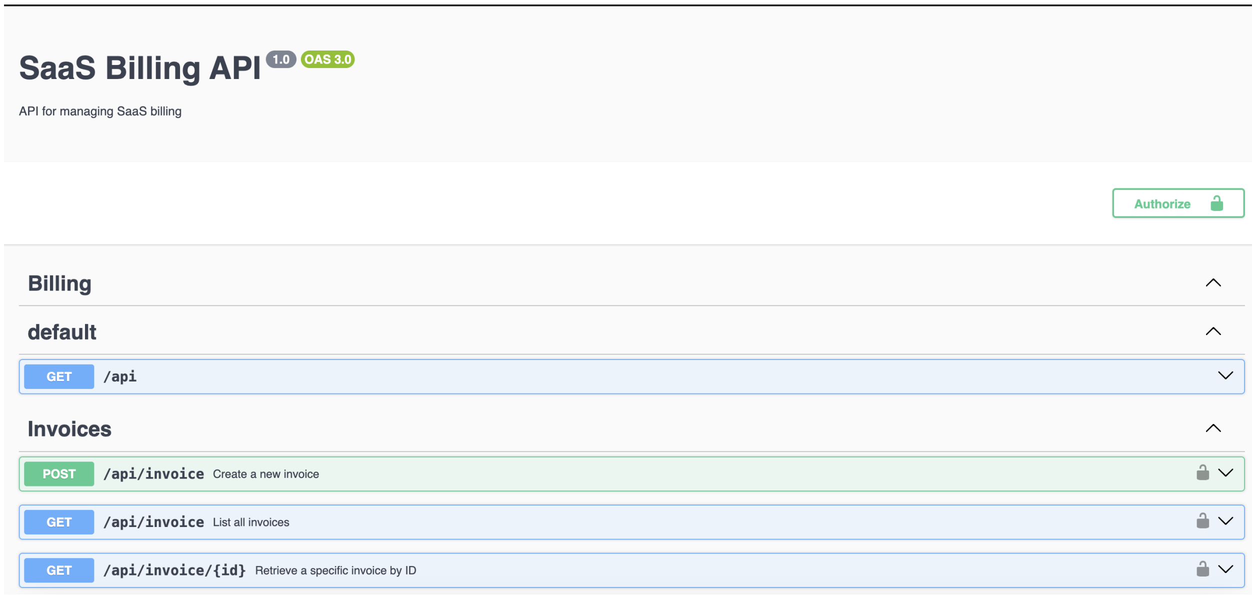
```
docker-compose  up -d && docker-compose logs -f
```



```
nest-app  | [Nest] 26  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/payment/:id, PUT} route +0ms
nest-app  | [Nest] 26  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/payment/:id, DELETE} route +0ms
nest-app  | [Nest] 26  - 09/06/2024, 12:15:40 PM     LOG [RoutesResolver] CustomerController {/api/customer}: +0ms
nest-app  | [Nest] 27  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/customer/:id/assign-subscription/:subscriptionPlanId, PUT} route +0ms
nest-app  | [Nest] 26  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/customer, POST} route +0ms
nest-app  | [Nest] 27  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/customer, GET} route +0ms
nest-app  | [Nest] 26  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/customer/:id/assign-subscription/:subscriptionPlanId, PUT} route +0ms
nest-app  | [Nest] 26  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/customer, GET} route +1ms
nest-app  | [Nest] 27  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/customer/:id, GET} route +1ms
nest-app  | [Nest] 27  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/customer/:id, PUT} route +0ms
nest-app  | [Nest] 27  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/customer/:id, DELETE} route +0ms
nest-app  | [Nest] 26  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/customer/:id, GET} route +0ms
nest-app  | [Nest] 26  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/customer/:id, PUT} route +0ms
nest-app  | [Nest] 26  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/customer/:id, DELETE} route +0ms
nest-app  | [Nest] 27  - 09/06/2024, 12:15:40 PM     LOG [RoutesResolver] SubscriptionController {/api/subscription}: +0ms
nest-app  | [Nest] 26  - 09/06/2024, 12:15:40 PM     LOG [RoutesResolver] SubscriptionController {/api/subscription}: +0ms
nest-app  | [Nest] 26  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/subscription, POST} route +0ms
nest-app  | [Nest] 27  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/subscription, POST} route +0ms
nest-app  | [Nest] 26  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/subscription, GET} route +0ms
nest-app  | [Nest] 26  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/subscription/:id, GET} route +0ms
nest-app  | [Nest] 27  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/subscription, GET} route +0ms
nest-app  | [Nest] 26  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/subscription/:id, PUT} route +0ms
nest-app  | [Nest] 27  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/subscription/:id, GET} route +0ms
nest-app  | [Nest] 26  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/subscription/:id, DELETE} route +0ms
nest-app  | [Nest] 27  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/subscription/:id, PUT} route +0ms
nest-app  | [Nest] 27  - 09/06/2024, 12:15:40 PM     LOG [RouterExplorer] Mapped {/api/subscription/:id, DELETE} route +0ms
nest-app  | query: SELECT "SubscriptionPlanEntity"."id" AS "SubscriptionPlanEntity_id", "SubscriptionPlanEntity"."name" AS "SubscriptionPlanEntity_name", "Su
onPlanEntity_duration", "SubscriptionPlanEntity"."billing_cycle" AS "SubscriptionPlanEntity_billing_cycle", "SubscriptionPlanEntity"."status" AS "Subscriptio
an" "SubscriptionPlanEntity" WHERE (("SubscriptionPlanEntity"."name" = $1)) LIMIT 1 -- PARAMETERS: ["Basic Plan"]
nest-app  | query: SELECT "SubscriptionPlanEntity"."id" AS "SubscriptionPlanEntity_id", "SubscriptionPlanEntity"."name" AS "SubscriptionPlanEntity_name", "Su
onPlanEntity_duration", "SubscriptionPlanEntity"."billing_cycle" AS "SubscriptionPlanEntity_billing_cycle", "SubscriptionPlanEntity"."status" AS "Subscriptio
an" "SubscriptionPlanEntity" WHERE (("SubscriptionPlanEntity"."name" = $1)) LIMIT 1 -- PARAMETERS: ["Basic Plan"]
nest-app  | query: SELECT "SubscriptionPlanEntity"."id" AS "SubscriptionPlanEntity_id", "SubscriptionPlanEntity"."name" AS "SubscriptionPlanEntity_name", "Su
onPlanEntity_duration", "SubscriptionPlanEntity"."billing_cycle" AS "SubscriptionPlanEntity_billing_cycle", "SubscriptionPlanEntity"."status" AS "Subscriptio
an" "SubscriptionPlanEntity" WHERE (("SubscriptionPlanEntity"."name" = $1)) LIMIT 1 -- PARAMETERS: ["Standard Plan"]
nest-app  | query: SELECT "SubscriptionPlanEntity"."id" AS "SubscriptionPlanEntity_id", "SubscriptionPlanEntity"."name" AS "SubscriptionPlanEntity_name", "Su
onPlanEntity_duration", "SubscriptionPlanEntity"."billing_cycle" AS "SubscriptionPlanEntity_billing_cycle", "SubscriptionPlanEntity"."status" AS "Subscriptio
an" "SubscriptionPlanEntity" WHERE (("SubscriptionPlanEntity"."name" = $1)) LIMIT 1 -- PARAMETERS: ["Standard Plan"]
nest-app  | query: SELECT "SubscriptionPlanEntity"."id" AS "SubscriptionPlanEntity_id", "SubscriptionPlanEntity"."name" AS "SubscriptionPlanEntity_name", "Su
onPlanEntity_duration", "SubscriptionPlanEntity"."billing_cycle" AS "SubscriptionPlanEntity_billing_cycle", "SubscriptionPlanEntity"."status" AS "Subscriptio
an" "SubscriptionPlanEntity" WHERE (("SubscriptionPlanEntity"."name" = $1)) LIMIT 1 -- PARAMETERS: ["Premium Plan"]
nest-app  | query: SELECT "SubscriptionPlanEntity"."id" AS "SubscriptionPlanEntity_id", "SubscriptionPlanEntity"."name" AS "SubscriptionPlanEntity_name", "Su
onPlanEntity_duration", "SubscriptionPlanEntity"."billing_cycle" AS "SubscriptionPlanEntity_billing_cycle", "SubscriptionPlanEntity"."status" AS "Subscriptio
an" "SubscriptionPlanEntity" WHERE (("SubscriptionPlanEntity"."name" = $1)) LIMIT 1 -- PARAMETERS: ["Premium Plan"]
nest-app  | query: SELECT "SubscriptionPlanEntity"."id" AS "SubscriptionPlanEntity_id", "SubscriptionPlanEntity"."name" AS "SubscriptionPlanEntity_name", "Su
onPlanEntity_duration", "SubscriptionPlanEntity"."billing_cycle" AS "SubscriptionPlanEntity_billing_cycle", "SubscriptionPlanEntity"."status" AS "Subscriptio
an" "SubscriptionPlanEntity" WHERE (("SubscriptionPlanEntity"."name" = $1)) LIMIT 1 -- PARAMETERS: ["Annual Basic Plan"]
nest-app  | query: SELECT "SubscriptionPlanEntity"."id" AS "SubscriptionPlanEntity_id", "SubscriptionPlanEntity"."name" AS "SubscriptionPlanEntity_name", "Su
onPlanEntity_duration", "SubscriptionPlanEntity"."billing_cycle" AS "SubscriptionPlanEntity_billing_cycle", "SubscriptionPlanEntity"."status" AS "Subscriptio
an" "SubscriptionPlanEntity" WHERE (("SubscriptionPlanEntity"."name" = $1)) LIMIT 1 -- PARAMETERS: ["Annual Basic Plan"]
nest-app  | query: SELECT "SubscriptionPlanEntity"."id" AS "SubscriptionPlanEntity_id", "SubscriptionPlanEntity"."name" AS "SubscriptionPlanEntity_name", "Su
onPlanEntity_duration", "SubscriptionPlanEntity"."billing_cycle" AS "SubscriptionPlanEntity_billing_cycle", "SubscriptionPlanEntity"."status" AS "Subscriptio
an" "SubscriptionPlanEntity" WHERE (("SubscriptionPlanEntity"."name" = $1)) LIMIT 1 -- PARAMETERS: ["Annual Premium Plan"]
nest-app  | query: SELECT "SubscriptionPlanEntity"."id" AS "SubscriptionPlanEntity_id", "SubscriptionPlanEntity"."name" AS "SubscriptionPlanEntity_name", "Su
onPlanEntity_duration", "SubscriptionPlanEntity"."billing_cycle" AS "SubscriptionPlanEntity_billing_cycle", "SubscriptionPlanEntity"."status" AS "Subscriptio
an" "SubscriptionPlanEntity" WHERE (("SubscriptionPlanEntity"."name" = $1)) LIMIT 1 -- PARAMETERS: ["Annual Premium Plan"]
nest-app  | query: SELECT "SubscriptionPlanEntity"."id" AS "SubscriptionPlanEntity_id", "SubscriptionPlanEntity"."name" AS "SubscriptionPlanEntity_name", "Su
onPlanEntity_duration", "SubscriptionPlanEntity"."billing_cycle" AS "SubscriptionPlanEntity_billing_cycle", "SubscriptionPlanEntity"."status" AS "Subscriptio
an" "SubscriptionPlanEntity" WHERE (("SubscriptionPlanEntity"."name" = $1)) LIMIT 1 -- PARAMETERS: ["Enterprise Plan"]
nest-app  | query: SELECT "SubscriptionPlanEntity"."id" AS "SubscriptionPlanEntity_id", "SubscriptionPlanEntity"."name" AS "SubscriptionPlanEntity_name", "Su
onPlanEntity_duration", "SubscriptionPlanEntity"."billing_cycle" AS "SubscriptionPlanEntity_billing_cycle", "SubscriptionPlanEntity"."status" AS "Subscriptio
an" "SubscriptionPlanEntity" WHERE (("SubscriptionPlanEntity"."name" = $1)) LIMIT 1 -- PARAMETERS: ["Enterprise Plan"]
nest-app  | query: SELECT "SubscriptionPlanEntity"."id" AS "SubscriptionPlanEntity_id", "SubscriptionPlanEntity"."name" AS "SubscriptionPlanEntity_name", "Su
onPlanEntity_duration", "SubscriptionPlanEntity"."billing_cycle" AS "SubscriptionPlanEntity_billing_cycle", "SubscriptionPlanEntity"."status" AS "Subscriptio
an" "SubscriptionPlanEntity" WHERE (("SubscriptionPlanEntity"."name" = $1)) LIMIT 1 -- PARAMETERS: ["Trial Plan"]
nest-app  | query: SELECT "SubscriptionPlanEntity"."id" AS "SubscriptionPlanEntity_id", "SubscriptionPlanEntity"."name" AS "SubscriptionPlanEntity_name", "Su
onPlanEntity_duration", "SubscriptionPlanEntity"."billing_cycle" AS "SubscriptionPlanEntity_billing_cycle", "SubscriptionPlanEntity"."status" AS "Subscriptio
an" "SubscriptionPlanEntity" WHERE (("SubscriptionPlanEntity"."name" = $1)) LIMIT 1 -- PARAMETERS: ["Trial Plan"]
nest-app  | [Nest] 27  - 09/06/2024, 12:15:40 PM     LOG [NestApplication] Nest application successfully started +39ms
nest-app  | [Nest] 26  - 09/06/2024, 12:15:40 PM     LOG [NestApplication] Nest application successfully started +39ms
nest-app  | API GATEWAY IS RUNNING ON 3000
```

