1. GlobalBooks SOA Design Document

This report details the architectural decisions, implementation specifics, and operational considerations for decomposing the GlobalBooks monolith into a Service-Oriented Architecture (SOA).

1.1. System Architecture Sketch

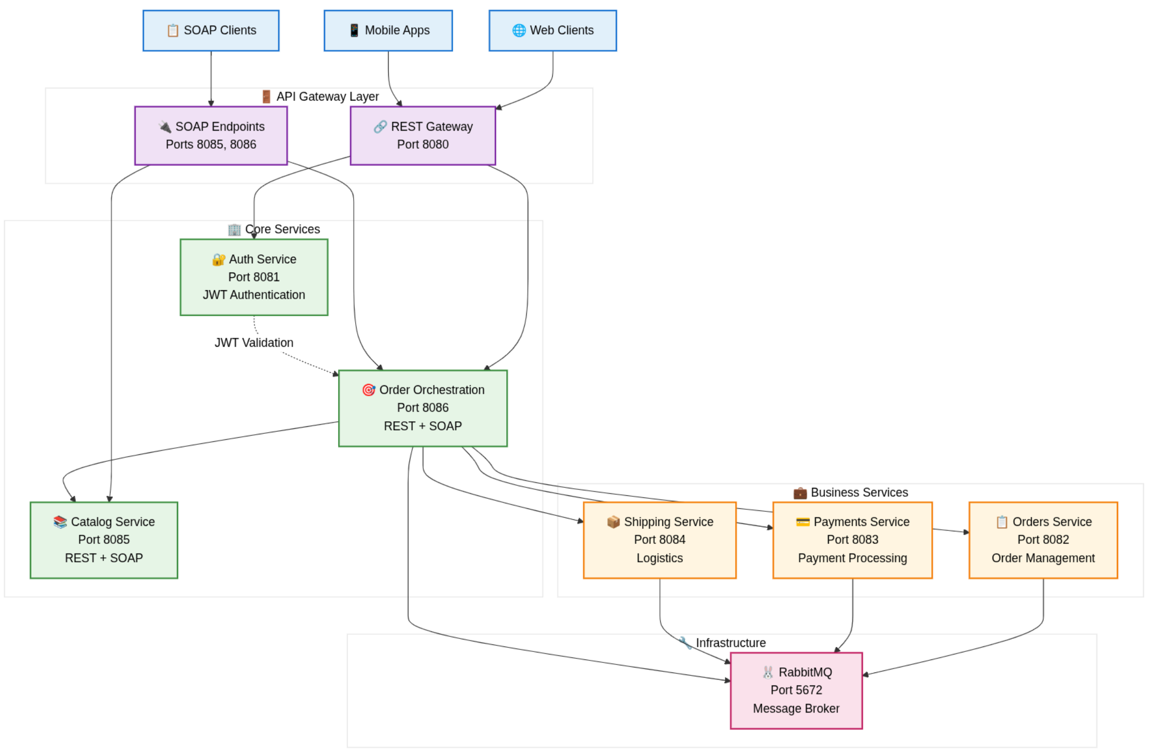
📋 SOAP Clients
📱 Mobile Apps
🌐 Web Clients
🚪 API Gateway Layer
🔌 SOAP Endpoints Ports 8085, 8086
🔗 REST Gateway Port 8080
🏢 Core Services
🔐 Auth Service Port 8081 JWT Authentication
JWT Validation
🎯 Order Orchestration Port 8086 REST + SOAP
📚 Catalog Service Port 8085 REST + SOAP
💼 Business Services
📦 Shipping Service Port 8084 Logistics
💳 Payments Service Port 8083 Payment Processing
📋 Orders Service Port 8082 Order Management
🔧 Infrastructure
🐰 RabbitMQ Port 5672 Message Broker

Figure 1: GlobalBooks SOA Architecture Overview

1.2. SOA Design Principles

When decomposing the monolith into independent services, we applied several key SOA design principles:

