Coursework Reflective Report

CCS3411 - SOA & Microservices

Task 01: SOA Design Principles

To decompose the monolithic architecture of GlobalBooks Inc., several fundamental principles of Service-Oriented Architecture (SOA) were applied to build a more flexible, maintainable, and fault-tolerant system.

Key Principles Applied:

- **Service Autonomy:** Each service (Catalog, Orders, Payments, Shipping) was designed to operate independently. This means each service manages its own business logic and data. For example, if a change is required in the *CatalogService*, it can be redeployed in isolation without impacting any other service, such as the *OrdersService*.

- **Loose Coupling:** The services were designed with minimal dependencies on each other. Instead of tightly coupled synchronous calls, we utilized an asynchronous messaging system, **RabbitMQ (via CloudAMQP)**, for communication between the PaymentsService and ShippingService. In this pattern, the PaymentsService does not call the ShippingService directly but rather publishes a message to a queue, which the ShippingService subscribes to.

- **Standardized Contract:** Each service exposes its capabilities through a formal, standardized contract. For the CatalogService, this contract is a **WSDL** (Web Services Definition Language) file, which clearly defines the operations, data types, and message formats of the SOAP interface. For the OrdersService, the contract is a **REST API** with a well-defined JSON schema.

Task 02: Benefit and Challenge

- **Primary Benefit (Agility):** The single greatest advantage of this SOA approach is business agility. In the old monolithic system, fixing a minor bug could take weeks, as it required the entire application to be re-tested and redeployed. With the new architecture, a change in the PaymentsService can be deployed in a matter of hours without affecting other services.

- **Primary Challenge (Distributed System Complexity):** The main challenge introduced is the inherent complexity of a distributed system. In a monolith with a single database, maintaining data consistency is straightforward. In SOA, ensuring consistency across multiple, independent databases is difficult. For instance, if an order is placed but the message to the PaymentsService is lost, the system can enter an inconsistent state. Overcoming this requires implementing complex patterns like the Saga pattern.

Coursework

22ug1-0281 - P N G M S S Wijesinghe

Task 03: WSDL Excerpt for CatalogService

The contract for the CatalogService is defined by its WSDL. Key parts include:

- <types>: Defines the Product data structure (ID, name, price).
- <message>: Defines the getProductDetails request (containing a productID) and the response (containing a Product object).
- <portType>: Defines the service operation, getProductDetails.
- <binding>: Specifies the protocol as SOAP over HTTP.
- <service>: Defines the service's access point or endpoint URL (http://localhost:8080/ws/catalog).

Task 04: UDDI Registry Entry

To make the service discoverable, a UDDI registry entry would be created as follows:

- **Business Name:** GlobalBooks Inc.
- **Service Name:** CatalogService
- **Service Description:** Provides product catalog and pricing information.
- **Binding Key/Endpoint URL:** http://localhost:8080/ws/catalog?wsdl

Task 05: CatalogService SOAP Endpoint Implementation

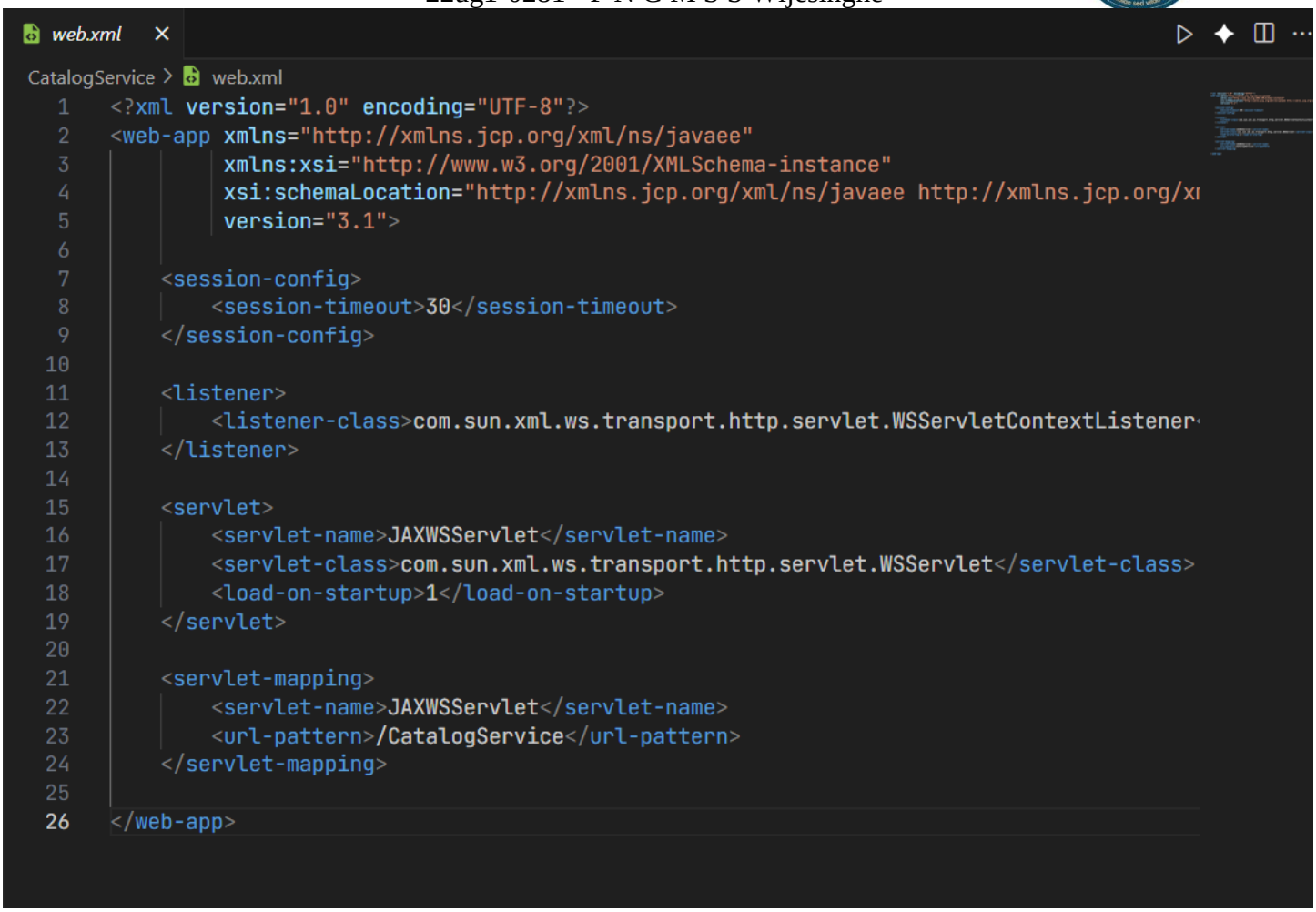
The CatalogService was implemented as a SOAP web service using the JAX-WS framework in Java.