Step # 7: Access API Documentation:

Visit `http://localhost:3000/api_docs` to view the API documentation for the SaaS billing application.


SaaS Billing API 1.0 OAS 3.0

API for managing SaaS billing

Authorize

Billing

default

GET /api

Invoices

POST /api/invoice Create a new invoice

GET /api/invoice List all invoices

GET /api/invoice/{id} Retrieve a specific invoice by ID

Test Implementation of Controller, Service, and E2E Testing

Implemented Test Cases for Every Controller and Service of Each Module:

In this step, we have successfully implemented comprehensive test cases for all controllers and services in each module of the application. The purpose of these tests is to ensure that the business logic of the application functions as expected, that proper data flows between components, and that edge cases and potential errors are handled correctly.

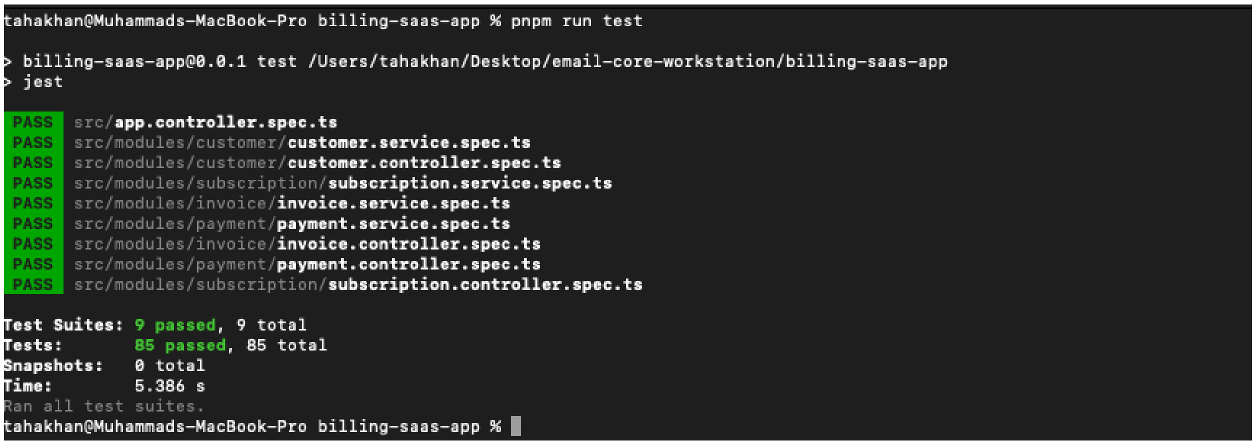
1. **Controller Test Cases:**
    - **Objective:** Verify that the controllers are correctly handling HTTP requests, routing them to the correct services, and returning appropriate responses.
    - **Scope:**
        - CRUD operations for entities like `Customer`, `Invoice`, `Payment`, `Subscription`, etc.

