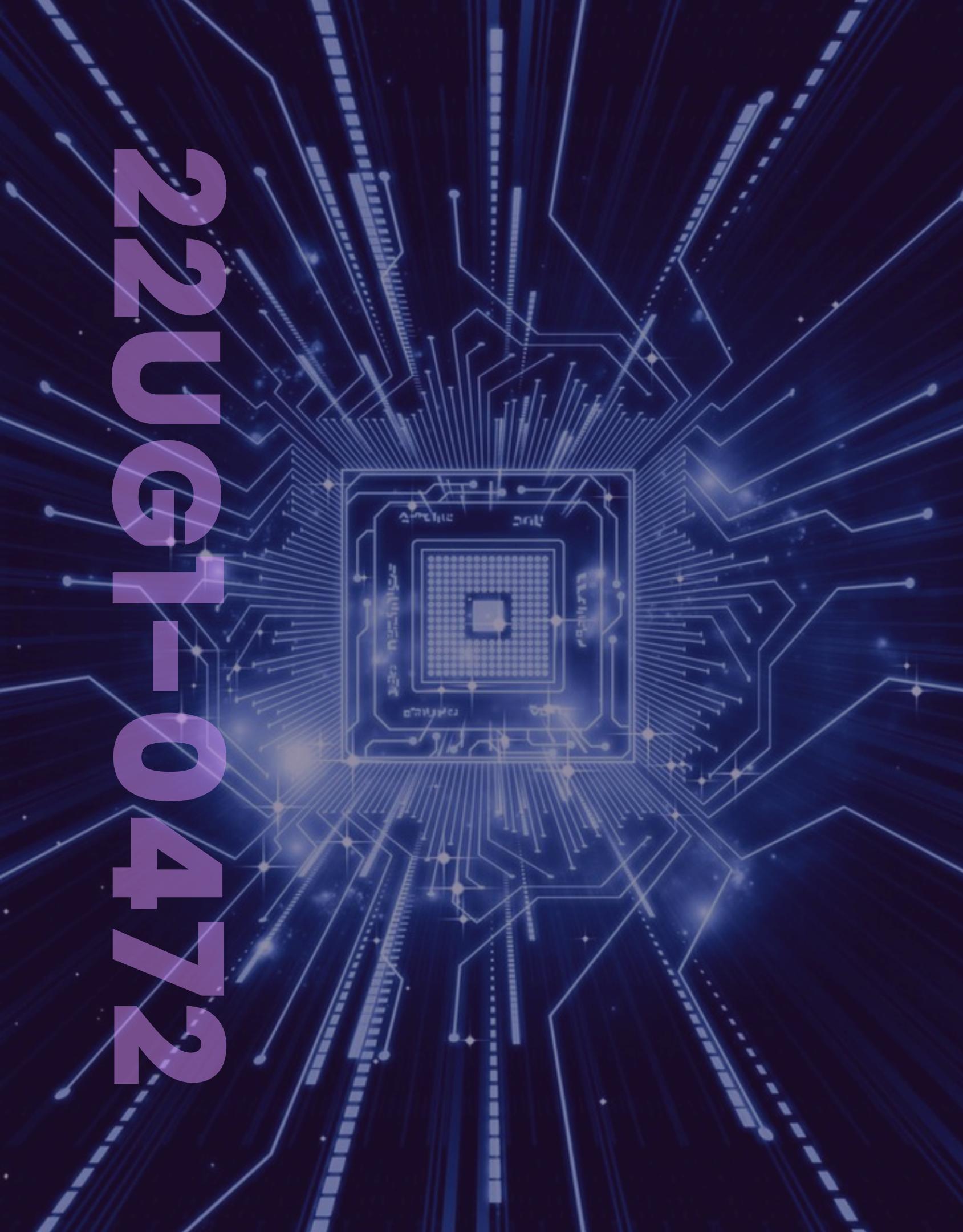


22 NOV 2024



# GLOBALBOOKS

## SOA REFACTORING PROJECT

A PRACTICAL IMPLEMENTATION OF SERVICE-ORIENTED ARCHITECTURE

K.G.P. Kavishka  
CCS3341: SOA & Microservices

# Agenda

## Presentation Outline

### The Problem

Understanding the limitations of the GlobalBooks Monolith.