- Service Loose Coupling: Services are designed to be independent of each other, minimizing direct dependencies. Changes in one service have minimal impact on others, allowing for independent development and deployment. For instance, the `OrdersService` interacts with `CatalogService` via well-defined interfaces (SOAP/REST) rather than direct code dependencies.
- Service Autonomy: Each service owns its data and logic, operating independently. This is evident in services like `PaymentsService` and `ShippingService`, which manage their respective domains without direct reliance on the internal workings of other services.
- Service Reusability: Services are designed to be generic and reusable across different business processes. The `CatalogService`, for example, provides book information that can be consumed by various applications, not just the order placement process.

- Service Discoverability: Services are designed to be easily discoverable by potential consumers. This is achieved through mechanisms like WSDL for SOAP services and API documentation for REST services.
- Service Composability: Services can be combined to form more complex business processes. The order orchestration process exemplifies this by coordinating interactions between `CatalogService`, `OrdersService`, `PaymentsService`, and `ShippingService`.
- Service Statelessness: Where possible, services are designed to be stateless, meaning they do not retain client-specific information between requests. This improves scalability and resilience.

1.3. Benefits and Challenges

Key Benefit: One of the primary benefits of our SOA approach is enhanced scalability and flexibility. Each service can be developed, deployed, and scaled independently based on its specific load requirements. For example, if the `CatalogService` experiences high traffic, it can be scaled horizontally without affecting the performance of the `PaymentsService` or `ShippingService`. This allows for more efficient resource utilization and better responsiveness to varying demands.

Primary Challenge: A significant challenge encountered is the increased operational complexity. Managing a distributed system with multiple independent services introduces complexities in terms of deployment, monitoring, logging, tracing, and debugging. Inter-service communication, network latency, and distributed transactions require careful consideration and robust error handling mechanisms, which are more intricate than in a monolithic application.

1.4. Service Portfolio

Service	Port	Protocols	Primary Function
REST Gateway	8080	REST	API gateway and routing
Auth Server	8081	REST	JWT authentication
Orders Service	8082	REST	Order management
Payments Service	8083	REST	Payment processing
Shipping Service	8084	REST	Logistics coordination
Catalog Service	8085	REST + SOAP	Product catalog with SOAP interface
Order Orchestration	8086	REST	Business process coordination

Table 1: Service Portfolio Overview

1.5. Technology Stack Implementation

Framework and Libraries:
- Spring Boot 2.7+ for microservice development
- Apache CXF for SOAP web service implementation
- Spring Security with JWT for authentication
- RabbitMQ for asynchronous messaging
- H2 Database for development (configurable for production)
- Maven for build management and multi-module structure

Security Implementation: The system implements centralized authentication through the Auth Service using JWT tokens. Clients authenticate with username/password, receive JWT tokens, and include them in subsequent requests. All services validate tokens independently using a shared secret.

1.6. CatalogService SOAP Implementation

The `CatalogService` exposes a SOAP interface using Spring WS framework, implementing the actual GlobalBooks catalog operations.

SOAP Endpoint Configuration:

```
@Endpoint
@Component
public class BookEndpoint {

    private static final String NAMESPACE_URI =
        "http://globalbooks.com/catalog";

    @Autowired
    private BookService bookService;

    @PayloadRoot(namespace = NAMESPACE_URI,
                 localPart = "getBookDetailsRequest")
    @ResponsePayload
    public Element getBookDetails(@RequestPayload Element request)
            throws Exception {

        String bookId = getTextContent(request, "bookId");
        Book book = bookService.getBookById(bookId);

        // Create SOAP response XML element
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        factory.setNamespaceAware(true);
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document doc = builder.newDocument();

        Element response = doc.createElementNS(NAMESPACE_URI,
                                               "getBookDetailsResponse");

        if (book != null) {
            Element bookElement = doc.createElement("book");
            Element titleElement = doc.createElement("title");
            titleElement.setTextContent(book.getTitle());
            bookElement.appendChild(titleElement);
            response.appendChild(bookElement);
        }

        return response;
    }
}
```

1.7. OrdersService REST API Design

The `OrdersService` provides RESTful endpoints for order management with actual implementation details from the GlobalBooks system.