    - Status codes validation (`201`, `404`, `409`, etc.).
    - Proper handling of path parameters, body data, and query parameters.
    - Error handling and exceptions (e.g., `NotFoundException`).

2. **Service Test Cases:**
    - **Objective:** Ensure that all business logic in services is functioning correctly, and each service method performs its task reliably.
    - **Scope:**
        - Core functionalities such as creating, updating, retrieving, and deleting data.
        - Special features like handling payment retries, prorated billing, subscription upgrades, and failed payment retries.
        - Event-based actions (e.g., sending notifications on invoice creation or payment success).
        - Data validation, relationships between entities, and transactional integrity.
3. **Methodology:**
    - Each service and controller method is tested to cover both happy-path and edge-case scenarios.
    - Mocks are used for external dependencies such as repositories, external APIs, and event emitters.
    - Ensure proper state changes are happening in the database (e.g., after a successful payment, the status of invoices and payments is updated correctly).
4. **Outcome:**
    - Test cases ensure that the application handles both valid and invalid inputs gracefully.
    - The structure of the tests improves confidence that future changes will not break existing functionality.

- Developers can now easily extend or modify functionality, as existing test coverage will provide feedback on the impacts of changes.

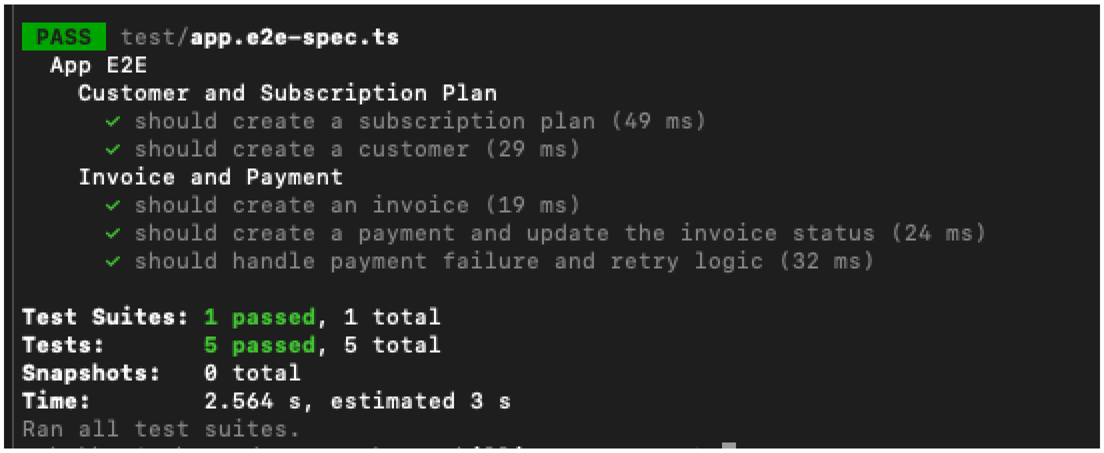
This step enhances the robustness of the application by ensuring that every module, controller, and service has been rigorously tested, reducing the risk of bugs in production and ensuring that core functionalities work as expected.



```
tahakhan@Muhammads-MacBook-Pro billing-saas-app % pnpm run test

> billing-saas-app@0.0.1 test /Users/tahakhan/Desktop/email-core-workstation/billing-saas-app
> jest

 PASS  src/app.controller.spec.ts
 PASS  src/modules/customer/customer.service.spec.ts
 PASS  src/modules/customer/customer.controller.spec.ts
 PASS  src/modules/subscription/subscription.service.spec.ts
 PASS  src/modules/invoice/invoice.service.spec.ts
 PASS  src/modules/payment/payment.service.spec.ts
 PASS  src/modules/invoice/invoice.controller.spec.ts
 PASS  src/modules/payment/payment.controller.spec.ts
 PASS  src/modules/subscription/subscription.controller.spec.ts

Test Suites: 9 passed, 9 total
Tests:       85 passed, 85 total
Snapshots:   0 total
Time:        5.386 s
Ran all test suites.
tahakhan@Muhammads-MacBook-Pro billing-saas-app %
```

Implemented APP E2E Testing:

End-to-end (E2E) testing has been implemented to verify that the entire application works as expected from start to finish, ensuring all components interact correctly and the user experience remains smooth.



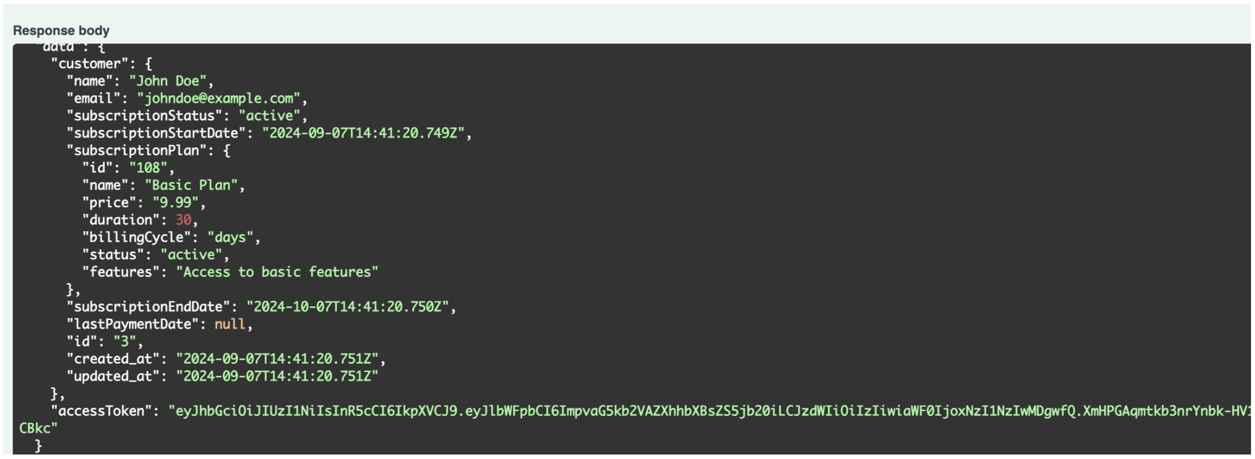
```
 PASS  test/app.e2e-spec.ts
  App E2E
    Customer and Subscription Plan
      ✓ should create a subscription plan (49 ms)
      ✓ should create a customer (29 ms)
    Invoice and Payment
      ✓ should create an invoice (19 ms)
      ✓ should create a payment and update the invoice status (24 ms)
      ✓ should handle payment failure and retry logic (32 ms)

Test Suites: 1 passed, 1 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        2.564 s, estimated 3 s
Ran all test suites.
```

Exploring Core Features and Use Cases

This section delves into the primary functionalities and practical applications of the system.