### The Solution

Applying Core SOA Principles.

### Architectural Overview

The high-level design of the new system.

### Service Deep Dive

- CatalogService (SOAP)
- OrdersService (REST)
- Orchestration (BPEL)
- Async Integration (RabbitMQ)

### Security & Governance

Securing the services and establishing policies.



# Agenda Presentation Outline

Live Demonstration Plan

Challenges & Solutions

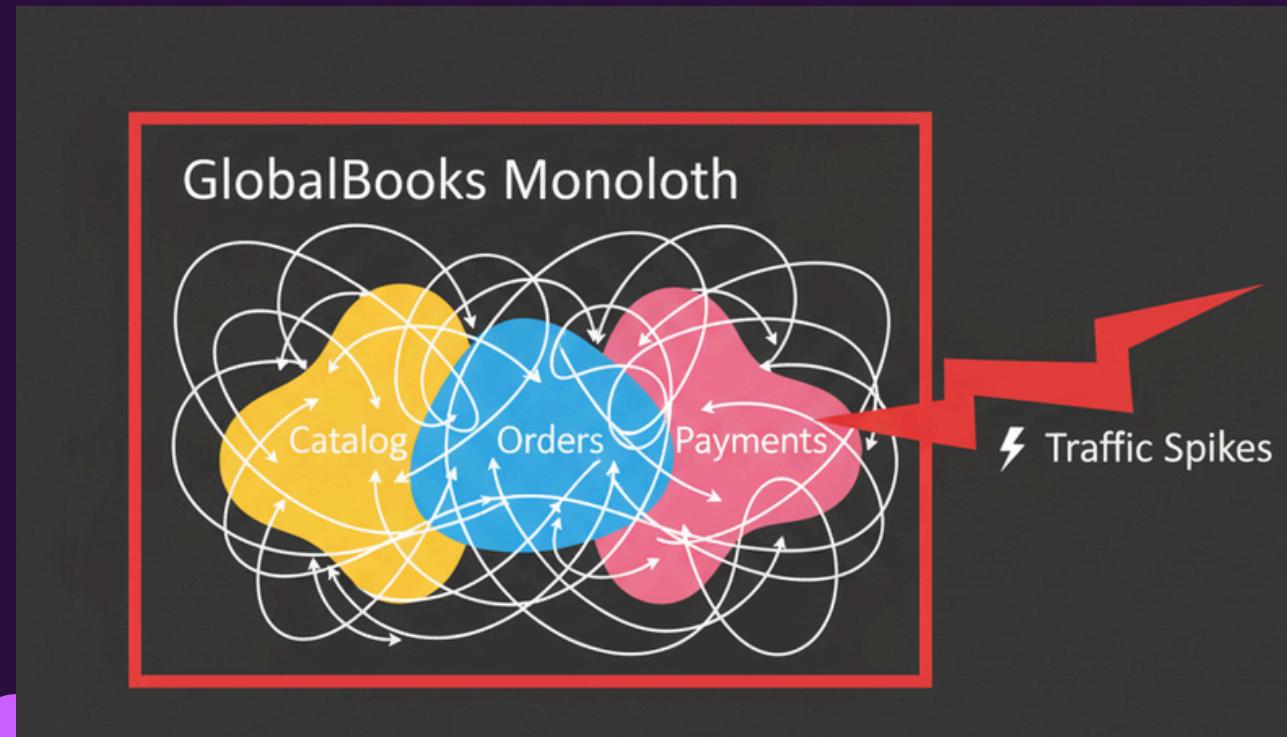
Conclusion & Future Work

Appendix: Project Deliverables



# The Problem

## Limitations of the GlobalBooks Monolithic System



### Poor Scalability

The entire application had to be scaled, even if only one feature was under heavy load during peak events. This led to system crashes.