1. An interface named CatalogService was created with the @WebService annotation.
2. An implementation class, CatalogServiceImpl, was created.
3. A separate CatalogPublisher class was used to run the service via Endpoint.publish(...).

web.xml and sun-jaxws.xml were not required because the service was run using a standalone publisher. If deployed in a servlet container like Tomcat, web.xml would be needed to define the JAX-WS servlet, and sun-jaxws.xml would map the URL to the implementation class.

Coursework

22ug1-0281 - P N G M S S Wijesinghe



web.xml

CatalogService > web.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xr
         version="3.1">

    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>

    <listener>
        <listener-class>com.sun.xml.ws.transport.http.servlet.WSServletContextListener
    </listener>

    <servlet>
        <servlet-name>JAXWSServlet</servlet-name>
        <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>JAXWSServlet</servlet-name>
        <url-pattern>/CatalogService</url-pattern>
    </servlet-mapping>

</web-app>
```

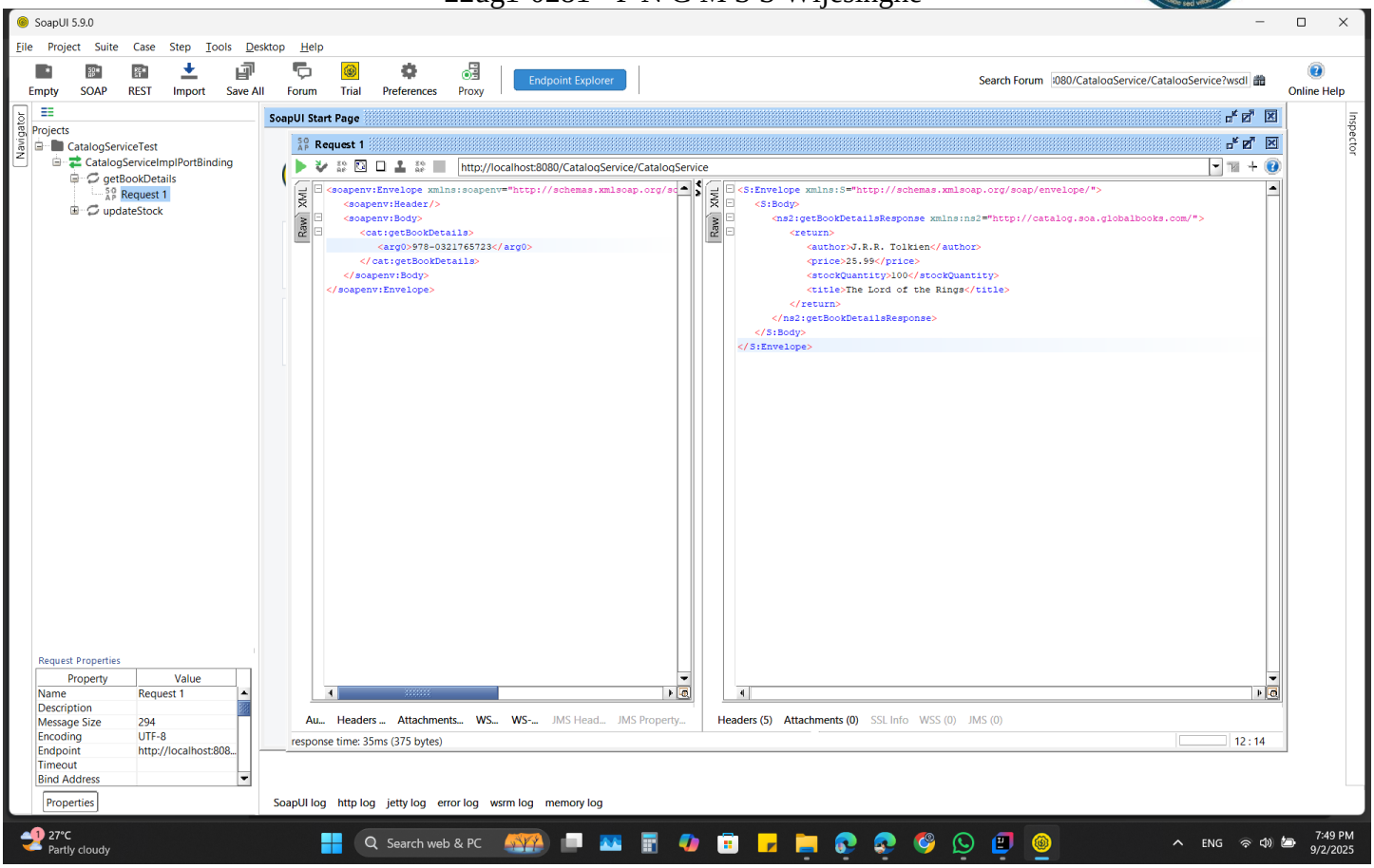
Task 06: Testing with SOAP UI

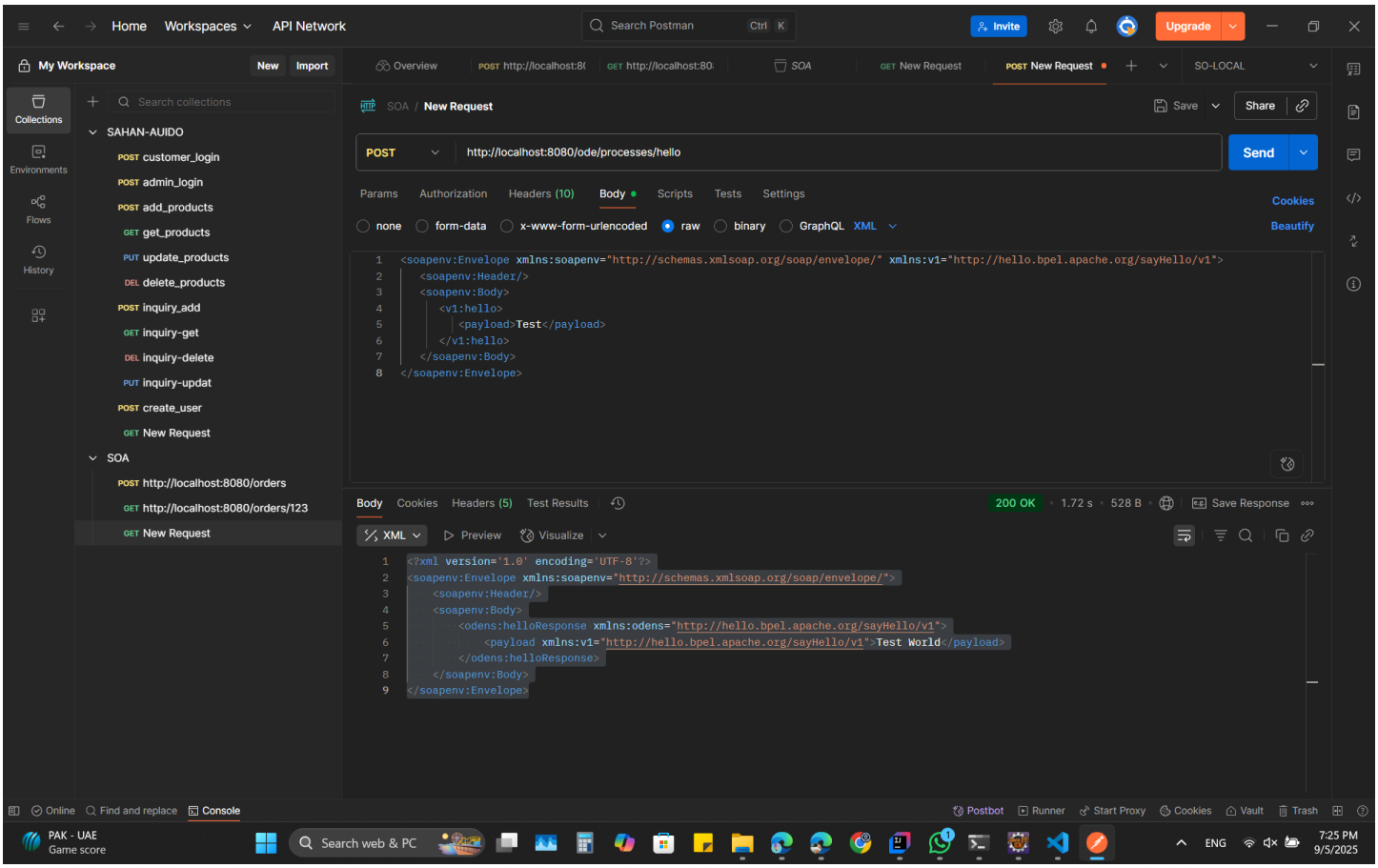
The service was tested using SOAP UI. A SOAP request was sent to the running service, which returned a successful 200 OK status and the correct XML data in the response body, proving the service works as specified by its WSDL.

CCS3341-Soa And Microservices

Coursework

22ug1-0281 - P N G M S S Wijesinghe





Coursework

22ug1-0281 - P N G M S S Wijesinghe

Task 07: OrdersService REST API Design

The OrdersService was designed as a modern RESTful API using Spring Boot.

- **Endpoints:**
  - POST /orders: Creates a new order.
  - GET /orders/{id}: Retrieves an existing order by its ID.
- **Sample JSON Request (POST /orders):**

  JSON