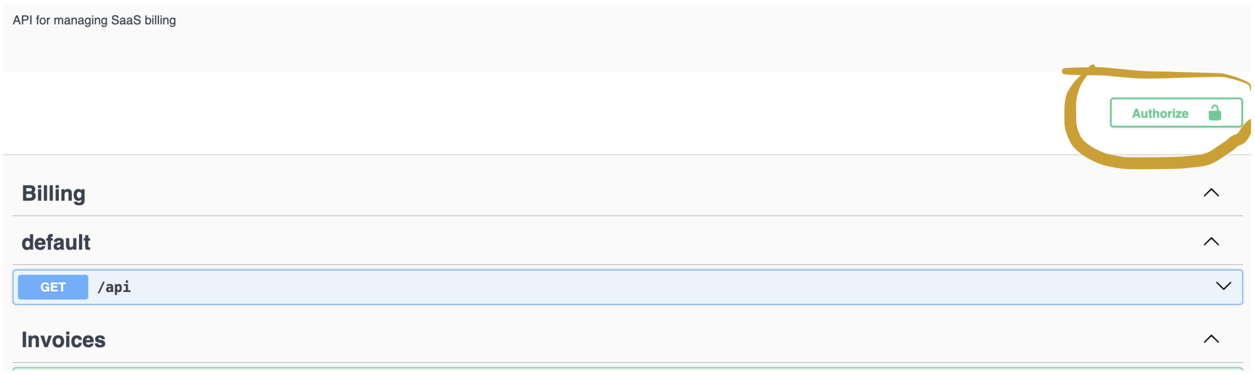
Prerequisites for Using APIs

1. I have implemented JWT authentication, so a JWT token is required to access all routes except the customer route.
2. Obtain the access token by creating a new customer at the following URL: `http://localhost:3000/api/customer`


Response body

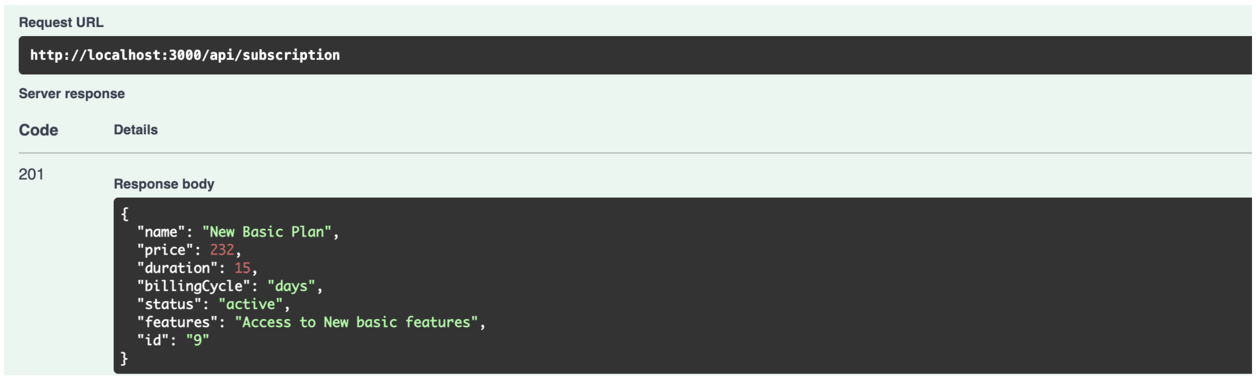
```
  "data": {
    "customer": {
      "name": "John Doe",
      "email": "johndoe@example.com",
      "subscriptionStatus": "active",
      "subscriptionStartDate": "2024-09-07T14:41:20.749Z",
      "subscriptionPlan": {
        "id": "108",
        "name": "Basic Plan",
        "price": "9.99",
        "duration": 30,
        "billingCycle": "days",
        "status": "active",
        "features": "Access to basic features"
      },
      "subscriptionEndDate": "2024-10-07T14:41:20.750Z",
      "lastPaymentDate": null,
      "id": "3",
      "created_at": "2024-09-07T14:41:20.751Z",
      "updated_at": "2024-09-07T14:41:20.751Z"
    },
    "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImpvaG5kb2VAZXhhbXBsZS5jb20iLCJzdWIiOiIzIiwiaWF0IjoxNzI1NzIwMDgwfQ.XmHPGAqmtkb3nrYnbk-HV
CBkc"
  }
```

3. Click the "Authorize" button, paste the JWT token, and click "Authorize" again. This will automatically include the JWT token in your requests, so you won't need to add it manually when accessing any routes.



Subscription Management:

- **Create and manage subscription plans with different pricing and billing cycles.**



Request URL

```
http://localhost:3000/api/subscription
```

Server response

Code Details

201

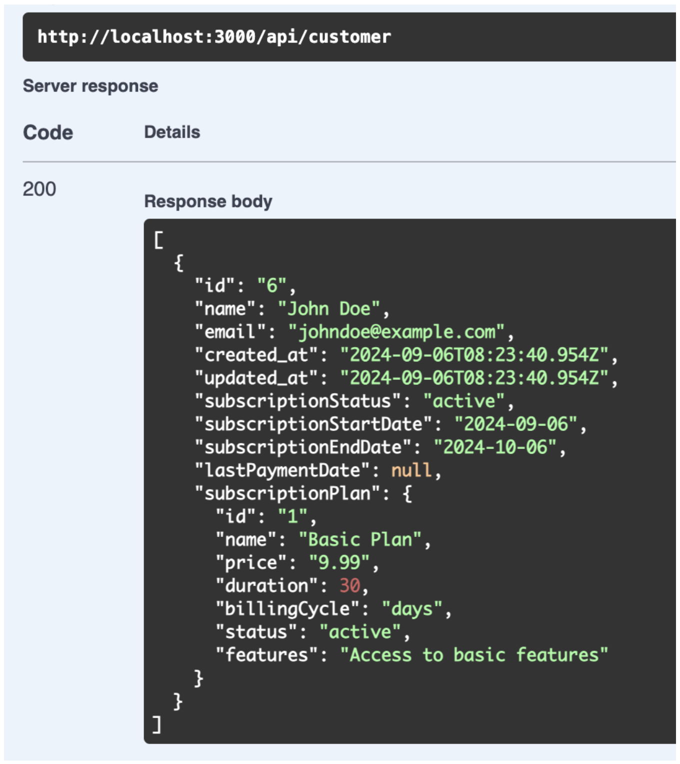
Response body

```
{
  "name": "New Basic Plan",
  "price": 232,
  "duration": 15,
  "billingCycle": "days",
  "status": "active",
  "features": "Access to New basic features",
  "id": "9"
}
```

- **Assign subscription plans to customers and manage their subscription status**

  Steps:

1. Retrieve a list of all customers.
2. Choose a customer ID that you wish to manage.
3. Assign a subscription plan to the selected customer using this API endpoint: `/api/customer/{id}/assign-subscription/{subscriptionPlanId}` .



```
http://localhost:3000/api/customer
```