Controller Implementation:

```
@RestController
@RequestMapping("/orders")
public class OrderController {
    private final OrderRepository orderRepository;

    public OrderController(OrderRepository orderRepository) {
        this.orderRepository = orderRepository;
    }

    @PostMapping
    public ResponseEntity<Order> createOrder(@RequestBody Order order) {
        Order savedOrder = orderRepository.save(order);
        return ResponseEntity.ok(savedOrder);
    }

    @GetMapping("/{id}")
    public ResponseEntity<Order> getOrderById(@PathVariable Long id) {
        Order order = orderRepository.findById(id);
        if (order == null) {
            return ResponseEntity.notFound().build();
        }
        return ResponseEntity.ok(order);
    }

    @GetMapping
    public ResponseEntity<List<Order>> getAllOrders() {
        return ResponseEntity.ok(orderRepository.findAll());
    }
}
```

Order Model Structure:

```json
{
  "id": null,
  "bookIsbns": ["string"],
  "customerId": "string",
  "bookDetails": {
    "title": "string",
    "author": "string",
    "quantity": 1
  }
}
```

1.8. Order Orchestration SOAP Service

The Order Orchestration Service provides SOAP endpoints for enterprise integration, implementing the ProcessOrder operation for external systems.

SOAP Endpoint Implementation:

```
@Endpoint
@Component
public class OrderSoapEndpoint {

    private static final String NAMESPACE_URI = "http://globalbooks.com/orders";

    @Autowired
    private OrderOrchestrationController orderController;

    @PayloadRoot(namespace = NAMESPACE_URI, localPart = "ProcessOrderRequest")
    @ResponsePayload
    public Element processOrder(@RequestPayload Element request) throws Exception {

        // Extract values from SOAP request
        String customerId = getTextContent(request, "customerId");
        String bookId = getTextContent(request, "bookId");
        String quantityStr = getTextContent(request, "quantity");

        // Create order request map
        Map<String, Object> orderRequest = new HashMap<>();
        orderRequest.put("customerId", customerId);
        orderRequest.put("bookId", bookId);
        orderRequest.put("quantity", Integer.parseInt(quantityStr));

        // Process order and create SOAP response
        Map<String, Object> result = orderController.processOrder(
            orderRequest, "Bearer SOAP-CLIENT-TOKEN");

        return createResponse(result);
    }
}
```

1.9. Message Queue Integration

RabbitMQ facilitates asynchronous communication between services with actual queue configurations implemented in the GlobalBooks system.

Actual Queue Definitions:

- `order.queue`: Order processing and creation
- `payment.queue`: Payment processing requests
- `shipping.queue`: Shipping coordination messages
- `paymentconfirm.queue`: Payment confirmation events
- `shippingconfirm.queue`: Shipping confirmation events

Queue Configuration Implementation:

```
@Configuration
@EnableRabbit
public class RabbitConfig {

    @Bean
    public Queue orderQueue() {
        return new Queue("order.queue", true);
    }

    @Bean
    public Queue paymentQueue() {
        return new Queue("payment.queue", true);
    }

    @Bean
    public Queue shippingQueue() {
        return new Queue("shipping.queue", true);
    }

    @Bean
    public Queue paymentConfirmQueue() {
        return new Queue("paymentconfirm.queue", true);
    }

    @Bean
    public Queue shippingConfirmQueue() {
        return new Queue("shippingconfirm.queue", true);
    }
}
```

Message Listener Implementation:

```
@Service
public class OrderQueueProcessor {

    @RabbitListener(queues = "order.queue")
    public void processOrderFromQueue(Map<String, Object> orderData) {
        // Order processing logic
        Order order = new Order();
        order.setCustomerId((String) orderData.get("userId"));
        Order savedOrder = orderRepository.save(order);

        // Send to payment queue
        amqpTemplate.convertAndSend("payment.queue", paymentMessage);
    }
}
```