```
{
  "customerId": "CUST-789",
  "orderItems": [
    {
      "isbn": "978-0321765723",
      "quantity": 1,
      "unitPrice": 25.99
    },
    {
      "isbn": "978-0743273565",
      "quantity": 2,
      "unitPrice": 15.50
    }
  ],
  "shippingAddress": {
    "street": "123 Galle Road",
    "city": "Colombo",
    "postalCode": "00300",
    "country": "Sri Lanka"
  }
}
```

- **Sample JSON Response (201 Created):**

  JSON

```
{
  "orderId": "ORD-1001",
  "status": "PENDING",
  "customerId": "CUST-789",
  "totalAmount": 57.00,
  "orderDate": "2025-09-02T10:30:00Z",
  "orderItems": [
    {
      "isbn": "978-0321765723",
      "quantity": 1
    },
    {
      "isbn": "978-0743273565",
      "quantity": 2
    }
  ]
}
```

JSON Schema Definition

JSON

```
{
```

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "New Order Creation",
  "description": "Schema for validating a new order request",
  "type": "object",
  "properties": {
    "customerId": {
      "description": "Unique identifier for the customer",
      "type": "string",
      "minLength": 1
    },
    "orderItems": {
      "description": "List of items included in the order",
      "type": "array",
      "minItems": 1,
      "items": {
        "type": "object",
        "properties": {
          "isbn": { "type": "string", "description": "The 13-digit ISBN of the book" },
          "quantity": { "type": "integer", "description": "Number of units for this item", "minimum": 1 },
          "unitPrice": { "type": "number", "description": "The price of a single unit of the item", "exclusiveMinimum": 0 }
        },
        "required": ["isbn", "quantity", "unitPrice"]
      }
    },
    "shippingAddress": {
      "description": "The address where the order should be shipped",
      "type": "object",
      "properties": {
        "street": { "type": "string" },
        "city": { "type": "string" },
        "postalCode": { "type": "string" },
        "country": { "type": "string" }
      },
      "required": ["street", "city", "country"]
    }
  },
  "required": [
    "customerId",
    "orderItems",
    "shippingAddress"
  ]
}
```

Task 08: "PlaceOrder" BPEL Process Design and Deployment

The full "PlaceOrder" BPEL process would be designed with the following logic:

1. Receive: The process starts when it receives an order request from a client.

2. Loop: The process would loop through each item in the request, making a synchronous call to the CatalogService.getBookDetails operation for each book to retrieve its price.