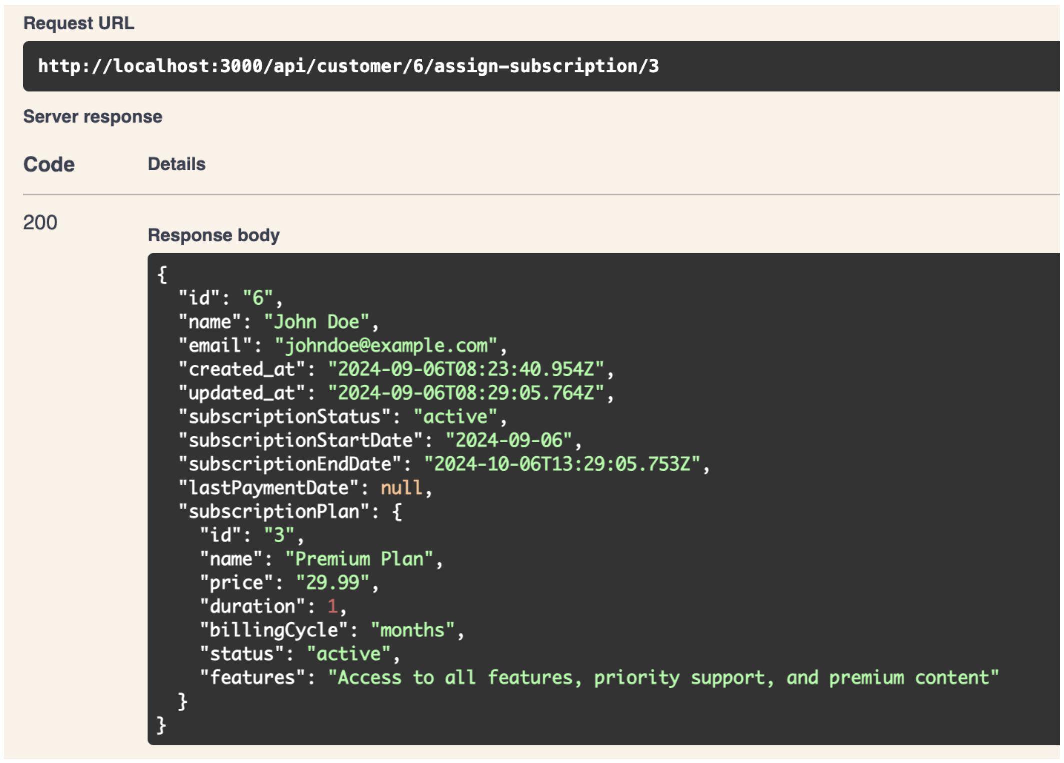
Server response

Code Details

200

Response body

```
[
  {
    "id": "6",
    "name": "John Doe",
    "email": "johndoe@example.com",
    "created_at": "2024-09-06T08:23:40.954Z",
    "updated_at": "2024-09-06T08:23:40.954Z",
    "subscriptionStatus": "active",
    "subscriptionStartDate": "2024-09-06",
    "subscriptionEndDate": "2024-10-06",
    "lastPaymentDate": null,
    "subscriptionPlan": {
      "id": "1",
      "name": "Basic Plan",
      "price": "9.99",
      "duration": 30,
      "billingCycle": "days",
      "status": "active",
      "features": "Access to basic features"
    }
  }
]
```



Request URL

```
http://localhost:3000/api/customer/6/assign-subscription/3
```

Server response

Code Details

200

Response body

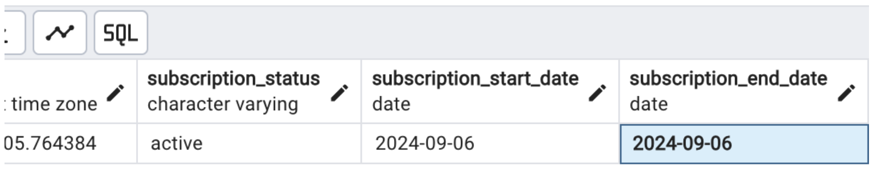
```
{
  "id": "6",
  "name": "John Doe",
  "email": "johndoe@example.com",
  "created_at": "2024-09-06T08:23:40.954Z",
  "updated_at": "2024-09-06T08:29:05.764Z",
  "subscriptionStatus": "active",
  "subscriptionStartDate": "2024-09-06",
  "subscriptionEndDate": "2024-10-06T13:29:05.753Z",
  "lastPaymentDate": null,
  "subscriptionPlan": {
    "id": "3",
    "name": "Premium Plan",
    "price": "29.99",
    "duration": 1,
    "billingCycle": "months",
    "status": "active",
    "features": "Access to all features, priority support, and premium content"
  }
}
```

Billing Engine:

- **Automatically generate invoices at the end of each billing cycle based on the customer’s subscription plan**

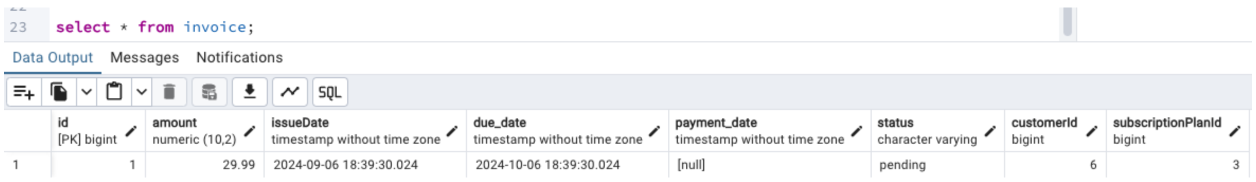
Steps :

1. Update the `subscription_end_date` to today's date to trigger generation at the end of each billing cycle.



time zone	subscription_status character varying	subscription_start_date date	subscription_end_date date
05.764384	active	2024-09-06	2024-09-06

2. Following the update, a cron job will execute at midnight to generate the customer's invoice.

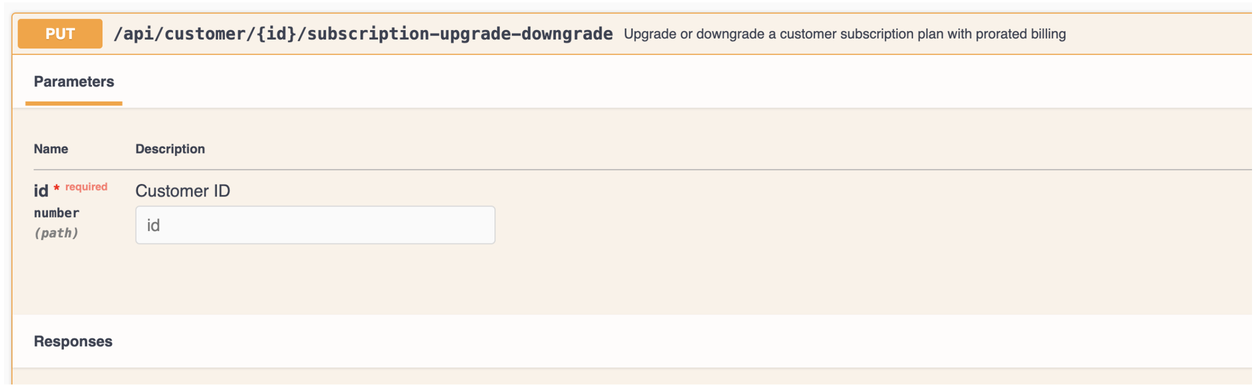


```sql
select * from invoice;
```

	id [PK] bigint	amount numeric (10,2)	issueDate timestamp without time zone	due_date timestamp without time zone	payment_date timestamp without time zone	status character varying	customerId bigint	subscriptionPlanId bigint
1	1	29.99	2024-09-06 18:39:30.024	2024-10-06 18:39:30.024	[null]	pending	6	3

- **Handle prorated billing for mid-cycle upgrades or downgrades.**

Utilize this API to adjust billing for mid-cycle subscription upgrades or downgrades.



PUT `/api/customer/{id}/subscription-upgrade-downgrade` Upgrade or downgrade a customer subscription plan with prorated billing

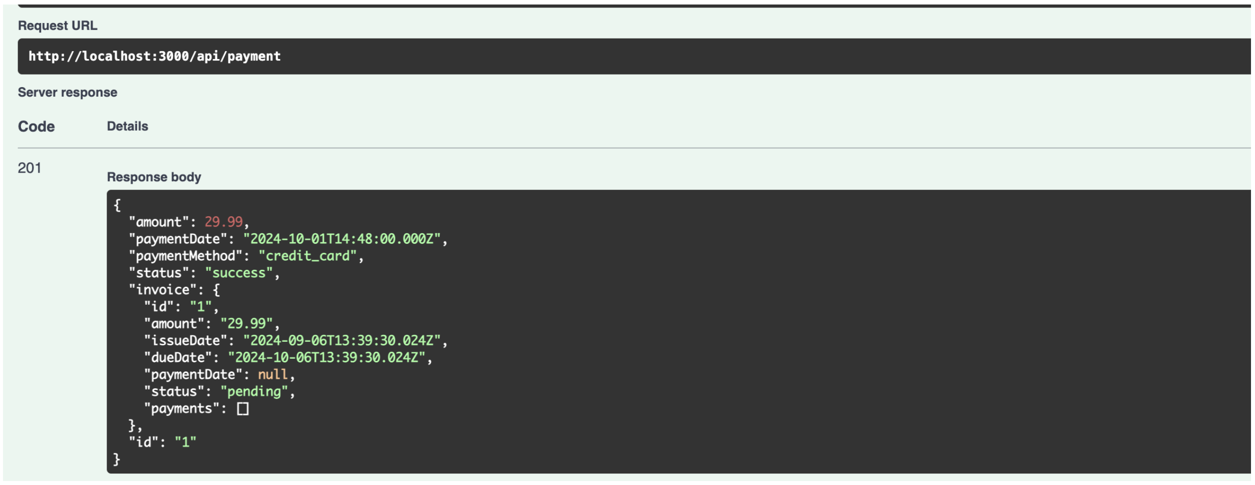
Parameters

Name	Description
id * required number (path)	Customer ID id

Responses

Payment Processing:

