CCS3341 SOA & Microservices Coursework

K.K.Pahan Bimsara
22ug1-0471

Introduction:

The previously effective monolithic application at GlobalBooks Inc. began to suffer significant setbacks in scalability, maintainability and deployment agility as the business gained size. To address these issues a strategic project was carried out to refactor the monolith into a set of autonomous, independent, services, based on the ideas of ServiceOriented Architecture (SOA). The system was broken down into four core services, which were Catalog, Orders, Payments, and Shipping. This report describes the design principles, technology, implementation strategies, and challenges encountered in this migration project.

Task 1 : SOA design principles

- **Service Loose Coupling** : I used RabbitMQ to structure an asynchronous messaging pattern in order to make the services independent. On creating a new order, the Ordersservice publishes an OrderCreated event to an exchange, and does not make direct calls to the Payments or Shipping services. This decouples the services and provides more resilience to the system.

- **Service Autonomy** : I made sure that each service was autonomous, that is, it had its own assigned database. As an illustration, the product database is under the exclusive control of the Catalogservice. Any other service access should occur via its published API which can be independently developed and deployed.

- **Standardized Service Contract** : I created formal services contracts to every service. The Catalogservice provides the SOAP-based web service that is defined by WSDL file to facilitate legacy partners. To a contemporary client, the OrdersService has opened a JSON-based RESTful interface.

Task 2 : Key benefit and key challenge

- **Key Benefit - Improved Agility** : The biggest advantage was that of improved agility. In independent services, a switch between one service (e.g. PaymentsService) does not need a system-wide redeployment. This minimally risks and minimally increases the time-to-market of new features.

- Key Challenge - Complexity of a Distributed System: The crucial issue was how to deal with complexity of a distributed system. A single call is now a multi-service call, bringing the problems of network latency and distributed debugging. This brought to the fore the significance of centralized logging and monitoring tools.

Task 3 : WSDL excerpt for the CatalogService

```
<definitions name="CatalogService"
    targetNamespace="http://globalbooks.com/catalog" ...>
    <types>
      <xsd:schema ...>
        <xsd:element name="ProductDetailsRequest" ... />
        <xsd:element name="ProductDetailsResponse" ... />
      </xsd:schema>
    </types>
    <portType name="CatalogServicePortType">
      <operation name="getProductDetails">
        <input message="tns:getProductDetailsRequest"/>
        <output message="tns:getProductDetailsResponse"/>
      </operation>
    </portType>
    ...
</definitions>
```

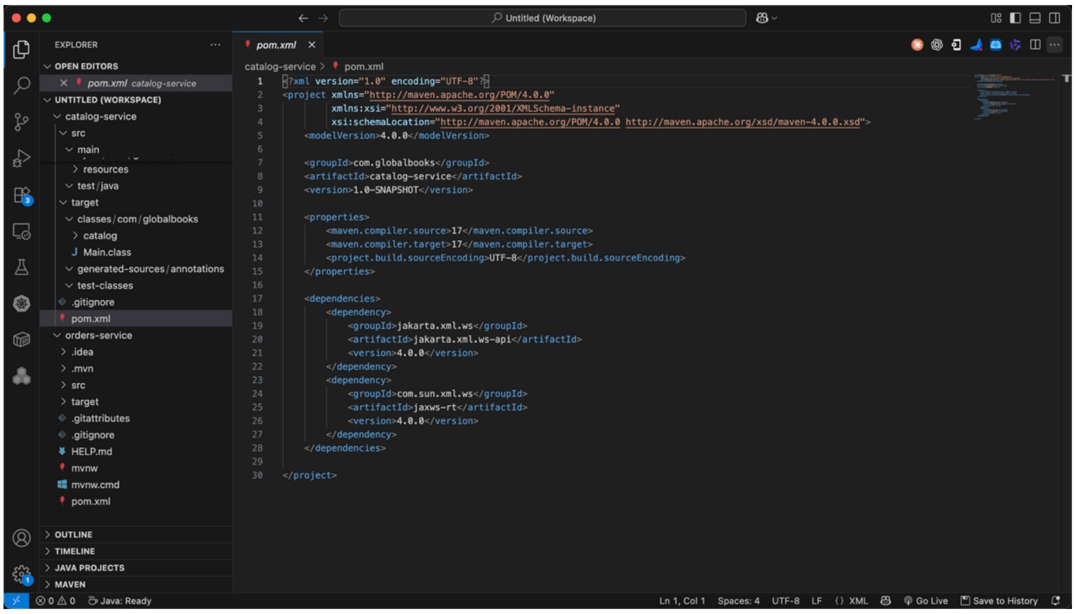
Task 4 : UDDI registry entry metadata.

- 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 5 : implemented the CatalogService SOAP endpoint.