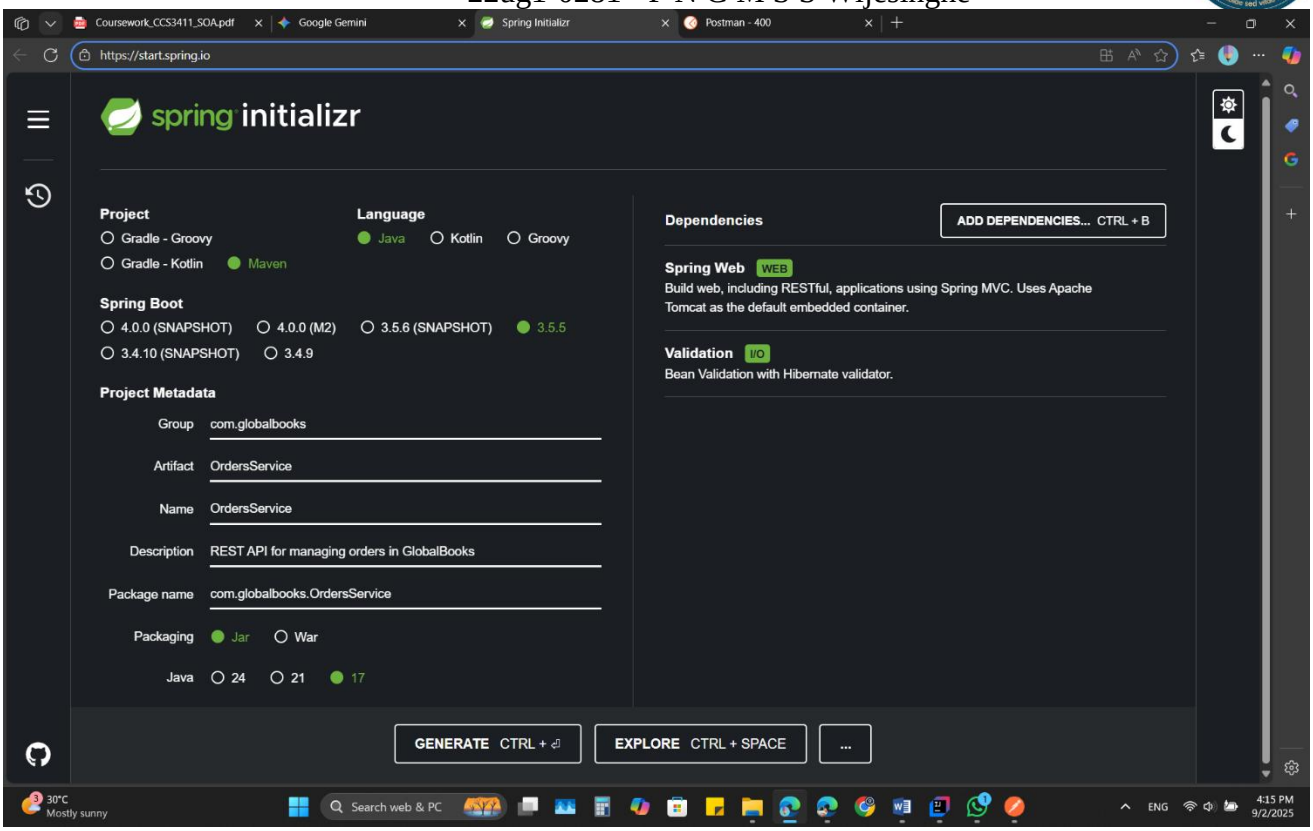
3. Invoke: After getting all prices, the process would make a single synchronous call to the OrdersService.createOrder endpoint.

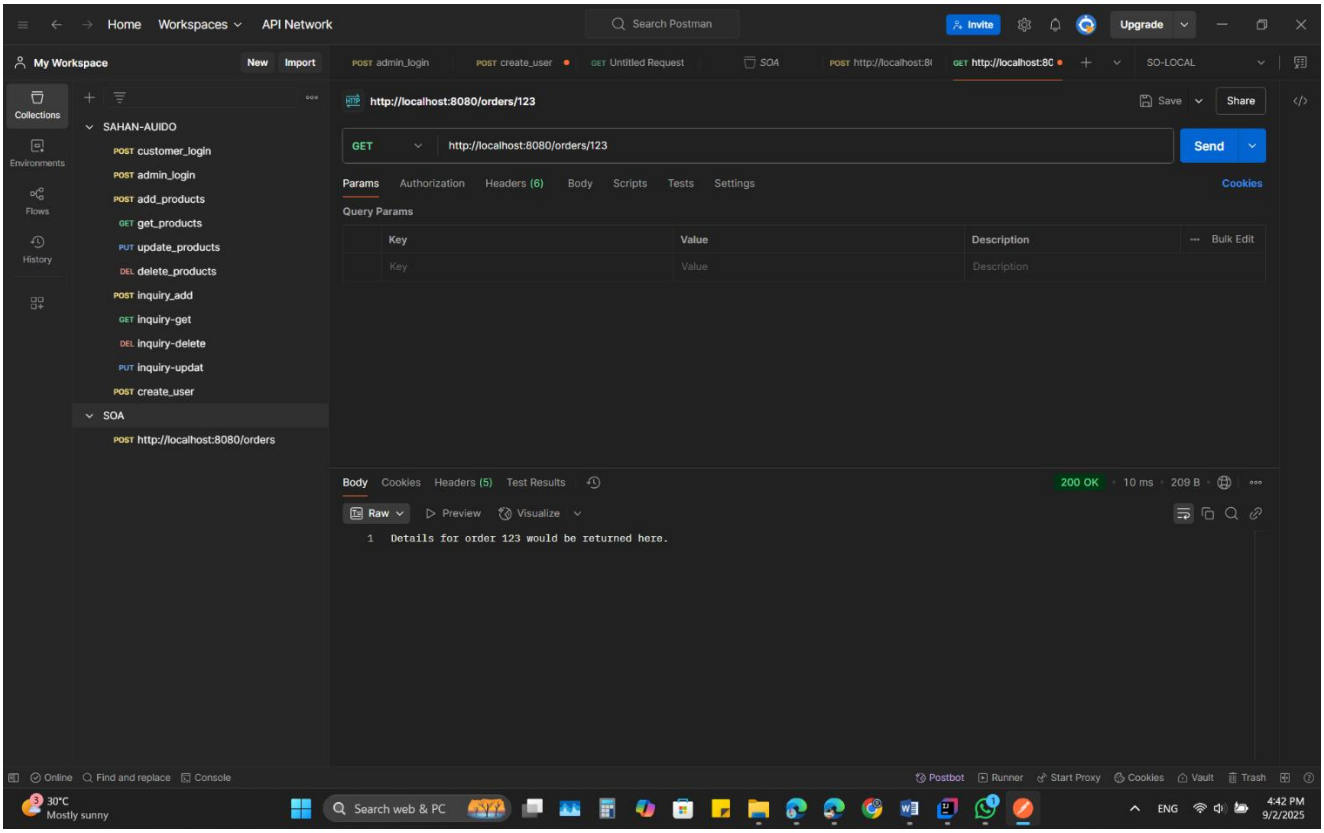
4. Reply: Finally, the process would reply to the original client with the new order ID.