- Record payments made by customers and update invoice status accordingly



Request URL

```
http://localhost:3000/api/payment
```

Server response

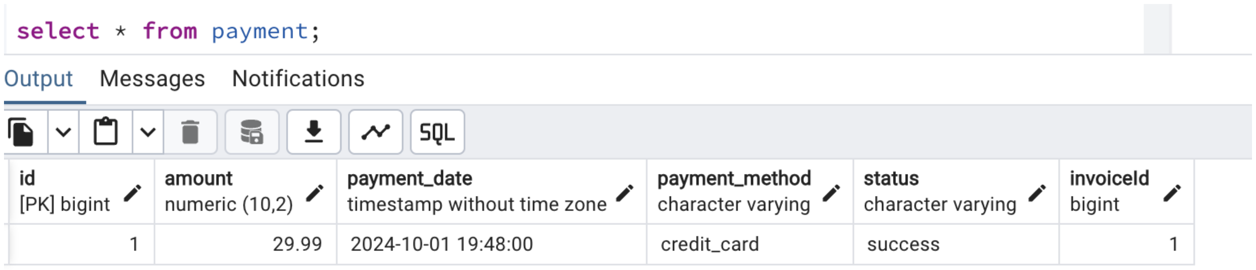
Code Details

201

Response body

```
{
  "amount": 29.99,
  "paymentDate": "2024-10-01T14:48:00.000Z",
  "paymentMethod": "credit_card",
  "status": "success",
  "invoice": {
    "id": "1",
    "amount": "29.99",
    "issueDate": "2024-09-06T13:39:30.024Z",
    "dueDate": "2024-10-06T13:39:30.024Z",
    "paymentDate": null,
    "status": "pending",
    "payments": []
  },
  "id": "1"
}
```

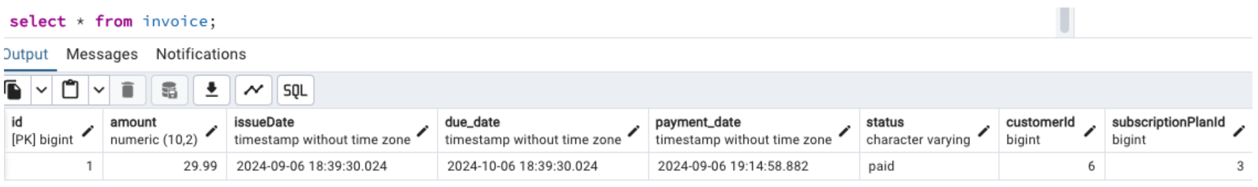
- As you can observe, I have made the payment for my pending invoice.
- A new payment will be recorded against this invoice.
- The invoice status will be updated to 'paid,' and the payment date will be recorded at the time of payment.
- The subscription end date will be extended from today to next month based on the plan selected.



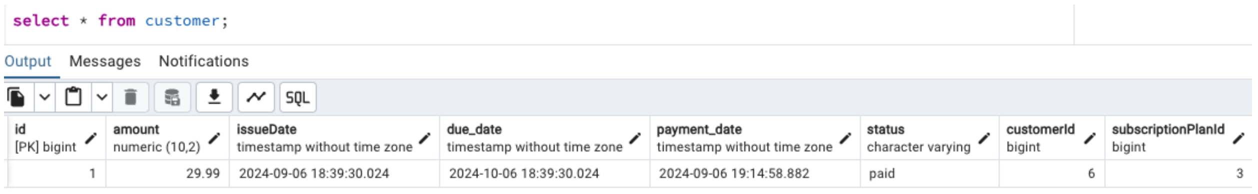
```
select * from payment;
```

Output Messages Notifications

id [PK] bigint	amount numeric (10,2)	payment_date timestamp without time zone	payment_method character varying	status character varying	invoiceId bigint
1	29.99	2024-10-01 19:48:00	credit_card	success	1


```
select * from invoice;
```
Output Messages Notifications

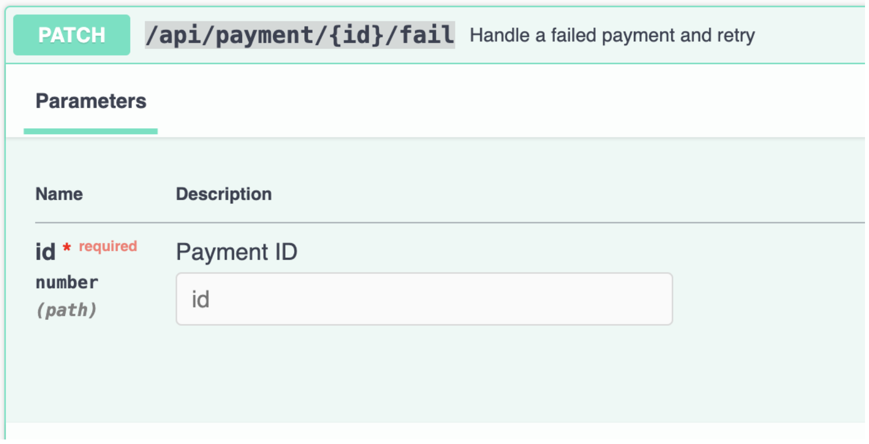
id [PK] bigint	amount numeric (10,2)	issueDate timestamp without time zone	due_date timestamp without time zone	payment_date timestamp without time zone	status character varying	customerId bigint	subscriptionPlanId bigint
1	29.99	2024-09-06 18:39:30.024	2024-10-06 18:39:30.024	2024-09-06 19:14:58.882	paid	6	3


```
select * from customer;
```
Output Messages Notifications

id [PK] bigint	amount numeric (10,2)	issueDate timestamp without time zone	due_date timestamp without time zone	payment_date timestamp without time zone	status character varying	customerId bigint	subscriptionPlanId bigint
1	29.99	2024-09-06 18:39:30.024	2024-10-06 18:39:30.024	2024-09-06 19:14:58.882	paid	6	3

- **Handle failed payments and implement retry logic.**

Utilize this API endpoint `/api/payment/{id}/fail` to manage failed payments and initiate retries.