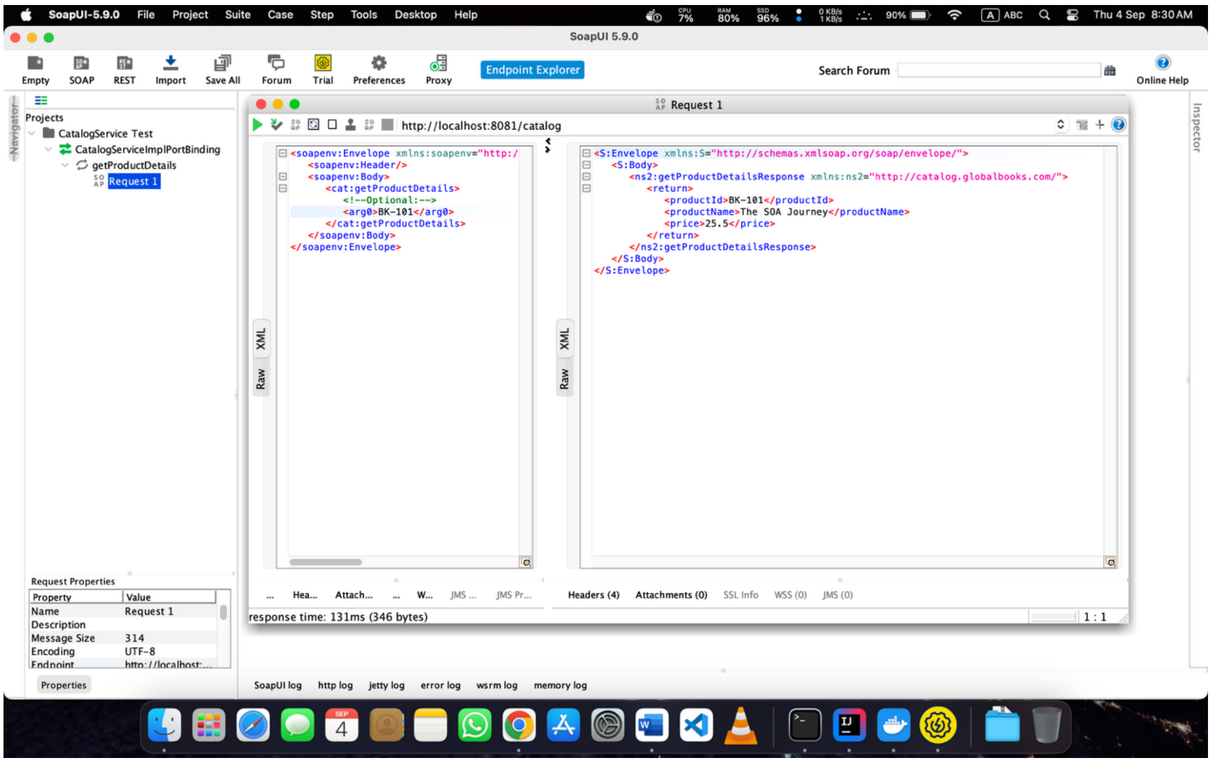
I implemented the CatalogService endpoint using JAX-WS. I used annotations like @WebService and @WebMethod to define the service contract directly in the Java code. For this project, I published the service using the simple Endpoint.publish() method, which runs on a lightweight, built-in HTTP server, making it easy to test.

```
public class Publisher {
  public static void main(String[] args) {
    Endpoint.publish("http://localhost:8081/catalog", new CatalogServiceImpl());
    System.out.println("CatalogService is live.");
  }
```



```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.globalbooks</groupId>
    <artifactId>catalog-service</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>jakarta.xml.ws</groupId>
            <artifactId>jakarta.xml.ws-api</artifactId>
            <version>4.0.0</version>
        </dependency>
        <dependency>
            <groupId>com.sun.xml.ws</groupId>
            <artifactId>jaxws-rt</artifactId>
            <version>4.0.0</version>
        </dependency>
    </dependencies>

</project>
```

