Service-Oriented Architecture Implementation for GlobalBooks Inc.

CCS3341 SOA & Microservices

21UG1287 - Naveen Prabodha

1. Introduction

GlobalBooks Inc. is a fast-growing e-commerce platform that initially relied on a Java-based monolithic application to manage catalog lookups, order placement, payments, and shipping. As the company expanded across multiple regions and faced increased demand during peak events, the limitations of the monolithic system became evident poor scalability, tightly coupled components, and complex redeployment cycles.

To address these challenges, this project refactors the monolithic system into a Service-Oriented Architecture (SOA), consisting of four autonomous services: Catalog, Orders, Payments, and Shipping. The implementation integrates SOAP and REST interfaces, asynchronous messaging with RabbitMQ, and orchestration using BPEL. The goal is to achieve scalability, maintainability, and flexibility in service composition while ensuring governance and security standards.

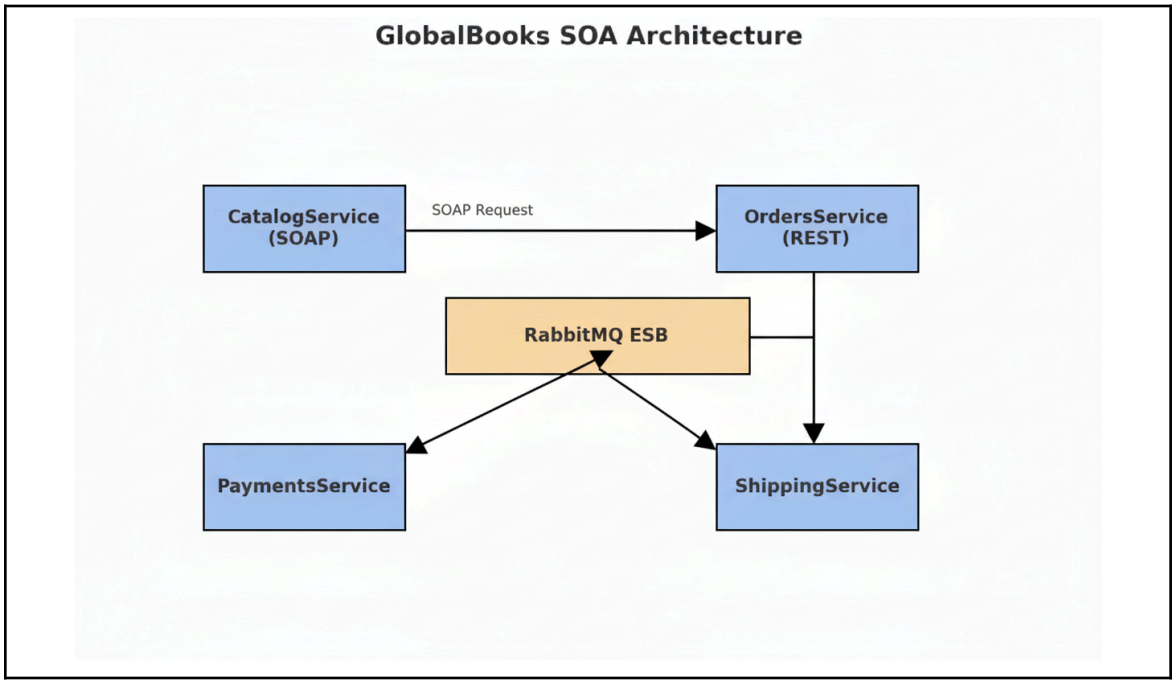

Figure 1: GlobalBooks SOA Architecture.

2. SOA Design Principles & Trade-offs

Applied Principles

1. Loose Coupling – Services are independent, communicating via standardized protocols (SOAP/REST).
2. Service Autonomy – Each service owns its data and logic (Catalog DB, Orders DB, etc.).
3. Discoverability – Services are published in a UDDI registry for dynamic client discovery.
4. Reusability – Catalog and Orders services are designed for multiple consumers (internal apps, partners, third-party retailers).
5. Composability – Services are orchestrated into the "PlaceOrder" business process using BPEL.

Benefit

Scalability: Each service can be deployed and scaled independently (e.g., OrdersService scaled separately during promotions).

Challenge

Data Consistency: Distributed services introduce challenges with eventual consistency and transaction management across multiple services. Event-driven patterns and compensating transactions were used to address this.

3. Service Design Artifacts

3.1 WSDL Excerpt for CatalogService

The CatalogService provides SOAP-based operations for book lookup and search. Below is a shortened excerpt of the WSDL definition:

```
<definitions name="CatalogService"
             targetNamespace="http://globalbooks.com/catalog"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:tns="http://globalbooks.com/catalog">

  <portType name="CatalogPortType">
      <operation name="getBook">
          <input message="tns:GetBookRequest"/>
          <output message="tns:GetBookResponse"/>
```

```
      </operation>
      <operation name="searchBooks">
          <input message="tns:SearchBooksRequest"/>
          <output message="tns:SearchBooksResponse"/>
      </operation>
  </portType>
</definitions>
```

3.2 UDDI Registry Entry Metadata (Task 4)

The services are published in a UDDI registry for discoverability. Example entry for CatalogService:

```
<businessService serviceKey="uuid:catalog-service-001"
                 businessKey="uuid:globalbooks-12345">
    <name>CatalogService</name>
    <description>Book catalog lookup and search service</description>
    <bindingTemplates>
        <bindingTemplate bindingKey="uuid:catalog-binding-001"
                         serviceKey="uuid:catalog-service-001">
            <accessPoint useType="endPoint">
                http://api.globalbooks.com/ws/catalog
            </accessPoint>
        </bindingTemplate>
    </bindingTemplates>
</businessService>
```

