Technical Test (Senior Backend Engineer)

Objective:

The goal of this test is to evaluate your experience and skill in building robust backend services using TypeScript and Node.js, applying Domain-Driven Design (DDD) and Hexagonal architecture principles. You'll be tasked with creating a microservice responsible for handling logistics with different shipping providers.

Context

You are required to build a backend microservice to manage logistics operations. This service will interact with multiple mocked shipping providers (NRW and TLS). Each provider exposes its APIs with different data formats and mechanisms for status updates.

Your microservice must:

- Handle delivery creation requests for a given order and generate a printable shipping label returned in the response.
- Allow querying the real-time status of any delivery.

The service must use a **non-relational database** (e.g., MongoDB, DynamoDB, or another document store) for persistence.

Requirements

1. Architecture & Code Design

- Apply **Hexagonal Architecture** (Ports & Adapters).
- Apply **Domain-Driven Design** principles.
- Implement a **microservice** in TypeScript using Node.js (choose the framework you prefer).
- Each provider (NRW, TLS) must be implemented behind an **abstraction layer**. Your service should integrate with all of them through a common interface, despite their differences.

2. Functional Scope

Implement the following endpoints:

- POST **/deliveries** -> Accepts an order payload and:
    - Selects a provider (randomly or based on simple logic),
    - Requests label generation from the selected provider,
    - Returns the label to the user.

- GET **/deliveries/:id/status** -> Returns the latest known delivery status from your database. Status updates must be processed asynchronously based on the provider's mechanism:
    - **Polling-based providers** (e.g., NRW): Fetch updates from the provider's API every hour.
    - **Webhook-based providers** (e.g., TLS): Handle incoming webhook calls from the provider to update delivery status.
    - You must simulate both mechanisms in your solution. For webhooks, implement a route that accepts provider callbacks. For polling, implement a scheduled background task.

3. Persistence

- Use a **non-relational database** to persist delivery data and status.
- Clearly structure your data models using appropriate repository patterns.

4. Testing

- Include **unit tests** for core domain logic. (100% coverage not needed)

- **[Optional]** Include **integration tests** for your API endpoints and provider interfaces.

5. [Optional] Documentation

- **Integration of auto-generated API documentation using OpenAPI (Swagger or similar)**.
- An overview of the architecture (ideally with a diagram).
- Example payloads and curl/Postman commands to test endpoints.

Deliverables

A GitHub repository containing:

- The project code.
- A README.md that includes:
    - Setup instructions to run the service locally (using Docker or similar).
    - How to trigger and test the asynchronous status update flows (polling and webhooks).

Evaluation Criteria

- Clear application of DDD and Hexagonal Architecture.
- Clean, modular, and testable code.
- Quality of abstraction across providers.
- Correct simulation of both polling and webhook status update mechanisms.
- Appropriate use of non-relational persistence.
- Proper documentation and setup instructions.
- Functional delivery creation and real-time status tracking.
- Usage of AI tools (e.g., Copilot, ChatGPT, Cursor…) is allowed and even encouraged, as long as you can confidently explain and defend any generated code or architecture decisions.

Estimated Time

This test is designed to be completed within approximately 6 to 10 hours. You are allowed one full week from receipt to complete the test at your convenience.