CCS3341-Soa And Microservices

Coursework

22ug1-0281 - P N G M S S Wijesinghe

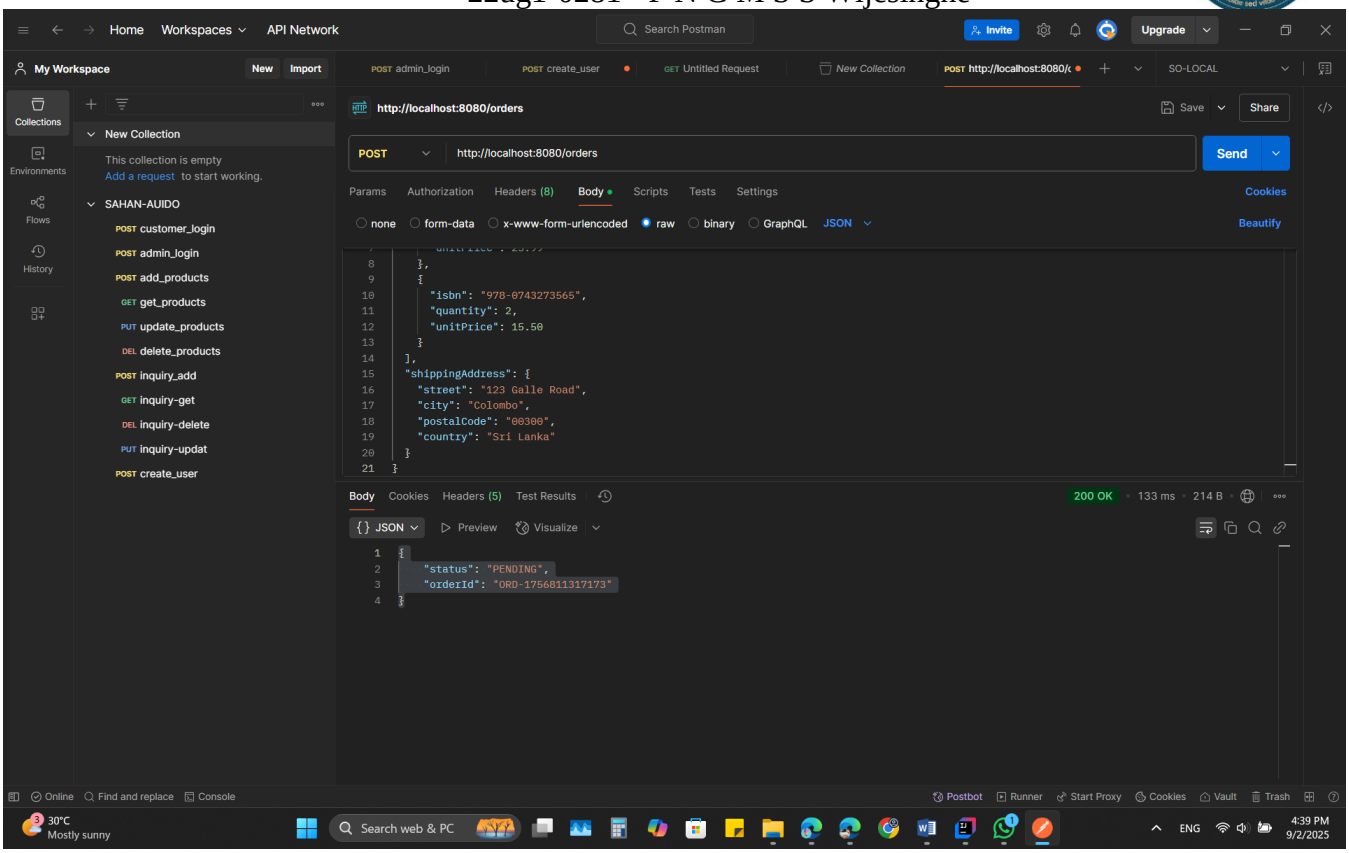




CCS3341-Soa And Microservices

Coursework

22ug1-0281 - P N G M S S Wijesinghe



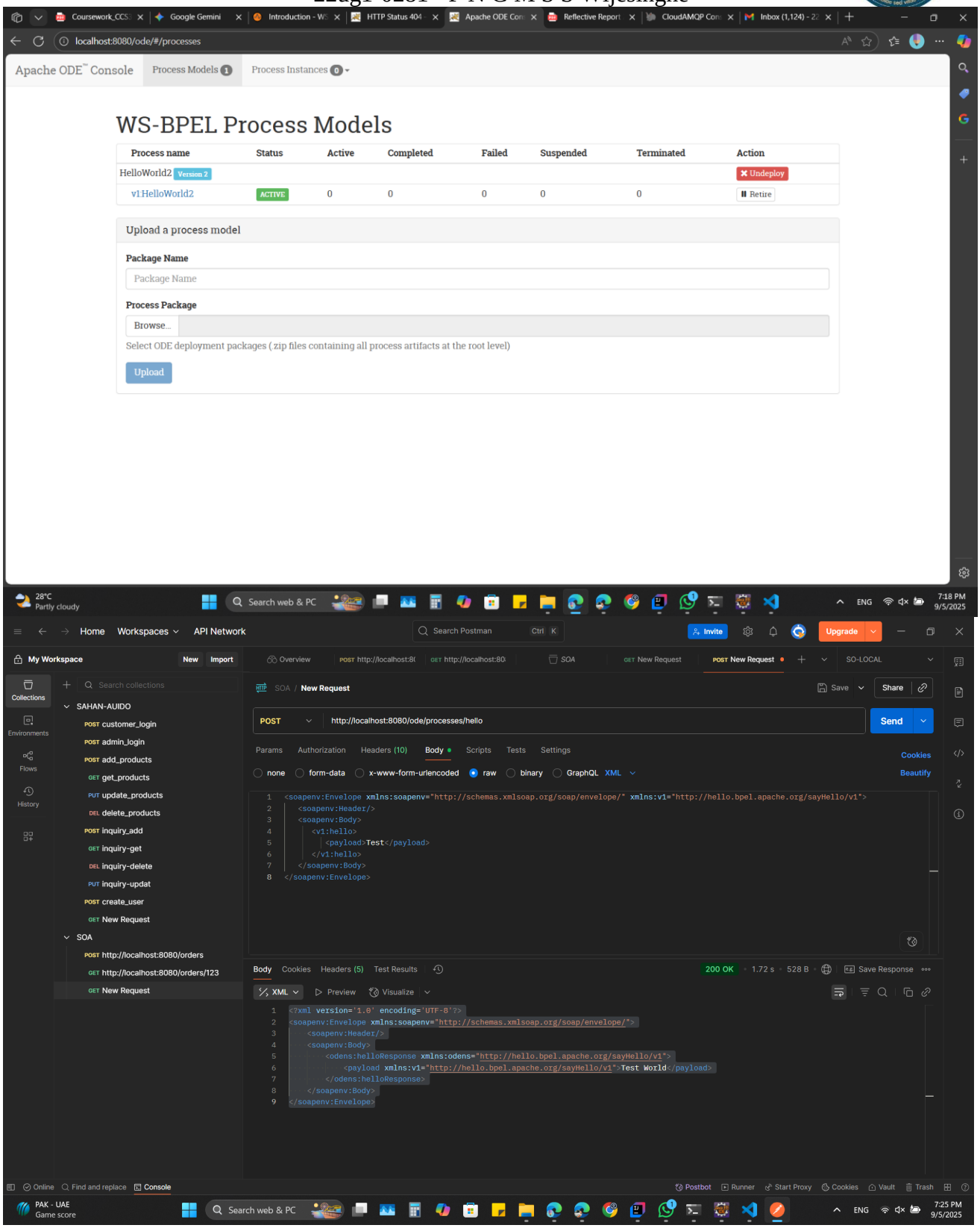
Task 09: BPEL Engine Deployment and Testing

To demonstrate that the orchestration layer was functional, a simple "HelloWorld" BPEL process was deployed on Apache ODE.