1.10. Order Processing Workflow

The actual order fulfillment process in the GlobalBooks system follows this implementation:

1. Order Submission: Client sends POST request to `/api/orders/process` with JWT token
2. Order Controller: OrderOrchestrationController receives request and sends to Spring Integration flow
3. Catalog Enrichment: CatalogService checks book availability via SOAP and enriches order data
4. Queue Processing: Enriched order sent to `order.queue` for asynchronous processing
5. Order Creation: OrderQueueProcessor creates order record in database
6. Payment Processing: Order data sent to `payment.queue` for payment handling
7. Shipping Coordination: After payment confirmation, shipping data sent to `shipping.queue`
8. Status Updates: Confirmation messages flow back through `paymentconfirm.queue` and `shippingconfirm.queue`

Actual Controller Implementation:

```
@RestController
@RequestMapping("/api/orders")
public class OrderOrchestrationController {

    @PostMapping("/process")
    public Map<String, Object> processOrder(
            @RequestBody Map<String, Object> orderRequest,
            @RequestHeader("Authorization") String token) {

        // Send to Spring Integration flow
        orderInputChannel.send(MessageBuilder
            .withPayload(orderRequest)
            .setHeader("Authorization", token)
            .build());
        // Return immediate response
        Map<String, Object> response = new HashMap<>();
        response.put("status", "success");
        response.put("message", "Order submitted for processing");
        return response;
    }
}
```

1.11. Error Handling and Resilience

Retry Mechanisms: Failed messages are retried with exponential backoff before being routed to Dead Letter Queues (DLQs) for manual inspection and resolution.

Circuit Breaker Pattern: Services implement circuit breakers to handle downstream failures gracefully, providing fallback responses when dependencies are unavailable.

Health Monitoring: Each service exposes health endpoints for operational monitoring, including database connectivity, message broker status, and dependency health checks.

1.12. Testing Strategy

Unit Testing: Individual service components tested in isolation with comprehensive mock implementations for external dependencies.

Integration Testing: End-to-end testing of service interactions, including REST API calls, SOAP service invocations, and message queue communication.

Contract Testing: SOAP services validated against WSDL contracts, and REST APIs tested against OpenAPI specifications to ensure compliance.

1.13. Security Implementation

Authentication: JWT-based authentication with configurable token expiration and refresh mechanisms.

Authorization: Role-based access control implemented at the service level with scope validation for different operations.

Transport Security: HTTPS/TLS configuration for production deployments with proper certificate management.

1.14. Deployment and Operations

Containerization: Services packaged as Docker containers with optimized layering for efficient deployment and scaling.

Configuration Management: Environment-specific configurations externalized through application properties and environment variables.

Monitoring and Observability: Comprehensive logging, metrics collection, and distributed tracing capabilities for operational visibility.

1.15. Conclusion

The GlobalBooks SOA implementation successfully demonstrates enterprise-grade microservices architecture with robust service decomposition, comprehensive security, and reliable message-driven communication. The solution provides a scalable foundation for online bookstore operations while maintaining compatibility with both modern REST clients and legacy SOAP-based enterprise systems.

Key Achievements:
- Successful microservices decomposition based on business capabilities
- Dual protocol support for diverse integration requirements
- Robust security implementation with JWT authentication
- Comprehensive testing framework covering all service types
- Message-driven architecture with quality of service guarantees

1.16. Service Autonomy Design

Each service demonstrates complete autonomy through:

1.16.1. Data Autonomy

- Independent data stores per service
- No direct database sharing between services
- Service-specific schemas and data models
- Transactional boundaries within service scope

1.16.2. Deployment Autonomy

- Independent deployment pipelines
- Service-specific configuration management
- Isolated runtime environments
- Zero-dependency deployment capability