PATCH /api/payment/{id}/fail Handle a failed payment and retry

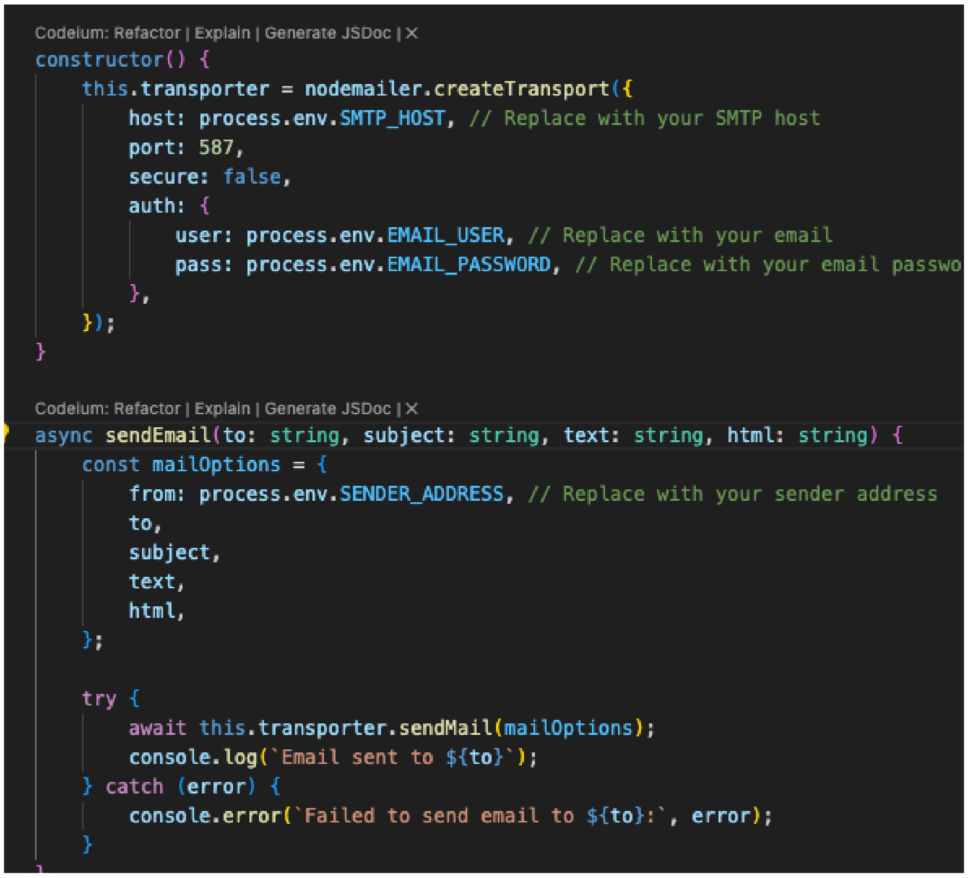
Parameters

Name	Description
id * required number *(path)*	Payment ID id

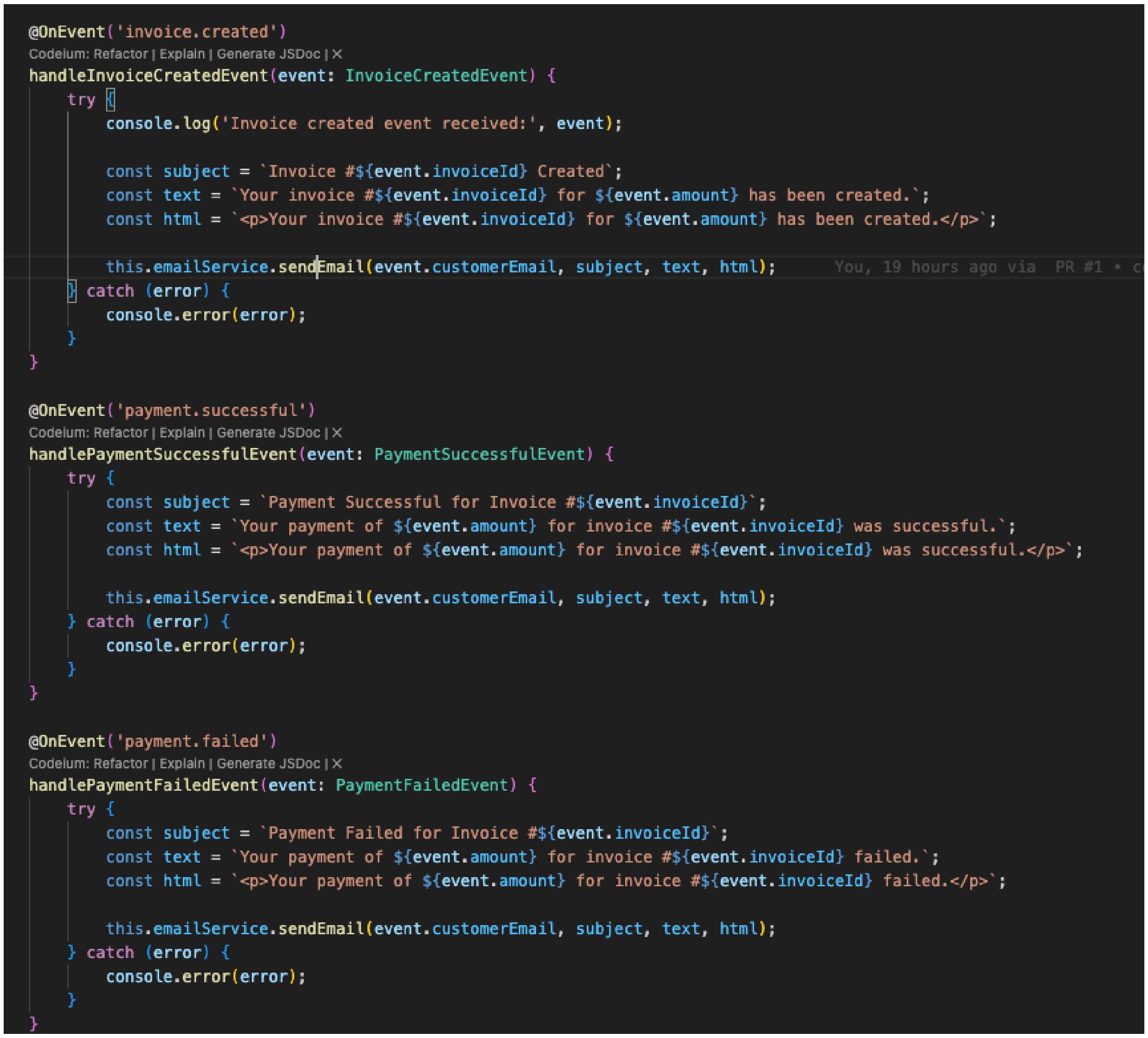
Notifications:

- **Send email notifications to customers when an invoice is generated, when a payment is successful, or when a payment fails**

I have implemented email notifications that are automatically sent to the customer's preferred address whenever a payment is processed or fails. Similarly, when an invoice is generated, an event triggers an email notification to inform the customer about the new invoice.



```
Codeium: Refactor | Explain | Generate JSDoc | ×
constructor() {
    this.transporter = nodemailer.createTransport({
        host: process.env.SMTP_HOST, // Replace with your SMTP host
        port: 587,
        secure: false,
        auth: {
            user: process.env.EMAIL_USER, // Replace with your email
            pass: process.env.EMAIL_PASSWORD, // Replace with your email passwo
        },
    });
}

Codeium: Refactor | Explain | Generate JSDoc | ×
async sendEmail(to: string, subject: string, text: string, html: string) {
    const mailOptions = {
        from: process.env.SENDER_ADDRESS, // Replace with your sender address
        to,
        subject,
        text,
        html,
    };

    try {
        await this.transporter.sendMail(mailOptions);
        console.log(`Email sent to ${to}`);
    } catch (error) {
        console.error(`Failed to send email to ${to}:`, error);
    }
```



```
@OnEvent('invoice.created')
Codeium: Refactor | Explain | Generate JSDoc | ×
handleInvoiceCreatedEvent(event: InvoiceCreatedEvent) {
    try {
        console.log('Invoice created event received:', event);

        const subject = `Invoice #${event.invoiceId} Created`;
        const text = `Your invoice #${event.invoiceId} for ${event.amount} has been created.`;
        const html = `<p>Your invoice #${event.invoiceId} for ${event.amount} has been created.</p>`;

        this.emailService.sendEmail(event.customerEmail, subject, text, html);      You, 19 hours ago via  PR #1 • c
    } catch (error) {
        console.error(error);
    }
}

@OnEvent('payment.successful')
Codeium: Refactor | Explain | Generate JSDoc | ×
handlePaymentSuccessfulEvent(event: PaymentSuccessfulEvent) {
    try {
        const subject = `Payment Successful for Invoice #${event.invoiceId}`;
        const text = `Your payment of ${event.amount} for invoice #${event.invoiceId} was successful.`;
        const html = `<p>Your payment of ${event.amount} for invoice #${event.invoiceId} was successful.</p>`;

        this.emailService.sendEmail(event.customerEmail, subject, text, html);
    } catch (error) {
        console.error(error);
    }
}

@OnEvent('payment.failed')
Codeium: Refactor | Explain | Generate JSDoc | ×
handlePaymentFailedEvent(event: PaymentFailedEvent) {
    try {
        const subject = `Payment Failed for Invoice #${event.invoiceId}`;
        const text = `Your payment of ${event.amount} for invoice #${event.invoiceId} failed.`;
        const html = `<p>Your payment of ${event.amount} for invoice #${event.invoiceId} failed.</p>`;

        this.emailService.sendEmail(event.customerEmail, subject, text, html);
    } catch (error) {
        console.error(error);
    }
}
```

InvoiceGenerationFunction(OptionalButPlus):

The deadline you gave me was reasonable, but I didn't get a chance to work on it for five days because I had office work and other deadlines to meet. I only had time to work on it over the weekend, which is why I couldn't implement the AWS serverless functionality. If I had two more days, I would definitely complete it.