### Low Agility

A tightly-coupled codebase meant that small changes required full system regression tests, making development slow and risky. Releasing new features took weeks.

### Technology Lock-in

The entire system was built on a single technology stack, making it difficult to adopt newer, more efficient technologies for specific tasks.

### High Maintenance Overhead

A single point of failure meant that a bug in one module could bring down the entire application.

# The Solution

## Applying SOA Principles

### Standardized Service Contract

Services communicate using well-defined, technology-agnostic contracts (WSDL for SOAP, JSON Schema for REST)

### Loose Coupling

Services are independent. The OrdersService doesn't need to know the inner workings of the PaymentsService. This is achieved through interfaces and a message broker

### Service Autonomy

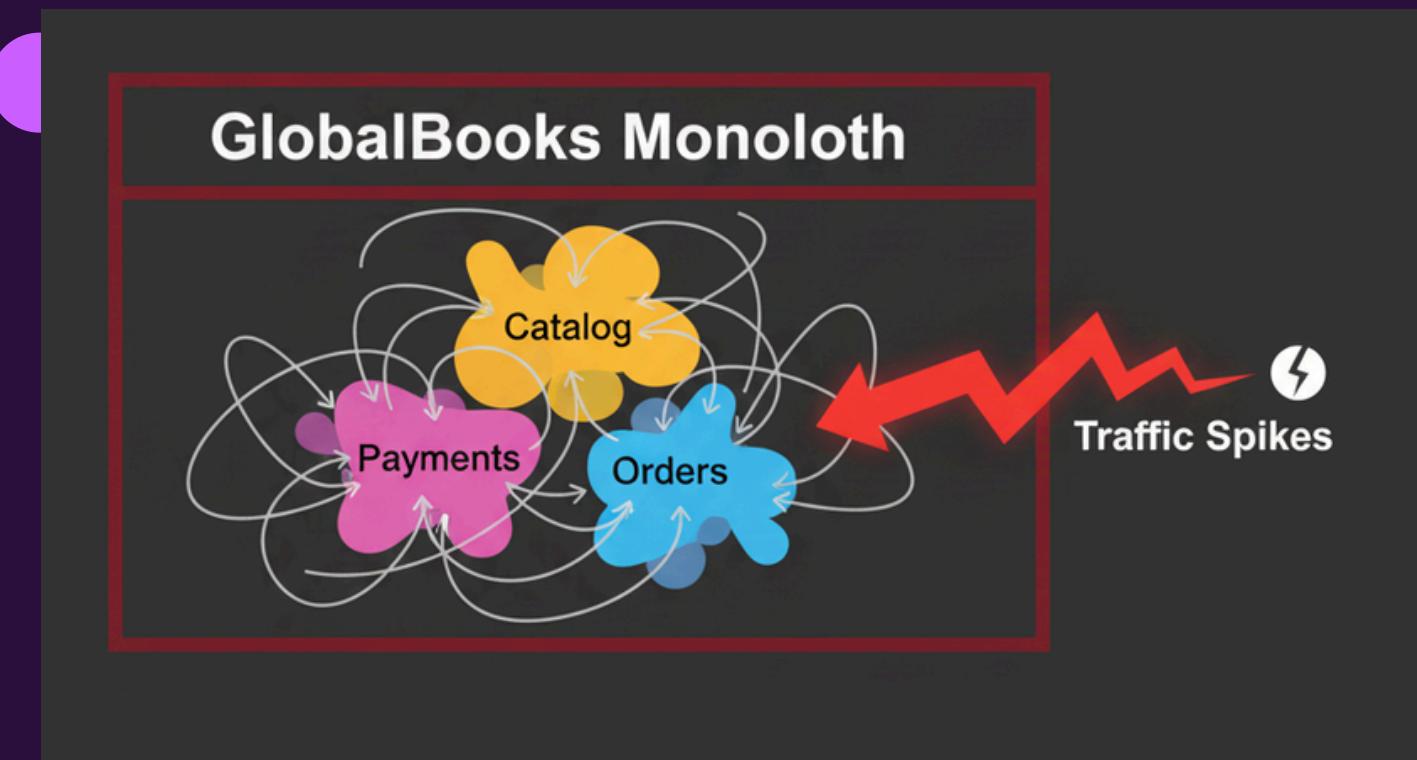
Each service manages its own logic and data. CatalogService controls product data, OrdersService controls order data.