1.16.3. Technology Autonomy

- Service-specific technology stack choices
- Independent framework and library selections
- Optimal tool selection per business capability
- Evolution path independence

1.17. Contract-First Design

1.17.1. SOAP Services Implementation

The system implements contract-first SOAP development:

```
<!-- Catalog Service WSDL Structure -->
<definitions targetNamespace="http://catalog.globalbooks.com/">
  <types>
    <xsd:schema targetNamespace="http://catalog.globalbooks.com/">
      <xsd:element name="getBookRequest">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="isbn" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="getBookResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="book" type="tns:Book"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </types>
</definitions>
```

1.17.2. REST API Design

RESTful services follow OpenAPI specifications:

```
{
  "openapi": "3.0.1",
  "paths": {
    "/orders": {
      "post": {
        "summary": "Create new order",
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "$ref": "#/components/schemas/OrderRequest"
              }
            }
          }
        }
      }
    }
  }
}
```

2. Integration Architecture

2.1. Message-Driven Integration

The system implements asynchronous, event-driven communication using RabbitMQ:

2.1.1. Exchange and Queue Configuration

```
exchanges:
  orders.exchange:
    type: topic
    durable: true
    routing-patterns:
      - "order.created"
      - "order.payment.*"
      - "order.shipping.*"

queues:
  payment.processing.queue:
    exchange: orders.exchange
    routing-key: "order.payment.process"
    dead-letter-exchange: dlx.orders

  shipping.notification.queue:
    exchange: orders.exchange
    routing-key: "order.shipping.notify"
    durable: true
```

2.1.2. Event-Driven Workflows

1. Order Creation: Orders service publishes order.created event
2. Payment Processing: Payments service consumes payment-related events
3. Shipping Coordination: Shipping service handles fulfillment events
4. Status Updates: Orchestration service coordinates cross-service workflows

2.2. Service Orchestration vs Choreography

The architecture implements a hybrid approach:

2.2.1. Orchestration (Order Processing)

- Centralized business process management in Order Orchestration Service
- Step-by-step workflow coordination
- Compensating transaction handling
- Complex business rule enforcement

2.2.2. Choreography (Event Broadcasting)

- Loosely coupled event publishing and consumption
- Autonomous service reactions to business events
- Parallel processing capabilities
- Resilient failure handling

3. Security Architecture

3.1. Authentication and Authorization

3.1.1. JWT-Based Security for REST Services

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public JwtAuthenticationFilter jwtAuthenticationFilter() {
        return new JwtAuthenticationFilter(jwtUtil());
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

3.1.2. WS-Security for SOAP Services

SOAP endpoints implement WS-Security standards:

- UsernameToken authentication
- X.509 certificate validation
- Message-level security
- SOAP header security tokens

3.2. Security Policies

3.2.1. Authentication Flow

1. User credentials validated against Auth Service
2. JWT token issued with expiration and claims
3. Token validated on each protected endpoint access
4. Token refresh mechanism for session management

3.2.2. Authorization Model

- Role-based access control (RBAC)
- Service-to-service authentication via JWT
- API key management for external integrations
- Rate limiting and request throttling

4. Technology Stack and Implementation

4.1. Core Framework Stack

Technology	Version	Purpose
Spring Boot	2.7.17	Application framework, auto-configuration, embedded containers
Spring WS	3.1.4	Contract-first SOAP web services development
Spring Integration	5.5.15	Enterprise integration patterns, message routing
Spring Security	5.7.5	Authentication, authorization, JWT token management
RabbitMQ	3.9	Message broker, asynchronous communication
Maven	3.8+	Multi-module project management, dependency resolution
Docker	20.0+	Containerization, infrastructure as code

Table 2: Technology Stack - Framework and Tool Selection

4.2. Design Patterns Implementation

4.2.1. Enterprise Integration Patterns

- Message Router: RabbitMQ exchange routing
- Message Translator: Protocol conversion between REST/SOAP
- Content-Based Router: Order type-specific processing
- Dead Letter Channel: Failed message handling