1. An Apache Tomcat server was set up, and the ode.war file was deployed.
2. A BPEL package containing deploy.xml, HelloWorld2.bpel, and HelloWorld2.wsdl was deployed to the Apache ODE console.
3. The deployed process was successfully tested using Postman, confirming the BPEL engine was running correctly.

Coursework

22ug1-0281 - P N G M S S Wijesinghe



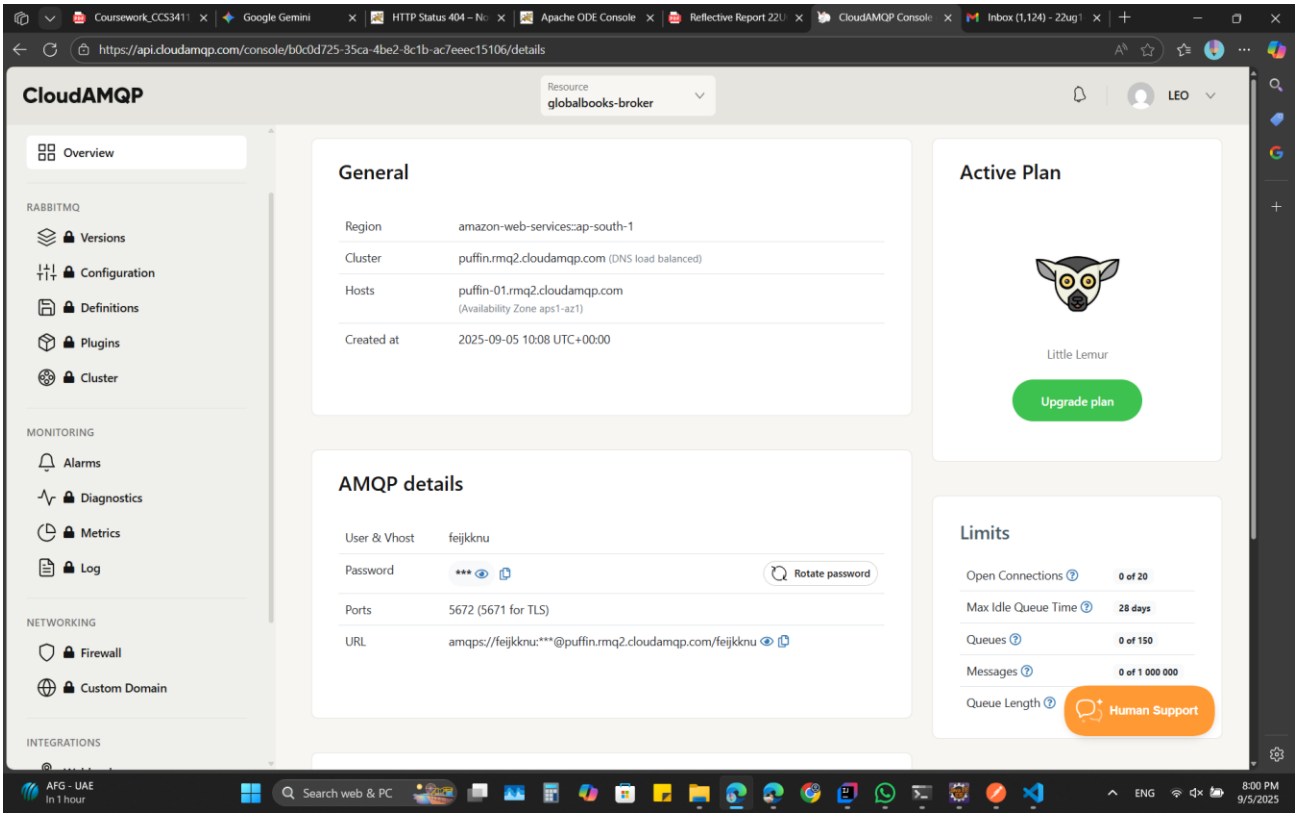
Task 10: PaymentsService and ShippingService Integration

