Order & Returns Management System

Build a robust backend application that handles a complex business workflow with intricate state management, background job processing, and third-party integrations. The focus is on writing high-quality, maintainable code that correctly implements complex business rules.

1. Business Scenario

"ArtiCurated," a boutique online marketplace, sells high-value artisanal goods. Their business is growing, and they need a reliable system to manage the entire lifecycle of an order, from payment to potential return and refund. Because their products are unique, their returns process has several manual approval steps and requires clear tracking.

2. Core Functional Requirements

You are tasked with building the backend for their order management system. It should be a single, well-structured application

2.1. Complex Order State Management

An `Order` must progress through a defined lifecycle. You must implement a state machine to manage this, enforcing the correct transitions. The states are:

- `PENDING_PAYMENT` -> `PAID` -> `PROCESSING_IN_WAREHOUSE` -> `SHIPPED` -> `DELIVERED`
- An order can also be `CANCELLED` from `PENDING_PAYMENT` or `PAID` (if not yet processed).

2.2. Multi-Step Returns Workflow

A `Return` request also has its own lifecycle and must be implemented as a state machine.

- A return can only be initiated for an order that is in the `DELIVERED` state.
- The states for a `Return` are:
  - `REQUESTED`: The customer initiates the return.
  - `APPROVED` / `REJECTED`: A store manager reviews the request and approves or rejects it.
  - `IN_TRANSIT`: The customer has shipped the item back.
  - `RECEIVED`: The warehouse confirms receipt of the returned item.
  - `COMPLETED`: The refund is successfully processed.

- The system must log the history of all state changes for both orders and returns for auditing purposes.

2.3. Asynchronous Background Jobs
Certain actions should not block the user or the main application thread. These must be implemented using a background job processing library (e.g., Hangfire, Celery, Sidekiq, or a simple custom solution) or decoupled through Message queues (e.g. RabbitMQ, Apache ActiveMQ, AWS SQS, etc...)

- **PDF Invoice Generation:** When an order transitions to `SHIPPED`, a background job should be queued to generate a PDF invoice (you can use a library to create a simple dummy PDF) and simulate emailing it.
- **Refund Processing:** When a return's status becomes `COMPLETED`, a background job must be queued to make an API call to a mock payment gateway service to process the refund.

3. Deliverables

Create a private GitHub repo and share with the below members:

- dmistryTal
- nileshmallick1606
- Sachin-Salunke-Talentica

Kindly note the names of each of the expected files should be the same. The automated evaluation mechanism expects that those file names are accurate, if not then it will impact the final score.

Your submission will be a single private GitHub repository containing the following:

1. **Source Code:** The complete, running source code for the application.

2. `README.md`: A clear overview of the project and detailed instructions on how to set up the database and run the application and its background workers.

3. `PROJECT_STRUCTURE.md`: Explaining the structure of the project and the purpose for each of the folder and key modules.

4. `WORKFLOW_DESIGN.md`**:** A document explaining your implementation
    - **State Machine Diagrams**: Include simple diagrams (text-based is fine) for both the Order and Return workflow`s.

- **Database Schema**: A diagram or description of your database tables, explaining the relationships and how you store the state history.

5. `API-SPECIFICATION.yml`**:** A simple document or Postman collection defining the API endpoints you built.
    - The file name should be `POSTMAN_COLLECTION.json in` case of a postman collection.
    - The file name should be `API-SPECIFICATION.md` if it is a markdown file.
    - The file name should be `API-SPECIFICATION.yml` if it is an API sepecification file.

6. `docker-compose.yml`: A single, working Docker Compose file that starts all required components of your system for easy validation.

7. `CHAT_HISTORY.md`: A summary document that chronicles your design journey with your AI assistant, highlighting key decision points and how you used AI to evaluate alternatives.

8. **Unit Tests & Coverage Report.**

9. **Video**: An 8-10 min video explaining:
    - Design, architecture and the different components and how they communicate with each other.
    - Explain the journey from initial brainstorming till the final implementation and the conversation with the coding assistant.
    - Key decisions and trade-offs.
    - Demo of the entire working of the application.
    - Test case coverage %
    - **Upload the video on your one drive and share the access with the below members:**
        1. Dipen.mistry@talentica.com
        2. Nilesh.Mallick@talentica.com
        3. Sachin.Salunke@talentica.com