4.2.2. Microservices Patterns

- API Gateway: Centralized routing and cross-cutting concerns
- Service Registry: Service discovery and health monitoring
- Circuit Breaker: Fault tolerance and resilience
- Bulkhead: Resource isolation and failure containment

5. Quality Attributes

5.1. Performance Characteristics

Metric	Target	Measurement Method
Response Time	< 200ms	95th percentile for standard operations
Throughput	1000+ req/sec	Concurrent request handling per service
Availability	99.5%	Uptime monitoring and health checks
Scalability	Horizontal	Stateless design enabling load distribution
Memory Usage	150-200MB	Per service JVM heap allocation
Startup Time	30-45 sec	Complete service ecosystem initialization

Table 3: Performance Targets and Quality Metrics

5.2. Scalability and Resilience

5.2.1. Horizontal Scaling Design

- Stateless service implementation
- Load balancer compatibility
- Database connection pooling
- Shared-nothing architecture

5.2.2. Fault Tolerance Mechanisms

- Service health monitoring and automatic restart
- Circuit breaker pattern for downstream dependencies
- Message persistence and delivery guarantees
- Graceful degradation during partial system failures

6. Testing Strategy

6.1. Comprehensive Testing Approach

6.1.1. Functional Testing

- Unit tests for individual service components
- Integration tests for inter-service communication
- Contract testing for API compatibility
- End-to-end workflow validation

6.1.2. Test Automation Suite

```
# Automated test execution
./tests/test-1-user-creation.sh      # Authentication workflow
./tests/test-2-rest-order.sh         # REST API integration
./tests/test-3-soap-order.sh         # SOAP service validation
./tests/run-all-tests.sh             # Complete test suite
```

6.1.3. Performance Testing

- Load testing for throughput validation
- Stress testing for breaking point identification

- Endurance testing for memory leak detection
- Volume testing for large dataset handling

7. Deployment and Operations

7.1. Deployment Architecture

7.1.1. Local Development Environment

```
# Infrastructure startup
docker-compose up -d rabbitmq

# Service deployment sequence
mvn spring-boot:run -pl auth-server -Dspring-boot.run.arguments=--server.port=8081
mvn spring-boot:run -pl catalog-service -Dspring-boot.run.arguments=--
server.port=8085
mvn spring-boot:run -pl order-orchestration-service -Dspring-boot.run.arguments=--
server.port=8086
```

7.1.2. Production Deployment Considerations

- Container orchestration with Docker Compose or Kubernetes
- Service discovery and load balancing
- Configuration externalization
- Centralized logging and monitoring
- Blue-green deployment strategies

7.2. Monitoring and Observability

7.2.1. Health Monitoring

- Service health endpoints (`/actuator/health`)
- Database connectivity validation
- Message broker connection monitoring
- Custom business metric collection

7.2.2. Distributed Tracing

- Correlation ID propagation across service calls
- Request/response logging and analysis
- Performance bottleneck identification
- Error propagation tracking

8. Governance and Standards

8.1. Service Governance Framework

8.1.1. Versioning Strategy

- Semantic versioning for backward compatibility
- URL-based versioning for REST APIs
- Namespace versioning for SOAP services
- Deprecation notice and sunset procedures

8.1.2. Quality Standards

- Code coverage requirements (80% minimum)
- API documentation completeness
- Security vulnerability scanning
- Performance benchmark compliance

8.2. Compliance and Risk Management

8.2.1. Data Protection

- PII handling and encryption standards
- Audit trail requirements
- Data retention policies
- Cross-border data transfer compliance

8.2.2. Operational Risk Mitigation

- Disaster recovery procedures
- Business continuity planning
- Security incident response
- Change management processes

9. Future Enhancements

9.1. Planned Technology Evolution

9.1.1. Cloud-Native Enhancements

- Kubernetes deployment and orchestration
- Service mesh implementation (Istio/Linkerd)
- Serverless function integration
- Cloud provider managed services adoption