Task 6 : Tested it using SoapUI



http://localhost:8081/catalog

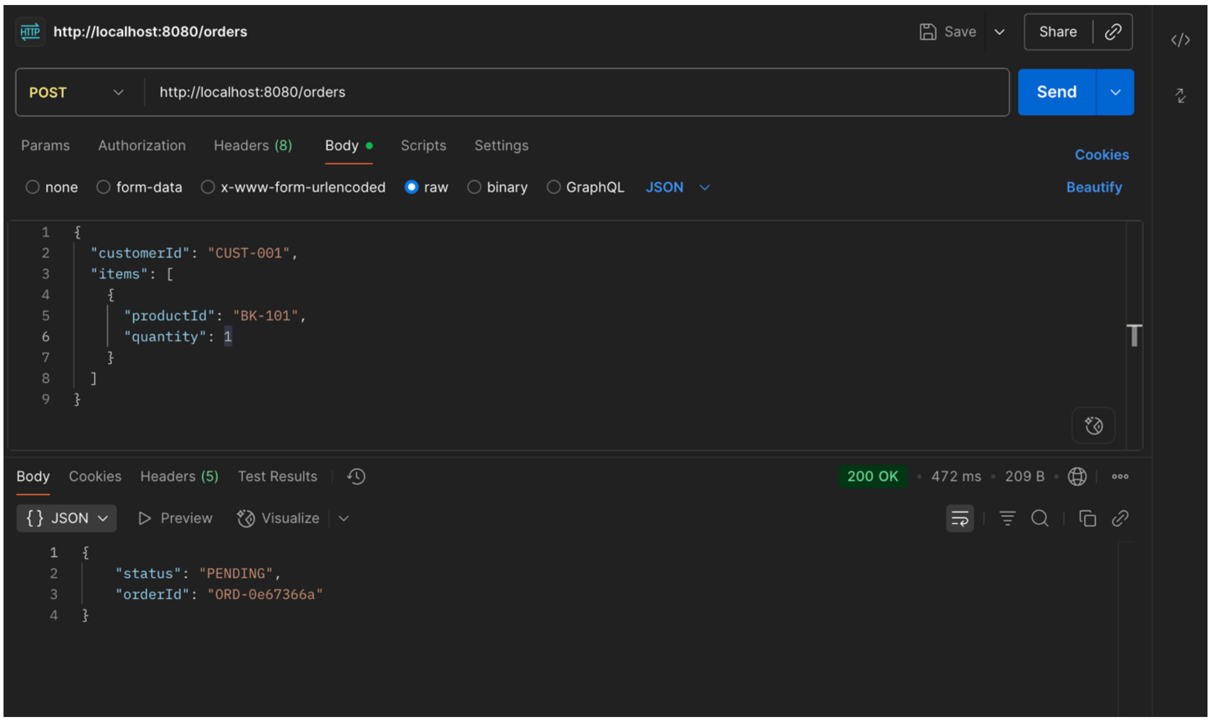
```xml
<soapenv:Envelope xmlns:soapenv="http:/
   <soapenv:Header/>
   <soapenv:Body>
      <cat:getProductDetails>
         <!--Optional:-->
         <arg0>BK-101</arg0>
      </cat:getProductDetails>
   </soapenv:Body>
</soapenv:Envelope>
```

```xml
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:getProductDetailsResponse xmlns:ns2="http://catalog.globalbooks.com/">
         <return>
            <productId>BK-101</productId>
            <productName>The SOA Journey</productName>
            <price>25.5</price>
         </return>
      </ns2:getProductDetailsResponse>
   </S:Body>
</S:Envelope>
```

response time: 131ms (346 bytes)

Property	Value
Name	Request 1
Description	
Message Size	314
Encoding	UTF-8
Endpoint	http://localhost:...

Task 7 : Design the OrdersService REST API



POST http://localhost:8080/orders

```
{
  "customerId": "CUST-001",
  "items": [
    {
      "productId": "BK-101",
      "quantity": 1
    }
  ]
}
```

200 OK • 472 ms • 209 B

```
{
    "status": "PENDING",
    "orderId": "ORD-0e67366a"
}
```

Task 8 : Outline the "PlaceOrder" BPEL process.

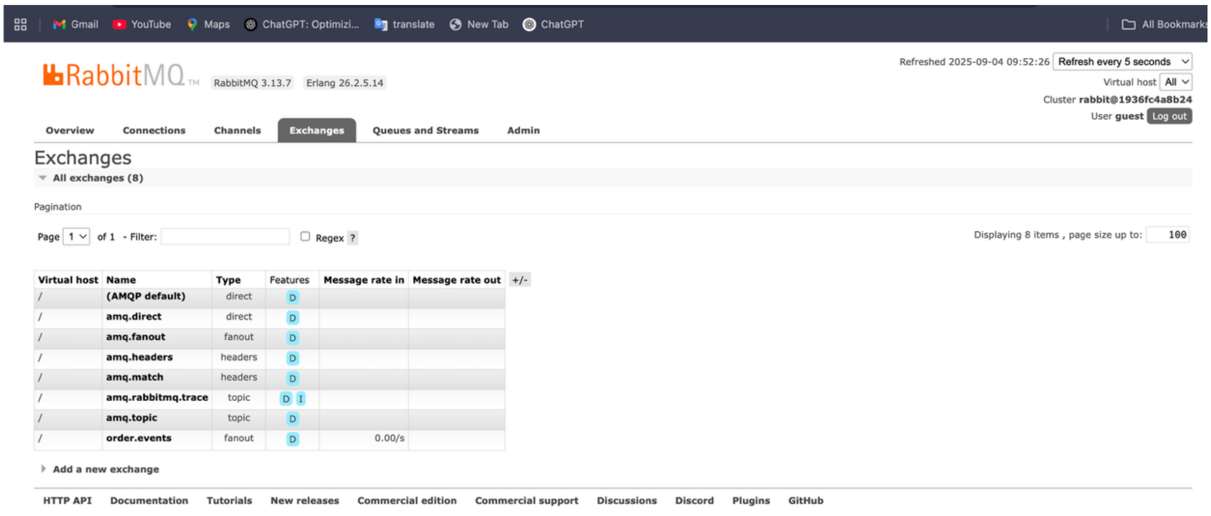
My PlaceOrder BPEL process plan orchestrates the services to call the services in a certain order: it would receive an order first, then loop over the items calling the Catalogservice to get a price and Invoke the Orderservice to generate the order and lastly Reply to the client with a confirmation.