3.3 Implementation of CatalogService SOAP Endpoint

The CatalogService was implemented in Java using Spring Web Services. Configuration files define service deployment.

```
sun-jaxws.xml
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime">
    <endpoint name="CatalogService"

implementation="com.globalbooks.catalog.CatalogEndpoint"
              url-pattern="/ws/catalog"/>
</endpoints>
```

```
web.xml
<servlet>
```

```
    <servlet-name>jaxws-servlet</servlet-name>

<servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servl
et-class>
</servlet>
<servlet-mapping>
    <servlet-name>jaxws-servlet</servlet-name>
    <url-pattern>/ws/*</url-pattern>
</servlet-mapping>
```

4. Testing the SOAP Service

The CatalogService SOAP endpoint was tested using SOAP UI. Test cases included valid ISBN lookups, invalid ISBN requests, and keyword searches. Assertions were added to verify response structure and WS-Security headers.

Example SOAP request envelope:

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:cat="http://globalbooks.com/catalog">
   <soapenv:Header>
                                                        <wsse:Security
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-w
ssecurity-secext-1.0.xsd">
         <wsse:UsernameToken>
            <wsse:Username>admin</wsse:Username>
            <wsse:Password>admin123</wsse:Password>
         </wsse:UsernameToken>
      </wsse:Security>
   </soapenv:Header>
   <soapenv:Body>
      <cat:GetBookRequest>
         <cat:isbn>978-0134685991</cat:isbn>
      </cat:GetBookRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

5. OrdersService REST API

The OrdersService was implemented as a RESTful API using Node.js and Express. It manages order creation, retrieval, update, and cancellation. The service integrates with RabbitMQ to publish order events.

Main endpoints:

```
POST /orders - Create new order
GET /orders/{id} - Retrieve order by ID
PATCH /orders/{id}/status - Update order status
DELETE /orders/{id} - Cancel order
GET /orders - List all orders
```

Sample JSON request:

```
{
  "customerId": "CUST001",
  "items": [
    {"isbn": "978-0134685991", "quantity": 2, "price": 45.99}
  ],
  "shippingAddress": {
      "street": "123 Main St", "city": "Boston", "country": "USA",
"zipCode": "02101"
  }
}
Sample JSON schema excerpt:
{
  "type": "object",
  "properties": {
    "customerId": {"type": "string"},
    "items": {"type": "array"},
    "shippingAddress": {"type": "object"}
  },
  "required": ["customerId", "items", "shippingAddress"]
}
```

6. PlaceOrder BPEL Process

The PlaceOrder process orchestrates service calls to CatalogService and OrdersService. The workflow receives an order, loops through each item to fetch book details and prices, calculates the total, and then invokes the OrdersService to create the order.

Excerpt of BPEL definition:

```
<sequence>
          <receive     name="ReceiveOrder"     operation="placeOrder"
createInstance="yes"/>
  <forEach name="ProcessItems" counterName="itemCounter">
         <invoke   name="GetBookDetails"   partnerLink="catalogService"
operation="getBook"/>
  </forEach>
         <invoke     name="CreateOrder"     partnerLink="ordersService"
operation="createOrder"/>
  <reply name="ReplyToClient" operation="placeOrder"/>
</sequence>
```

7. Integration of Payments & Shipping Services

Integration between Orders, Payments, and Shipping is handled asynchronously via RabbitMQ. OrdersService publishes order events which are consumed by PaymentsService and ShippingService.

Queue configuration: *payment-queue, shipping-queue*

Error-handling: *Messages failing multiple retries are routed to a dead-letter queue for later inspection.*

8. Security Configurations

SOAP services secured using WS-Security UsernameToken.

Example configuration:

```
<bean id="securityInterceptor"

class="org.springframework.ws.soap.security.wss4j2.Wss4jSecurityInter
ceptor">
    <property name="validationActions" value="UsernameToken"/>
</bean>
```

REST services protected with OAuth2 Bearer tokens (JWT).

Sample token response:

```
{
  "access_token": "eyJhbGciOiJIUzI1...",
  "token_type": "Bearer",
  "expires_in": 3600
}
```

9. QoS Mechanism

Reliable messaging ensured through RabbitMQ durable queues and publisher confirms. This prevents message loss in case of broker failure.

10. Governance Policy

Versioning: URL-based (/v1/orders), namespaces (http://globalbooks.com/orders/v1).

SLA: Availability – 99.5%, Response time – <200ms.

Deprecation: 6-month notice period with parallel version support.

11. Deployment on Cloud

All four services were containerized using Docker and orchestrated via docker-compose. This setup enables local cloud-like deployment. RabbitMQ, OrdersService, PaymentsService, and ShippingService run as independent containers, ensuring scalability and isolation.

12. Reflective Analysis

SOAP vs REST: SOAP chosen for Catalog due to legacy compatibility, REST for Orders due to simplicity and adoption.

Event-driven vs synchronous APIs: Asynchronous messaging improved resilience but introduced eventual consistency challenges.

Scaling: Independent scaling of services was achieved, though monitoring and distributed logging remain essential for production.

13. Conclusion

This project successfully decomposed the monolithic GlobalBooks system into a Service-Oriented Architecture with four autonomous services. It demonstrated SOAP and REST integration, orchestration via BPEL, and governance strategies to ensure service reliability. Future improvements may include API Gateway integration, service mesh adoption, and enhanced monitoring with centralized dashboards.

14. References

W3C SOAP and WSDL specifications

RabbitMQ Documentation

Spring Web Services Reference

Node.js Documentation

Express.js Documentation