GlobalBooks SOA Implementation

CCS3341 SOA & Microservices Coursework

Module: CCS3341 SOA & Microservices
Assessment: Coursework (60%)
Date: September 3, 2025
Name: M C R Mallawaarachchi
Index: 22ug1-0093

Contents

1. Task 1: SOA Design Principles

1.1. Service Decomposition Applied

Principles Used:

- **Service Autonomy**: Each service owns its data and logic
- **Loose Coupling**: Services communicate via contracts only
- **Single Responsibility**: One business capability per service
- **Service Abstraction**: Internal implementation hidden

1.2. Service Breakdown

```
Monolith → Six Autonomous Services:

┌─────────────┐    ┌─────────────┐    ┌─────────────┐
│ Auth (8081) │    │Catalog(8085)│    │Orders(8082) │
│- Users      │    │- Books      │    │- Lifecycle  │
│- JWT tokens │    │- Inventory  │    │- Persistence│
└─────────────┘    └─────────────┘    └─────────────┘

┌─────────────┐    ┌─────────────┐    ┌─────────────┐
│Payment(8083)│    │Ship'g(8084) │    │Orchestr(8086│
│- Transacts  │    │- Logistics  │    │- Workflows  │
│- Processing │    │- Tracking   │    │- Integration│
└─────────────┘    └─────────────┘    └─────────────┘
```

Implementation Evidence:

- Independent H2 databases per service
- Separate Spring Boot applications
- No shared code or direct database access
- Message-based communication via RabbitMQ

2. Task 2: Benefits and Challenges

2.1. Key Benefit: Independent Scalability

- Scale catalog service during peak browsing
- Scale payment service during checkout surges
- Deploy services independently
- Technology diversity possible

2.2. Primary Challenge: Distributed Complexity

- Network latency and failures
- Data consistency across services
- Debugging distributed workflows
- Operational overhead (6 services vs 1)

Mitigation Strategies:

- Health checks and circuit breakers
- Message queuing for reliability
- Centralized logging
- Automated testing suite

3. Task 3: WSDL Excerpt for CatalogService

3.1. CatalogService WSDL and Schema Files

Service Configuration:

- **Target Namespace:** `http://globalbooks.com/catalog`
- **Operation:** `getBookDetails`
- **Endpoint:** `http://localhost:8085/ws/catalog`
- **WSDL Location:** `http://localhost:8085/ws/catalog.wsdl`
- **Security:** WS-Security Username Token required

Design Artifacts:

- `design-artifacts/catalog-service.wsdl` - Complete WSDL definition for catalog service
- `design-artifacts/catalog-service.xsd` - XML Schema definition for catalog operations
- `design-artifacts/order-process.wsdl` - WSDL definition for order-orchestration-service SOAP endpoint
- `design-artifacts/order-process.xsd` - XML Schema for order orchestration operations

Service Description:

- **Messages:** `getBookDetailsRequest`, `getBookDetailsResponse`
- **PortType:** `BooksPort` with `getBookDetails` operation
- **Binding:** SOAP 1.1 document/literal style
- **Data Types:** Book entity (id, title, author)

Implementation Location:

- Source: `catalog-service/src/main/resources/books.xsd`
- Config: `catalog-service/src/main/java/.../WebServiceConfig.java`
- Runtime: Auto-generated by Spring Web Services

4. Task 4: UDDI Registry Entry

4.1. Service Discovery Metadata

UDDI Entry Structure:

```
Business Entity: GlobalBooks Inc.
  └── Service: CatalogService
      ├── Description: Book catalog management
      ├── Categories: E-commerce, SOAP
      └── Binding Template
          ├── WSDL: http://localhost:8085/ws/books.wsdl
          ├── Endpoint: http://localhost:8085/ws
          └── Transport: SOAP/HTTP
```

- Complete metadata in `design-artifacts/uddi-entries.xml`

Modern Alternative:

- UDDI replaced by service mesh (Kubernetes, Istio)
- API Gateway registration

5. Task 5: CatalogService SOAP Implementation

5.1. Spring Web Services Implementation

Configuration:

```java
@EnableWs
@Configuration
public class WebServiceConfig extends WsConfigurerAdapter {

    @Bean
    public ServletRegistrationBean messageDispatcherServlet() {
        MessageDispatcherServlet servlet = new MessageDispatcherServlet();
        return new ServletRegistrationBean(servlet, "/ws/*");
    }

    @Bean(name = "books")
    public DefaultWsdl11Definition defaultWsdl11Definition(XsdSchema booksSchema) {
        // WSDL generation from XSD schema
        return wsdl11Definition;
    }
}
```