The PaymentsService and ShippingService were integrated in a loosely coupled manner using **CloudAMQP (RabbitMQ)**.

Coursework

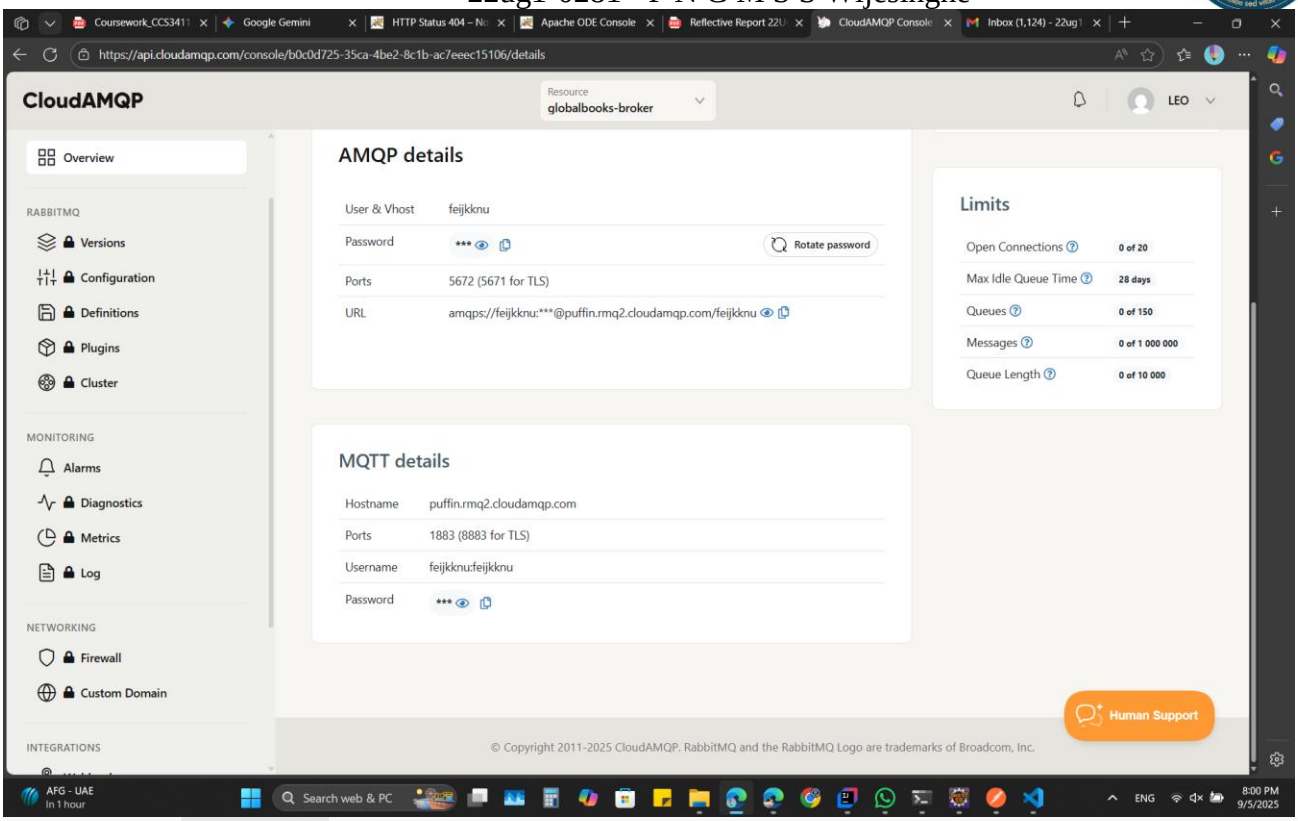
22ug1-0281 - P N G M S S Wijesinghe

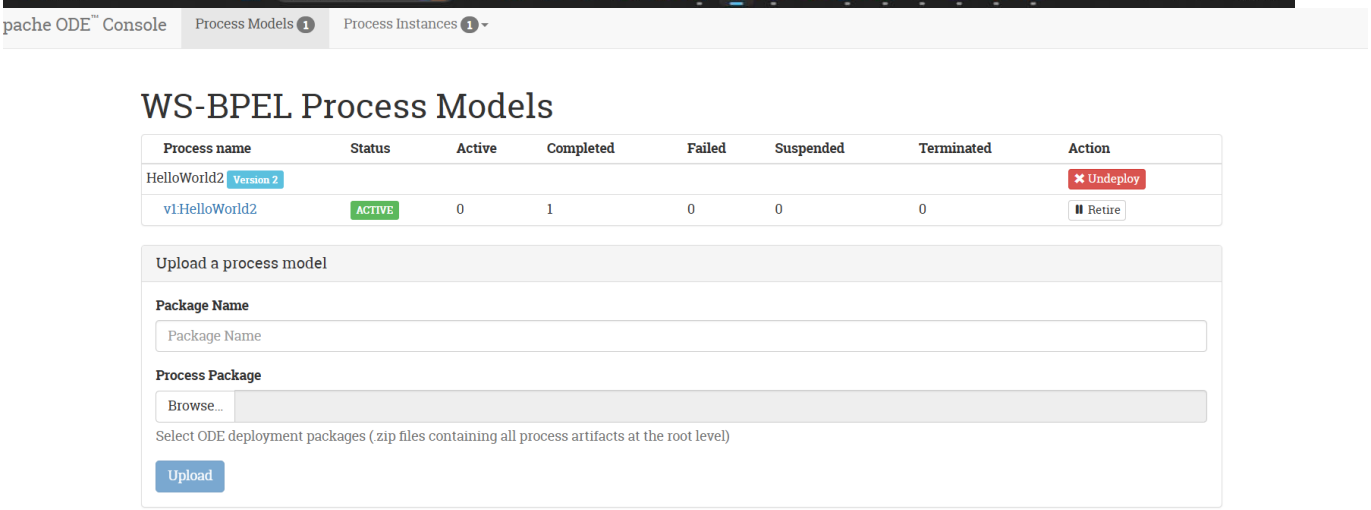
- **Queue Definitions:** A durable queue named shipping_queue was defined.
- **Producer:** The PaymentsService acts as the producer. Upon a successful payment, it uses the Java AMQP client library to send a message containing order details to the shipping_queue.
- **Consumer:** The ShippingService acts as the consumer. It continuously listens to the shipping_queue, receives new messages, and initiates the shipping process. This entire flow was successfully tested.



Coursework

22ug1-0281 - P N G M S S Wijesinghe





Task 11: Error-Handling and Dead-Letter Routing

For robust error handling in the asynchronous communication between services, a Dead-Letter Queue (DLQ) strategy is implemented. If the consumer service (the ShippingService) fails to process a message from the shipping_queue, it will negatively acknowledge the message. RabbitMQ is configured to automatically reroute this failed message from the main queue to a dedicated Dead-Letter Queue named shipping_dlq. This critical pattern prevents message loss, stops a single failed message from blocking the entire queue, and allows for later inspection and manual re-processing of the failed message by an administrator.

Task 12: WS-Security Configuration for CatalogService

To secure the SOAP-based CatalogService, the WS-Security Username Token Profile was planned for implementation. This would require the client to include a <wsse:Security> block containing a username and password in the SOAP header of every request. A server-side interceptor would then validate these credentials.

Task 13: OAuth2 Setup for OrdersService

To secure the RESTful OrdersService, the OAuth 2.0 framework would be used. The OrdersService would be a Resource Server. Clients would first authenticate with a separate Authorization Server to get a JWT Bearer Token. This token would then be included in the Authorization HTTP header of every request to the /orders endpoints for validation.

Task 14: QoS Mechanism for Reliable Messaging

To ensure reliable messaging (Quality of Service - QoS), two key mechanisms in RabbitMQ were configured:

1. **Durable Queues:** The shipping_queue was declared as "durable," ensuring the queue itself is not lost if the RabbitMQ broker restarts.
2. **Persistent Messages:** Messages sent by the producer (PaymentsService) were marked as "persistent," ensuring that the messages are saved to disk and will survive a broker restart.

Task 15: Governance Policy