### Service Composability & Reusability

Smaller services are orchestrated (using BPEL) to create larger, composite business processes like "PlaceOrder"

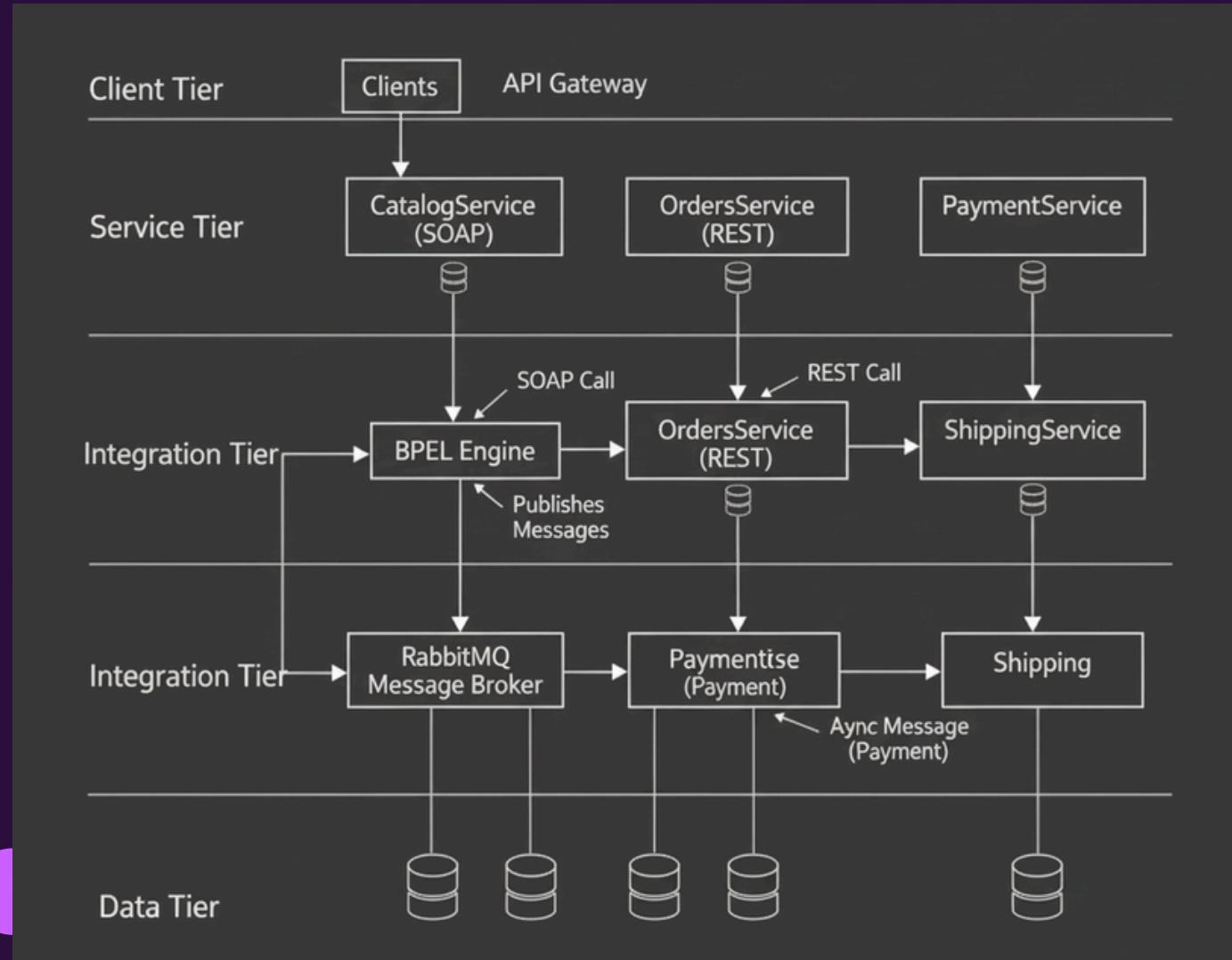
### Service Discoverability

A UDDI registry allows services to find each other dynamically, eliminating hard-coded dependencies



# Architectural Overview

## The New "GlobalBooks Inc." Ecosystem



### Service Tier

- **Synchronous Services:** CatalogService (SOAP) & OrdersService (REST) for real-time interactions.
- **Asynchronous Services:** PaymentsService & ShippingService for background processing.

### Integration Tier

- **Orchestration:** BPEL Engine manages the end-to-end PlaceOrder workflow.
- **Messaging:** RabbitMQ message broker for reliable, decoupled communication.

### Data Tier

- Each service has its own dedicated database, ensuring autonomy

# Service Deep Dive

## CatalogService (SOAP)



### Technology

Java (JAX-WS) running on Apache Tomcat

### Contract

Defined by a formal WSDL (CatalogService.wsdl)

- Operations: getBookDetails, searchBooks, addBook

### Configuration

- web.xml: Defines the servlet listener
- sun-jaxws.xml: Links the implementation class to the public endpoint

### Security

WS-Security to ensure only authorized systems can access it

### Discovery

Published in the Apache jUDDI private registry

The screenshot shows a Java-based SOA project structure in an IDE. The project is named 'GLOBALBOOKSSOA-MAIN'. The 'src' directory contains packages like 'com.globalbooks.payments' and 'com.globalbooks.config'. The 'tests' directory contains 'soapui\_catalog\_tests.xml' and 'orders\_postman\_collection.json'. The 'src/main/java/com/globalbooks/payments/config' package contains files like 'RedisConfig.java', 'PaymentConsumer.java', 'PaymentProducer.java', 'PaymentsApplication.java', and 'RootController.java'. The 'src/main/resources' directory contains 'application.properties' and 'logback-spring.xml'. The 'src/main/docker' directory contains 'Dockerfile', '.DCKpom.xml', and 