GlobalBooks SOA System

Reflective Report

CCS3341 SOA & Microservices Coursework

Name: M C R Mallawaarachchi
Index: 22ug1-0093
Module: CCS3341 SOA & Microservices
Date: September 3, 2025

1. Introduction

Working on the GlobalBooks SOA system has been an eye-opening experience that brought theoretical concepts to life. This project involved building a complete microservices-based online bookstore using six interconnected services. What started as a coursework assignment became a deep dive into real-world distributed system challenges and solutions.

The system I built demonstrates SOA principles through practical implementation - something that textbooks can only describe. Each service runs independently, communicates through well-defined interfaces, and contributes to a larger business goal. It's been rewarding to see how abstract architectural concepts translate into working code.

2. Technical Implementation Reflections

2.1. Architecture Design Decisions

Deciding to split the system into six services felt overwhelming at first, but it made perfect sense as development progressed. Each service (auth-server, catalog-service, orders-service, payments-service, shipping-service, and order-orchestration-service) has its own responsibility and runs on separate ports (8081-8086).

What worked really well:

- Clean separation made debugging much easier
- I could work on different services independently
- The mix of SOAP and REST showed me when to use each approach
- JWT authentication kept everything secure without being complicated

Looking back, I wish I had:

- Added service discovery from the start (manually managing ports got tedious)
- Implemented circuit breakers for better fault handling
- Used centralized configuration management

2.2. Protocol Implementation Experience

2.2.1. SOAP Web Services

Working with SOAP through Spring Web Services was initially challenging but incredibly educational. The catalog-service's contract-first approach taught me the value of defining clear service contracts upfront. Watching Spring automatically generate WSDL from my XSD schema felt like magic.

Implementing WS-Security with Username Token authentication added enterprise-grade security to SOAP endpoints. Learning to work with WSS4J2 interceptors and SOAP security headers deepened my understanding of web service security standards.

The biggest lesson: SOAP's verbosity isn't just overhead - it's documentation, validation, and security built into the service contract. When integration issues arose, having well-defined schemas and security policies saved hours of debugging.

2.2.2. REST API Development

Building REST APIs for the orders-service felt more natural and intuitive. The HTTP methods map nicely to business operations, and JSON responses are much easier to work with than XML.

What I learned about good REST design:

- Proper HTTP status codes make a huge difference for API consumers
- Consistent naming conventions reduce confusion
- Error responses need to be as thoughtful as success responses

2.3. Message Queue Architecture

Setting up RabbitMQ with five queues (order, payment, shipping, paymentconfirm, shippingconfirm) was probably the most satisfying part of the project. Seeing messages flow asynchronously between services made the system feel alive and responsive.

The queue architecture solved a major problem: what happens when one service is slow or temporarily unavailable? Instead of everything grinding to a halt, messages queue up and get processed when services are ready. This loose coupling makes the system much more resilient than I initially expected.

2.4. Security Implementation

Implementing a dual authentication model was one of the most challenging aspects of this project. The system uses JWT authentication for REST endpoints and WS-Security for SOAP services, creating a comprehensive security architecture.

JWT Authentication: Using HS256 with 10-hour token expiration and BCrypt for passwords creates a solid security foundation for REST services. The stateless nature of JWT tokens means services can validate tokens independently without requiring shared session storage.

WS-Security Implementation: Adding WS-Security Username Token authentication to SOAP endpoints required integrating WSS4J2 interceptors with Spring Web Services. This implementation demonstrates enterprise-grade SOAP security standards while maintaining the clean separation between REST and SOAP authentication models.

The dual approach shows how different protocols can coexist with appropriate security measures - JWT for modern REST APIs and WS-Security for enterprise SOAP integrations.

3. Development Process Insights

3.1. Technology Stack Evaluation

Spring Boot 2.7.17 was an excellent choice for this project. The auto-configuration saved countless hours of setup, and the ecosystem integration (Spring Security, Spring Web Services, Spring Integration) made complex tasks manageable.

The learning curve was steep initially, but once I understood the Spring way of doing things, development became much faster. The extensive documentation and community support made problem-solving much easier.

3.2. Testing and Validation

Creating comprehensive test suites for user creation, REST orders, SOAP orders, and catalog service functionality gave me confidence that the system actually works. There's something satisfying about running automated tests and seeing everything pass.

The tests also caught integration issues early - problems that would have been much harder to debug in a fully deployed system.

4. Challenges and Solutions

4.1. What Kept Me Up at Night

Service Coordination: Getting six services to start up in the right order and talk to each other properly was harder than expected. I spent way too much time troubleshooting "connection refused" errors before realizing I needed better dependency management.

Message Flow Design: My first attempt at the RabbitMQ setup was overly complicated. It took three redesigns to get the five-queue architecture that actually works efficiently.

Protocol Integration: Deciding when to use SOAP vs REST wasn't always obvious. I learned that SOAP works better for formal, contract-driven integrations, while REST is perfect for simple, resource-oriented operations.

4.2. How I Solved Them

Docker Compose became my best friend for managing service dependencies and startup order. Externalizing configuration through properties files made environment management much easier. Comprehensive logging across all services was a lifesaver for debugging distributed system issues.

5. What I Actually Learned

5.1. Technical Skills That Stuck

This project taught me SOA isn't just about splitting up a monolith - it's about designing systems that can evolve and scale independently. I now understand why companies like Netflix and Amazon invest so heavily in microservices architecture.

Working with Spring Boot's ecosystem gave me confidence in enterprise Java development. The security implementation with JWT taught me practical authentication patterns I'll definitely use again.

The message queue architecture opened my eyes to event-driven design. Asynchronous processing isn't just a performance optimization - it's a completely different way of thinking about system interactions.

5.2. Beyond the Code

This project improved my system design thinking significantly. I learned to break down complex problems into manageable services and design clear interfaces between them.

Debugging distributed systems taught me patience and systematic problem-solving. When something breaks across multiple services, you need a methodical approach to track down the root cause.

Writing comprehensive documentation made me realize how important clear communication is in software development. Future me (and anyone else working on this code) will thank past me for the detailed documentation.

6. If I Were to Do This Again

6.1. Quick Wins

I'd definitely add distributed tracing with Zipkin from the start - tracking requests across multiple services was a pain without it. Centralized logging with the ELK stack would save hours of debugging time.

Circuit breakers with Hystrix should be a day-one decision, not an afterthought. They're essential for building resilient distributed systems.

6.2. Bigger Picture

For production deployment, I'd design this system with Kubernetes in mind from the beginning. Container orchestration becomes critical when you have multiple services to manage.

A service mesh like Istio would handle a lot of the cross-cutting concerns (security, monitoring, traffic management) that I implemented manually.

7. Final Thoughts

Building the GlobalBooks SOA system was challenging, frustrating, and ultimately incredibly rewarding. It's one thing to read about microservices architecture in textbooks - it's completely different to actually build one and deal with all the real-world complications.

The project gave me genuine appreciation for the complexity of distributed systems and the engineering discipline required to build them properly. Every design decision has trade-offs, and understanding those trade-offs only comes through hands-on experience.

Most importantly, this project taught me that SOA isn't just a technical architecture pattern - it's a way of organizing both code and teams to build systems that can evolve over time. That's a lesson that extends far beyond this coursework.

The six services I built, the protocols I implemented, the message queues I configured, and even the bugs I fixed all contributed to a deeper understanding of how modern software systems actually work. That's knowledge I'll carry forward into my career as a software developer.