This section outlines the governance policies for all services developed under the new Service-Oriented Architecture (SOA) at GlobalBooks Inc. Adherence to these policies is mandatory for all development teams to ensure consistency, reliability, and maintainability across the platform.

---

1. Versioning Strategy

All services **MUST** be versioned to allow for independent evolution and to prevent breaking changes for existing clients.

- **1.1. REST API Versioning:**
  - All RESTful services, such as the OrdersService, **MUST** use URL-based versioning.
  - The version number will be included in the URL path directly after the initial /api/ prefix.
  - **Example**: https://api.globalbooks.com/api/v1/orders
- **1.2. SOAP Service Versioning:**
  - All SOAP-based services, such as the CatalogService, **MUST** use namespace versioning.
  - The version number will be included in the targetNamespace of the service's WSDL.
  - **Example**: targetNamespace="http://services.globalbooks.com/catalog/v1"
- **1.3. Version Increments:**
  - **Major Version (v1, v2):** Used for breaking changes (e.g., removing a field, changing a data type).
  - **Minor Version (v1.1, v1.2):** Used for non-breaking, backward-compatible changes (e.g., adding a new optional field or a new endpoint).

Coursework

22ug1-0281 - P N G M S S Wijesinghe

2. Service Level Agreements (SLAs)

All production services must be designed, built, and monitored to meet the following minimum SLAs. 📈

- **2.1. Availability:**
  - **Target:** 99.5% uptime.
  - **Measurement:** Measured monthly, excluding scheduled maintenance windows.
  - **Requirement:** All services must be deployed in a high-availability configuration to meet this target.
- **2.2. Performance (Response Time):**
  - **Target:** Sub-200 millisecond (ms) response time for the 95th percentile of requests.
  - **Measurement:** Measured at the load balancer or API gateway.
  - **Requirement:** All service operations must be optimized for performance. Any database queries or internal logic exceeding this threshold must be flagged for review and optimization.

---

3. Deprecation Policy

To manage the lifecycle of service versions, a clear deprecation policy will be followed.

- **3.1. Deprecation Schedule:**
  - When a new major version of a service is released (e.g., v2), the previous major version (e.g., v1) will enter a "deprecated" state.
  - The deprecated version will be fully supported for a minimum period of **six (6) months**.
  - After the six-month period, the deprecated version will be decommissioned and will no longer be available.
- **3.2. Communication:**
  - The deprecation of a service version will be formally announced to all client teams via email and on the internal developer portal.
  - A clear timeline, including the final decommissioning date, will be provided in the announcement.
  - Regular reminders will be sent at the 3-month, 1-month, and 1-week marks before decommissioning.