'docker-compose.yml'. The 'src/main/cloud' directory contains 'cloud\_deployment.md'. The 'tests' directory also contains 'orders\_postman\_collection.json' and 'Coursework\_CCS3411\_SOA.pdf'. The 'src/main/test' directory contains 'soapui\_catalog\_tests.xml'. The 'src/main/test' directory is currently selected. The 'CatalogService.wsdl' file content is displayed in the main editor area, showing XML definitions for operations like 'getBookById' and 'getBooksByCategory'.

```
<?xml version="1.0" encoding="UTF-8"?>
<con:soapui-project name="CatalogService Tests" soapui-version="5.7.0"
                     xmlns:con="http://eviware.com/soapui/config">

    <!-- Task 6: SOAP UI Project for CatalogService Tests -->
    <!-- This project tests the SOAP endpoints with functional assertions.
         Viva Explanation: Demonstrates testing WS-* services; test cases validate operations,
         assertions check response structure/content. Use in SOAP UI for execution. -->

    <con:settings>
        <con:setting id="GlobalProperties">@Properties@</con:setting>
    </con:settings>

    <con:interface name="CatalogService" type="wsdl">
        <con:definition>http://localhost:8080/catalog?wsdl</con:definition>
        <con:operations>
            <con:operation name="getBookById" action="getBookById">
                <con:call name="getBookById"/>
            </con:operation>
            <con:operation name="getBooksByCategory" action="getBooksByCategory">
                <con:call name="getBooksByCategory"/>
            </con:operation>
        </con:operations>
    </con:interface>

    <con:testSuite name="CatalogService Test Suite">
        <con:testCase name="Test Get Book By ID">
            <con:testStep type="request" name="getBookById Request">
                <con:settings>
                    <con:setting id="Endpoint">http://localhost:8080/catalog</con:setting>
                </con:settings>
            </con:testStep>
        </con:testCase>
    </con:testSuite>
</con:soapui-project>
```