The following design patterns are used in this application:

1. Repository Pattern

- **Where it's used**: The repository pattern is used in the service layer to encapsulate the logic for interacting with the database. In app, repositories like `CustomerRepository`, `InvoiceRepository`, and `SubscriptionPlanRepository` are used to abstract data persistence from the business logic.
- **How it works**: Instead of writing database queries in the service layer, the repository provides an interface to perform CRUD operations. This pattern improves separation of concerns by isolating the database layer from the application logic.

Example:

```
const customer = await this.customerRepository.findOne({ where: { id: customerId } });
```

2. Service Layer Pattern

- **Where it's used**: The **service layer** pattern is seen in your `CustomerService`, `InvoiceService`, `SubscriptionService`, and `PaymentService`. It provides a level of abstraction over business logic, keeping controllers thin and focused on handling HTTP requests.
- **How it works**: The service layer contains business rules and logic. For instance, in `upgradeOrDowngradeSubscription()`, you calculate prorated costs, create invoices, and update subscription details. The service interacts with repositories and coordinates between them.
- **Example**:

  ```
  const updatedCustomer = await this.customerService.update(customerId, updateCustomerDto);
  ```

3. Dependency Injection (DI) Pattern

- **Where it's used**: This pattern is inherent to NestJS and used throughout the application. Each class that depends on another class (e.g., services, repositories) is injected via the constructor. This decouples components from each other and makes the system more modular and testable.
- **How it works**: Services and repositories are passed into constructors, and NestJS handles their lifecycle. For example, the `CustomerService` depends on the `CustomerRepository` and `SubscriptionPlanRepository`, but it does not need to instantiate these dependencies.

Example:

```
constructor(
  @InjectRepository(CustomerEntity) private readonly customer
Repository: Repository<CustomerEntity>,
) {}
```

4. Event-Driven Pattern (Asynchronous Messaging)

- **Where it's used**: I use event-driven notifications, which would apply the event-driven pattern. For example, when an invoice is created, an event like `InvoiceCreatedEvent` could be emitted, and the notification service listens for that event to send an email notification.
- **How it works**: Events are published when something significant happens (e.g., invoice generation or failed payment). Other services (e.g., notification) subscribe to these events and handle them asynchronously.

Example:

```
this.eventEmitter.emit('invoice.created', { invoiceId: newInv
oice.id });
```

5. Singleton Pattern:

- **Where it's used**: I use the Singleton pattern to ensure that services like database connections, JWT service, and event emitter are instantiated only once throughout the application. This minimizes resource usage and provides a global access point for these services.

- **How it works**: The Singleton pattern restricts the instantiation of a class to one object. This is achieved by making the constructor private and providing a static method that ensures only one instance is created and reused.

Summary of Patterns I am Using:

1. **Repository Pattern**: Used for data access and separation of business logic from the persistence layer.
2. **Service Layer Pattern**: Handles the core business logic and orchestrates between the controller and repository.
3. **Dependency Injection**: Used throughout the application, as it's inherent in NestJS.
4. **Event-Driven/Observer Pattern**: Used for sending notifications or triggering actions asynchronously after certain events (like invoice generation or payment success).

Each of these design patterns is being used to make application more modular, scalable, and maintainable.

Deliverables:

1. I have provided you with a GitHub repository link to clone the project.

```
https://github.com/tahakhan-dev/billing-saas-app.git
```

2. This is a brief documentation to help you set up and configure the project. I've provided all the necessary steps for the setup.
3. You can access the basic API documentation, which is built using Swagger. http://localhost:3000/api_docs

Task 2: Code Refactoring

Problematic Code:

JavaScript:

```
 app.get('/product/:productId', (req, res) => {
       db.query(`SELECT * FROM products WHERE id=${req.param
s.productId}`, (err,
   result) => {
           if (err) throw err;
           res.send(result);
       });
});
```

1. **SQL Injection Vulnerability:**

   - **Problem:** The original code constructs an SQL query by directly embedding the `productId` from the request URL into the SQL statement. This practice is dangerous because it makes the application vulnerable to SQL injection attacks, where an attacker can manipulate the SQL query by crafting a malicious `productId`
   - **Impact:** SQL injection can lead to unauthorized access to or manipulation of the database. An attacker might execute arbitrary SQL commands, potentially causing data breaches, data corruption, or even complete compromise of the database.
   - **Solution:** Use parameterized queries (also known as prepared statements) to safely pass user input to the SQL query. Instead of directly embedding the `productId` into the query string, you should use a placeholder (`?` in MySQL, `$1` in PostgreSQL, etc.) and pass the user input as a separate parameter.
   - **Implementation:**

```
const query = 'SELECT * FROM products WHERE id = ?';
db.query(query, [req.params.productId], (err, result) => {
    // Handle the result or error
});
```

2. **Poor Error Handling:**

- **Problem:** The original code uses `throw err` within the callback function of the database query. While this will stop the code execution if an error occurs, it does not handle the error in a user-friendly manner. It might crash the entire server if not properly caught and handled elsewhere in the application.
- **Impact:** If an error occurs, the server might crash, leading to downtime and a poor user experience. Additionally, the lack of meaningful error messages makes it difficult for users to understand what went wrong and how they might rectify the issue.
- **Solution:** Instead of throwing the error with `throw err`, you should handle it gracefully by logging the error and returning an appropriate response to the client. This can be done using `console.error()` for logging and `res.status()` for setting the HTTP status code.
- **Implementation:**

```
db.query(query, [req.params.productId], (err, result) => {
    if (err) {
        console.error('Database query error:', err);
        return res.status(500).json({ message: 'Internal Serv
    }
    // Handle the result
});
```

3. **Missing Proper HTTP Status Codes:**
    - **Problem:** The original code does not return appropriate HTTP status codes based on the outcome of the operation. For example, it does not handle scenarios where the requested product is not found in the database.
    - **Impact:** Not returning the correct HTTP status codes can lead to confusion for the client applications consuming this API. It also violates RESTful API principles, where responses should include status codes that indicate the success or failure of a request.

- **Solution:** Use `res.status()` to return the correct HTTP status codes depending on the result of the operation. For example, return `404 Not Found` if the requested resource does not exist, and return `200 OK` for successful operations.
- **Implementation:**

```
if (result.length === 0) {
    return res.status(404).json({ message: 'Product not found
}
res.status(200).json(result);
```

Refactored Code:

```
app.get('/product/:productId', (req, res) => {
    const productId = req.params.productId;

    // Use a parameterized query to prevent SQL injection
    const query = 'SELECT * FROM products WHERE id = ?';
    db.query(query, [productId], (err, result) => {
        if (err) {
         // Log the error for debugging purposes
            console.error(err);

         // Send a 500 Internal Server Error response with a
generic message
            return res.status(500).send('Internal Server Erro
r');
        }
        if (result.length === 0) {
            return res.status(404).send('Product not found');
        }
        res.status(200).send(result);
    });
});
```