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

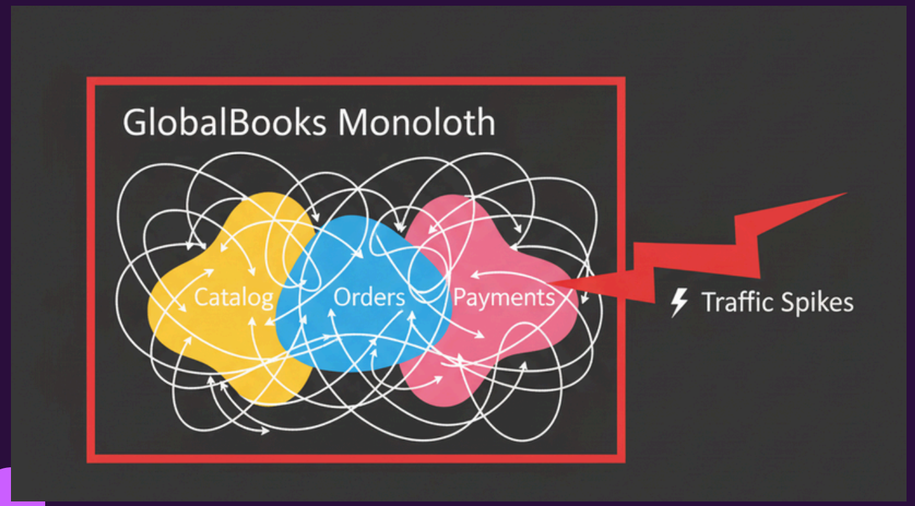
Challenges & Solutions

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

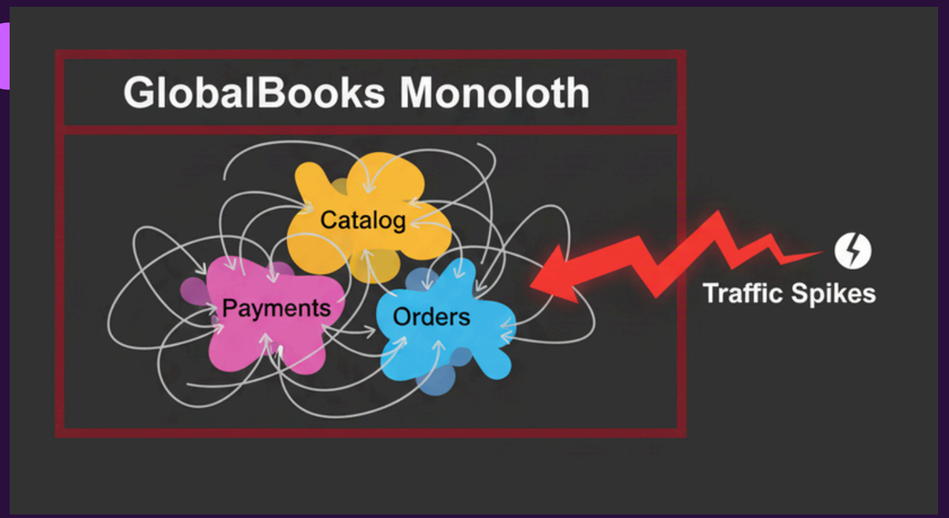
Each service manages its own logic and data. CatalogService controls product data, OrdersService controls order data.