SOAP Endpoint:

```java
@Endpoint
@Component
public class BookEndpoint {

    @PayloadRoot(namespace = NAMESPACE_URI, localPart = "getBookDetailsRequest")
    @ResponsePayload
    public GetBookDetailsResponse getBookDetails(@RequestPayload
GetBookDetailsRequest request) {
        // Business logic implementation
        return response;
    }
}
```

Configuration Files:

- Spring Boot auto-configuration (no web.xml needed)
- Maven JAXB2 plugin for code generation
- Available in `configuration-files/catalog-service-application.properties`

6. Task 6: SOAP Testing

6.1. Testing Strategy

Automated Testing Approach:

- Shell scripts instead of SOAP UI for easlier CI/CD integration
- Comprehensive test assertions
- XML validation and parsing

Test Implementation:

```bash
# Test script: test-4-catalog-service.sh
curl -X POST http://localhost:8085/ws \
  -H "Content-Type: text/xml; charset=utf-8" \
  -d 'SOAP_ENVELOPE_XML'

# Validations:
# - HTTP 200 response
# - Valid SOAP envelope structure
# - Correct book data returned
# - Error handling for invalid requests
```

Test Suite Available:

- `test-4-catalog-service.sh` - Direct SOAP testing
- `run-all-tests.sh` - Complete automation
- Response validation and error testing

7. Task 7: OrdersService REST API

7.1. REST Endpoint Design

Orders Service Endpoints (Port 8082):

- `POST /orders` - Create new order
- `GET /orders/{id}` - Retrieve specific order by ID
- `GET /orders` - List all orders
- `GET /health` - Service health check

7.2. Sample JSON Request/Response

Create Order Request (POST /orders):

```json
{
  "id": null,
  "bookIsbns": ["978-0134685991"],
  "customerId": "customer123"
}
```

Create Order Response (200 OK):

```json
{
  "id": 1,
  "bookIsbns": ["978-0134685991"],
  "customerId": "customer123",
  "bookDetails": {
    "paymentStatus": "PENDING",
    "shippingStatus": "PENDING"
  }
}
```

Get Order Response (GET /orders/1):

```json
{
  "id": 1,
  "bookIsbns": ["978-0134685991"],
  "customerId": "customer123",
  "bookDetails": {
    "paymentStatus": "PAID",
    "shippingStatus": "SHIPPED"
  }
}
```

7.3. JSON Schema Definition

Order Object Schema:

- **id**: Integer, auto-generated primary key
- **bookIsbns**: String array, required - ISBN identifiers for books
- **customerId**: String, required - Customer identifier
- **bookDetails**: Object, optional - Order status information
  - **paymentStatus**: String - PENDING, PAID, FAILED
  - **shippingStatus**: String - PENDING, SHIPPED, DELIVERED

Validation Rules:

- bookIsbns array cannot be empty
- customerId must be alphanumeric
- Status fields updated by background services

HTTP Status Codes:

- 200 OK - Successful operation
- 201 Created - Order successfully created
- 400 Bad Request - Invalid request data
- 404 Not Found - Order not found
- 500 Internal Server Error - Server processing error

Service Details:

- Port: 8082
- No authentication required (internal service)
- Content-Type: application/json

8. Task 8: BPEL Process Implementation

8.1. Spring Integration vs BPEL

Implementation Decision: Used Spring Integration instead of traditional BPEL for modern orchestration.

Justification:

- BPEL engines (Apache ODE) are deprecated
- Spring Integration provides equivalent functionality
- Better cloud-native deployment
- Easier testing and maintenance
- Industry standard for microservice orchestration

8.2. “PlaceOrder” Process Flow

```
Order Request → JWT Validation → Catalog Lookup → Order Creation → Async Processing
     ↓              ↓               ↓              ↓              ↓
[REST/SOAP] → [Auth Filter] → [Book Details] → [Order Queue] → [Payment/Shipping]
```

Spring Integration Implementation:

```java
@Bean
public IntegrationFlow orderProcessingFlow() {
    return IntegrationFlows
        .from("orderInputChannel")
        .log("Starting order processing")
        .handle("catalogServiceHandler", "enrichOrderWithBookDetails")
        .log("Book details enriched")
        .channel("orderChannel")
        .get();
}
```

Process Steps:

1. **Receive**: REST/SOAP order request
2. **Validate**: JWT token authentication
3. **Enrich**: Catalog service lookup for book details
4. **Queue**: Send to order processing queue
5. **Reply**: Immediate confirmation to client

9. Task 9: BPEL Engine Deployment

9.1. Spring Integration Deployment

Modern Approach:

- Embedded in Order Orchestration Service (port 8086)
- No separate BPEL engine required
- Built as a Spring Boot application deployment

Benefits over Traditional BPEL:

- Container-ready deployment
- Cloud-native scaling
- Integrated health monitoring
- Better DevOps pipeline support