# Service Deep Dive

## OrdersService (REST)



### Technology

Spring Boot (Java)

### Interface

A pragmatic RESTful API

- Endpoints: POST /orders, GET /orders/{id}
- Data Format: JSON, with a defined JSON Schema for validation

### Security

Secured using OAuth2 (Client Credentials Flow). Clients must present a valid access token to interact with the API

### Key Role

Acts as the entry point for the PlaceOrder process and produces messages for the payment and shipping queues

The screenshot shows a dark-themed IDE interface with several tabs open at the top: pom.xml, application.properties, payments, logback-spring.xml, RabbitConfig.java, LCKpom.xml, and PaymentEv. The left side features an Explorer view with a tree structure of project files under 'GLOBALBOOKSSOA-MAIN'. The 'services' node has a 'catalog-soap' and 'orders-rest' child node, with 'orders-rest' being expanded to show its contents: 'src', 'target', 'classes', 'generated-sources', 'generated-test-sources', 'maven-archiver', 'maven-status', 'test-classes', 'orders-rest-1.0.0.jar', 'orders-rest-1.0.0.jar.original', 'Dockerfile', and 'pom.xml'. The right side is the 'OUTLINE' view, which displays the XML code of the 'pom.xml' file. The code includes details about the project's groupId (org.springframework.boot), artifactId (orders-rest), version (2.7.0), and packaging (jar). It also specifies the name (OrdersService REST API) and description (Spring Boot REST service for order management). The properties section sets the Java version to 11. The dependencies section includes a dependency on org.springframework.boot:spring-boot-starter-web. A note at the bottom of the pom.xml code indicates: '<!-- FIXFD: Added Jackson for JSON processing -->'.

```
<?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/POX/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.7.0</version>
    <relativePath/>
</parent>

<groupId>com.globalbooks</groupId>
<artifactId>orders-rest</artifactId>
<version>1.0.0</version>
<packaging>jar</packaging>

<name>OrdersService REST API</name>
<description>Spring Boot REST service for order management</description>

<properties>
    <java.version>11</java.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>

<!-- FIXFD: Added Jackson for JSON processing -->
```

# Service Deep Dive

## BPEL Orchestration



### Receive

The process starts when it receives a customer's order request.