Service Composability & Reusability

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

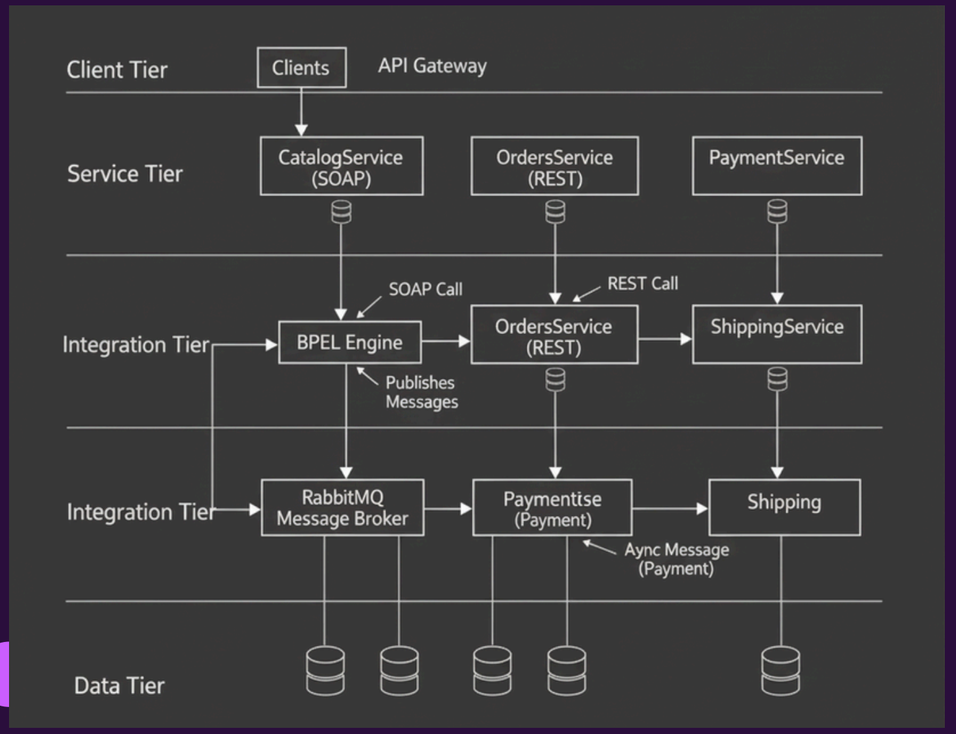
Service Discoverability

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

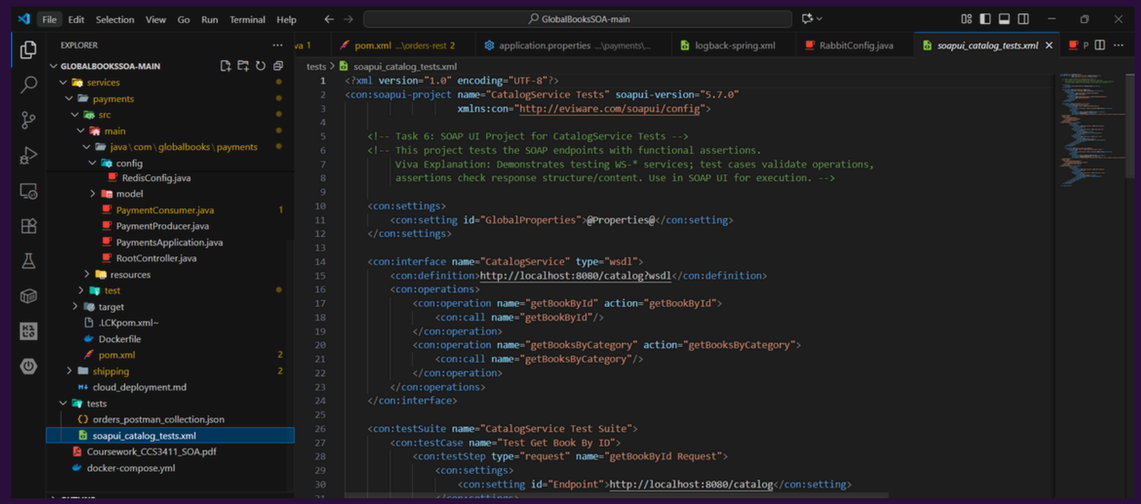
- web.xml: Defines the servlet listener
- sun-jaxws.xml: Links the implementation class to the public endpoint

Security

WS-Security to ensure only authorized systems can access it

Discovery

Published in the Apache jUDDI private registry



Service Deep Dive

OrdersService (REST)

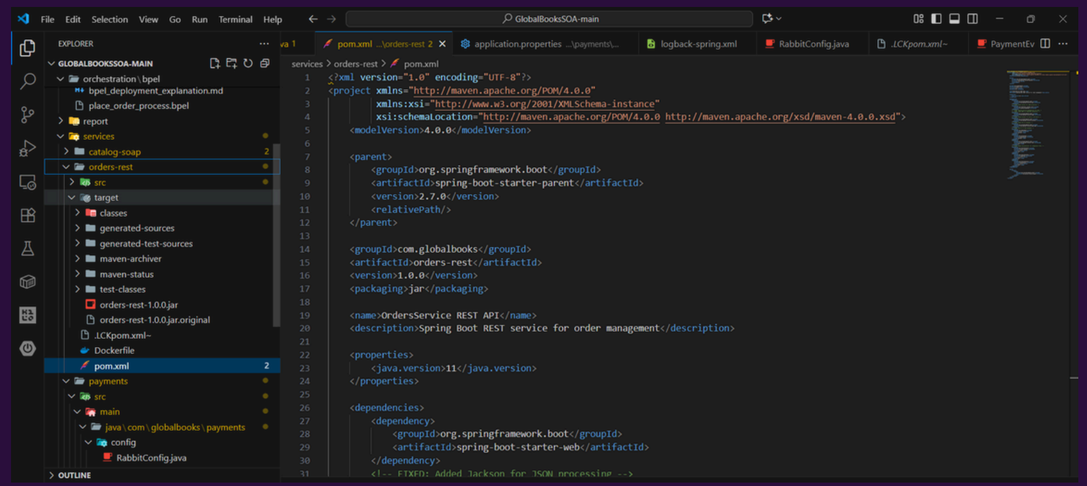
Technology

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

Service Deep Dive

BPEL Orchestration

Receive

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

Invoke CatalogService (SOAP Call)

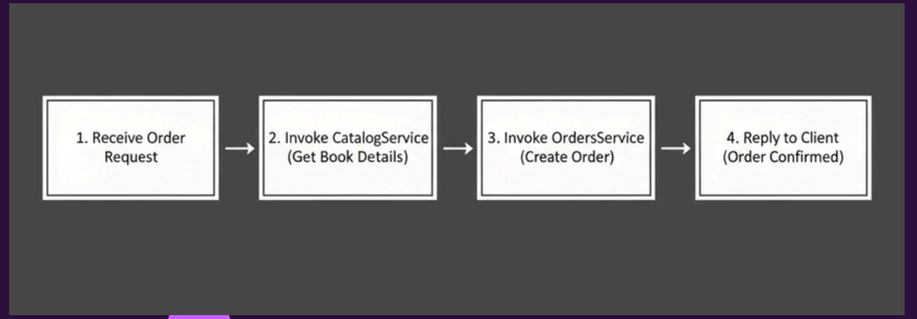
It synchronously calls the CatalogService to verify the book's price and stock availability

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)