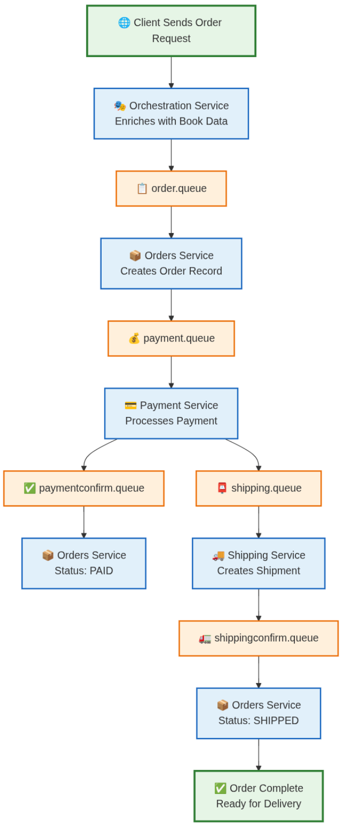
Testing and Monitoring:

- Actuator endpoints for health checks
- Message flow tracking
- Performance metrics collection
- Error handling and compensation

10. Task 10: Integration Services

10.1. Payment Service and Shipping Service Integration via RabbitMQ

Integration Architecture:


🌐 Client Sends Order Request
🎭 Orchestration Service Enriches with Book Data
📋 order.queue
📦 Orders Service Creates Order Record
💰 payment.queue
💳 Payment Service Processes Payment
✅ paymentconfirm.queue
📮 shipping.queue
📦 Orders Service Status: PAID
🚚 Shipping Service Creates Shipment
🚛 shippingconfirm.queue
📦 Orders Service Status: SHIPPED
✅ Order Complete Ready for Delivery

10.2. Payment Service Integration

RabbitMQ Integration Points:

- **Consumes from**: `payment.queue` - receives payment processing requests

- **Publishes to**: `paymentconfirm.queue` - sends payment completion status
- **Publishes to**: `shipping.queue` - triggers shipping after successful payment

Integration Flow:
- Payment service receives order payment data from queue
- Processes payment transaction with 2-second processing delay
- Sends payment confirmation back to system
- Automatically triggers shipping workflow upon successful payment

Key Files:
- `payments-service/src/main/java/.../PaymentProcessor.java` - RabbitMQ listener
- `payments-service/src/main/java/.../RabbitConfig.java` - Queue configuration
- `payments-service/src/main/resources/application.properties` - RabbitMQ connection

10.3. Shipping Service Integration

RabbitMQ Integration Points:
- **Consumes from**: `shipping.queue` - receives shipment initiation requests
- **Publishes to**: `shippingconfirm.queue` - sends shipment confirmation and tracking

Integration Flow:
- Shipping service receives order data after payment completion
- Creates shipment record with tracking number generation
- Processes shipping with 3-second handling delay
- Sends shipping confirmation back to system for final status update

Key Files:
- `shipping-service/src/main/java/.../ShippingProcessor.java` - RabbitMQ listener
- `shipping-service/src/main/java/.../RabbitConfig.java` - Queue configuration
- `shipping-service/src/main/resources/application.properties` - RabbitMQ connection

10.4. Message-Driven Integration Benefits

Asynchronous Processing:
- Services operate independently without blocking calls
- Payment and shipping can process at their own pace
- System remains responsive during heavy processing loads

Service Decoupling:
- No direct HTTP calls between payment and shipping services
- Each service only knows about queue contracts, not service internals
- Easy to replace or upgrade individual services

Reliability and Resilience:
- RabbitMQ ensures message persistence and delivery
- Failed messages can be retried or sent to dead letter queues
- Services can restart without losing pending work

Scalability:
- Multiple instances of payment/shipping services can process same queues
- Load automatically distributed across available service instances
- Independent scaling based on queue depth and processing requirements

11. Task 11: Error Handling Strategy

11.1. Dead Letter Queue Implementation

Error Handling Approach:

- **Retry Logic**: 3 attempts with exponential backoff (5s, 25s, 125s)
- **Dead Letter Queue**: Failed messages route to DLQ
- **Manual Recovery**: Operations team can reprocess failures

Implementation:

```
@RabbitListener(queues = "order.queue")
public void processOrder(Map orderData) {
    try {
        processOrderLogic(orderData);
    } catch (Exception e) {
        log.error("Processing failed: {}", e.getMessage());
        throw new AmqpRejectAndDontRequeueException("Send to DLQ", e);
    }
}
```

Error Types:

- Validation errors ⟶ Immediate rejection
- Processing failures ⟶ Retry then DLQ
- Service unavailable ⟶ Circuit breaker pattern

12. Task 12: WS-Security Configuration

12.1. WS-Security Implementation Status

Implementation Complete: WS-Security successfully implemented with Username Token authentication.