### Invoke CatalogService (SOAP Call)

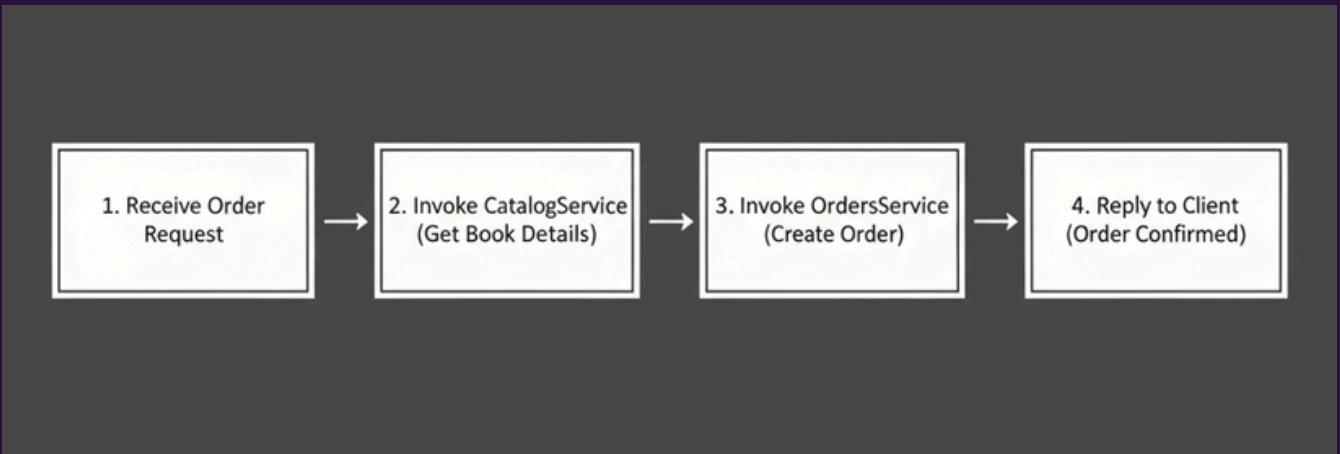
It synchronously calls the CatalogService to verify the book's price and stock availability

### Invoke OrdersService (REST Call)

If the book is available, it calls the OrdersService to create the order record

### Reply

It sends a confirmation back to the original client immediately. The process is now complete from the client's perspective. Payment and Shipping happen in the background



# Service Deep Dive

## Async Integration (RabbitMQ)

### Technology

RabbitMQ Message Broker

### Producer

The OrdersService, after successfully creating an order, publishes two messages

### Queues

- payments\_queue: Messages are consumed by the PaymentsService.
- shipping\_queue: Messages are consumed by the ShippingService.

### Benefit

The OrdersService doesn't wait for payment or shipping to complete. If the PaymentsService is down, messages will wait safely in the queue until it's back online. This makes the system resilient

### Reliability

Implemented persistent messages to ensure no data is lost even if the broker restarts

# Challenges & Solutions

## Key Technical Hurdles and How They Were Overcome

### Distributed Data Consistency

- Problem: How to handle a payment failure after an order has already been created?
- Solution: Implemented a basic compensation logic. If the PaymentsService fails, it could publish a message to an order\_cancellation\_queue to trigger a rollback

### Complex Testing Environment

- Problem: Testing the end-to-end flow required all services and infrastructure (Tomcat, RabbitMQ, BPEL Engine) to be running simultaneously
- Solution: Created a step-by-step startup script and used tools like SOAP UI and Postman to build a repeatable test suite

### Security Configuration

- Problem: Configuring both WS-Security (XML-based) and OAuth2 (Token-based) correctly was complex
- Solution: Isolated and tested each security mechanism independently before integrating them into the main BPEL flow



# Conclusion & Future Work

## Project Summary & Next Steps

### Successfully Achieved

- Decomposed the monolith into four independent, loosely coupled services.
- Implemented a multi-protocol architecture (SOAP & REST).
- Created a resilient system using asynchronous messaging.
- Secured the architecture using industry-standard protocols.

### Future Improvements

- API Gateway: Introduce a single entry point for all client applications to simplify client logic and centralize cross-cutting concerns like rate limiting.
- Containerization: Package each service into a Docker container and manage them with Docker Compose or Kubernetes for easier deployment and scaling.
- Distributed Tracing: Implement tools like Jaeger or Zipkin to trace requests as they travel across multiple services, making debugging easier.





**GLOBALBOOKS INC.**

SOA REFACTORING PROJECT



# THANK YOU

PRESENTATION BY 22UG1-0472 ( K.G. PASINDU KAVISHKA )