9.1.2. Advanced Capabilities

- Event sourcing for audit and replay capabilities
- CQRS pattern for read/write separation
- GraphQL API layer for flexible data access
- Machine learning integration for intelligent routing

9.2. Scalability Roadmap

9.2.1. Infrastructure Scaling

- Multi-region deployment capabilities
- CDN integration for global content delivery
- Database sharding and replication strategies
- Caching layer implementation (Redis/Hazelcast)

9.2.2. Operational Maturity

- GitOps-based deployment pipelines
- Chaos engineering and resilience testing
- Advanced monitoring with Prometheus/Grafana

- Automated security scanning and compliance checking

10. Conclusion

The GlobalBooks SOA implementation successfully demonstrates the transition from monolithic to service-oriented architecture while maintaining enterprise integration requirements. The solution balances modern microservices patterns with traditional SOA principles, providing a robust foundation for scalable e-commerce operations.

Key achievements include comprehensive protocol support (REST and SOAP), secure authentication mechanisms, event-driven integration, and extensive testing coverage. The architecture supports both current business requirements and future scalability demands through careful service decomposition and technology selection.

The implementation serves as a practical reference for SOA principles application in contemporary software systems, demonstrating how traditional enterprise architecture patterns can be modernized for cloud-native environments while preserving backward compatibility and integration capabilities.

11. Appendices

11.1. Appendix A: Service API Reference

Complete API documentation is available in the companion document `API-ENDPOINTS.md`, which provides:

- Detailed endpoint specifications for all REST and SOAP services
- Request/response examples with sample data
- Authentication and authorization requirements
- Error codes and troubleshooting guides

11.2. Appendix B: Configuration Templates

11.2.1. Spring Boot Application Configuration

```
# Example: catalog-service application.properties
server.port=8085
spring.application.name=catalog-service

# Database Configuration
spring.datasource.url=jdbc:h2:mem:catalogdb
spring.jpa.hibernate.ddl-auto=create-drop

# RabbitMQ Configuration
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest

# SOAP Web Service Configuration
spring.ws.wsdl.location=classpath:wsdl/books.wsdl
spring.ws.path=/ws
```

11.2.2. Docker Compose Infrastructure

```
version: '3.8'
services:
  rabbitmq:
    image: rabbitmq:3.9-management
    ports:
      - "5672:5672"
      - "15672:15672"
    environment:
      RABBITMQ_DEFAULT_USER: guest
      RABBITMQ_DEFAULT_PASS: guest
    volumes:
      - rabbitmq_data:/var/lib/rabbitmq

volumes:
  rabbitmq_data:
```

11.3. Appendix C: Testing Scripts

11.3.1. User Authentication Test

```bash
#!/bin/bash
# Test user registration and JWT authentication

echo "Testing User Authentication Flow..."

# Register new user
curl -X POST http://localhost:8081/register \
  -H "Content-Type: application/json" \
  -d '{"username": "chamal1120", "password": "password"}'

# Authenticate and extract JWT token
JWT_RESPONSE=$(curl -s -X POST http://localhost:8081/authenticate \
  -H "Content-Type: application/json" \
  -d '{"username": "chamal1120", "password": "password"}')

echo "Authentication successful: $JWT_RESPONSE"
```

11.3.2. SOAP Service Test

```bash
#!/bin/bash
# Test SOAP catalog service functionality

echo "Testing SOAP Catalog Service..."

curl -X POST http://localhost:8085/ws \
  -H "Content-Type: text/xml; charset=utf-8" \
  -H "SOAPAction: \"\"" \
  -d '<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:cat="http://catalog.globalbooks.com/">
   <soap:Body>
      <cat:getBookRequest>
         <cat:isbn>978-0134685991</cat:isbn>
      </cat:getBookRequest>
   </soap:Body>
</soap:Envelope>'
```