Task 9 : Explain deployment and testing on a BPEL engine.

To implement it theoretically, I would have used Apache ODE engine. It would be done by packaging the BPEL files in a zip file and uploading it to the ODE console. To test, I would invoke SoapUI to invoke the WSDL endpoint of the BPEL process itself and watch the ODE console to validate that it invoked the downstream services properly.

Task 10 : Explain how you integrated PaymentsService and ShippingService

I adopted a RabbitMQ, event-driven and asynchronous implementation. The Ordersservice is a producer, and it broadcasts an OrderCreated event to a Fanout Exchange that is called order.events. The PaymentsService and ShippingService would be consumers, and each had its queue attached to this exchange. This separates the services and enhances the resilience of the systems.



RabbitMQ 3.13.7 Erlang 26.2.5.14

Refreshed 2025-09-04 09:52:26 Refresh every 5 seconds
Virtual host All
Cluster rabbit@1936fc4a8b24
User guest

Exchanges

All exchanges (8)

Pagination

Page 1 of 1 - Filter: Regex ?

Displaying 8 items , page size up to: 100

Virtual host	Name	Type	Features	Message rate in	Message rate out
/	(AMQP default)	direct	D		
/	amq.direct	direct	D		
/	amq.fanout	fanout	D		
/	amq.headers	headers	D		
/	amq.match	headers	D		
/	amq.rabbitmq.trace	topic	D I		
/	amq.topic	topic	D		
/	order.events	fanout	D	0.00/s	

Task 11 : Describe your error-handling and dead-letter routing strategy.

Two of the most frequently used patterns in my error-handling approach to consumers of messages were: Retries with Exponential Backoff in the case of temporary failures, and Dead-Letter Queue (DLQ) to send messages that fail many times. This is to make sure that there is no loss of messages and can also inspect failed transactions manually.

Task 12 : Detail WS-Security configuration for CatalogService.

In order to ensure the security of the SOAP service, I adopted the WS-Security UsernameToken Profile. A client would append a header of credentials and a <wsse:Security> to the SOAP message. A JAX-WS Handler would intercept these credentials on the server and then perform the appropriate validation before the server would move on to the next stage of processing the request.

Task 13 : Explain OAuth2 setup for OrdersService

In the case of the REST API, I decided to use OAuth2 in the Client Credentials Grant flow. An Authorization Server would initially issue a JWT Access Token a client. It would then add this token to Authorization: Bearer <token> header of each request. I would have Spring Security to configure the Ordersservice to support this token before it can access the service.

Task 14 : Explain one QoS mechanism you configured for reliable messaging.

Persistent Messages in RabbitMQ was the most important QoS mechanism I have configured to guarantee reliable messaging. I enabled the RabbitMQ broker to store the messages in disk by setting the delivery mode to persistent. This will ensure that messages are not lost when a broker is restarted.

Task 15 : Draft the governance policy.

developed a policy of governance that addressed three key aspects:

- Versioning: Versioning of REST services (e.g. /api/v1/orders) and versioning of SOAP services using Namespace versioning.

- SLA Targets: Having clear targets such as a 99.5% uptime and a response time that is less than 200ms.

- Depreciation Plan: This involves a formal strategy that is stated in advance by at least 6 months prior to the retirement of outdated versions of service.

Task 16 : Deploy all four services to a cloud platform.

The Cloud deployment plan I would recommend is Amazon Web Services (AWS). The applications of Spring boot would be put in the Docker containers and run on Elastic Container Service (ECS) by AWS. I would have a separate database in Amazon RDS per service, and managed message broker service using Amazon MQ on top of Amazon RabbitMQ.