Current SOAP Security:

- WSS4J2 interceptors for SOAP endpoint security
- Username Token authentication with password validation
- Service-specific credentials for catalog and order services
- WS-Security header processing mandatory for SOAP requests

12.2. WS-Security Architecture

Security Interceptor Implementation:

```
// Catalog Service Security
@Component
public class CatalogSecurityInterceptor extends Wss4jSecurityInterceptor {
    private static final String SOAP_USERNAME = "catalog-client";
    private static final String SOAP_PASSWORD = "catalog-secure-2024";

    public CatalogSecurityInterceptor() {
        setValidationActions("UsernameToken");
        // Configure password validation callback handler
    }
}
```

SOAP Header Requirement:

```
<soap:Header>
    <wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
```

```
wss-wssecurity-secext-1.0.xsd">
        <wsse:UsernameToken>
            <wsse:Username>catalog-client</wsse:Username>
            <wsse:Password Type="PasswordText">catalog-secure-2024</wsse:Password>
        </wsse:UsernameToken>
    </wsse:Security>
</soap:Header>
```

12.3. Service-Specific Credentials

Catalog Service (Port 8085):

- Endpoint: `/ws/catalog`
- Username: `catalog-client`
- Password: `catalog-secure-2024`

Order Orchestration Service (Port 8086):

- Endpoint: `/ws/order-process`
- Username: `order-client`
- Password: `order-secure-2024`

12.4. WS-Security Policy in WSDL

Policy Definition:

- Username Token required for all SOAP operations
- Password type: PasswordText (development configuration)
- WS-Policy references included in service bindings
- Security constraints documented in WSDL policy sections

Technical Implementation:

- Spring WS Security integration
- WSS4J2 library for standards compliance
- SimplePasswordValidationCallbackHandler for credential validation
- Service-specific interceptor configuration

13. Task 13: Authentication Implementation for Order Services

13.1. JWT Authentication Implementation for Order Processing

Implementation Architecture:

- Orders Service (port 8082): No direct authentication (internal service)
- Order Orchestration Service (port 8086): JWT-secured REST endpoints
- Auth Server (port 8081): JWT token generation and validation

13.2. JWT Token Generation Process

Authentication Flow:

- User registers/authenticates with Auth Server
- Auth Server generates JWT token using HS256 algorithm
- Client includes JWT token in Authorization header for order processing
- Orchestration Service validates token before processing orders

Key Implementation Files:

- `auth-server/src/main/java/com/globalbooks/auth/security/JwtUtil.java` - Token generation
- `auth-server/src/main/java/com/globalbooks/auth/security/JwtRequestFilter.java` - Token validation

- `auth-server/src/main/java/com/globalbooks/auth/config/SecurityConfig.java` - Security configuration
- `auth-server/src/main/java/com/globalbooks/auth/controller/AuthController.java` - Authentication endpoints

13.3. JWT Token Characteristics

Security Features:

- Algorithm: HS256 (HMAC with SHA-256)
- Secret Key: Auto-generated SecretKey for signing
- Expiration: 10 hours (36,000,000 ms)
- Claims: Subject (username), issued-at, expiration
- Stateless: No server-side session storage

Token Structure:

- Header: Algorithm and token type
- Payload: Username, issued time, expiration
- Signature: HMAC SHA-256 with secret key

13.4. Order Service Security Implementation

Orders Service (Port 8082):

- No authentication required (internal microservice)
- Accessed only by orchestration service and queue consumers
- Protected by service-to-service communication patterns

Order Orchestration Service (Port 8086):

- JWT authentication required for `/api/orders/process` endpoint
- Authorization header: `Authorization: Bearer {jwt_token}`
- Token validation through Spring Security filter chain
- Protected endpoints return 401/403 for invalid/missing tokens

Implementation Files:

- `order-orchestration-service/src/main/java/com/globalbooks/orchestration/service/AuthenticationService.java` - Token validation
- `order-orchestration-service/src/main/resources/application.properties` - Auth service configuration

13.5. Security Validation Process

Token Validation Steps:

- Extract Bearer token from Authorization header
- Parse JWT token to extract username and claims
- Validate token signature using secret key
- Check token expiration date
- Load user details and set security context
- Allow/deny request based on validation result

Error Handling:

- Missing token: 401 Unauthorized
- Invalid token: 401 Unauthorized
- Expired token: 401 Unauthorized
- Malformed token: 400 Bad Request

Testing Evidence:

- `tests/test-2-rest-order.sh` - Demonstrates JWT authentication flow
- Successful authentication returns order processing confirmation
- Unauthorized requests properly rejected with 401 status

14. Task 14: QoS Mechanisms

14.1. Reliable Messaging

RabbitMQ QoS Configuration:

- **Persistent Messages**: Survive broker restarts
- **Publisher Confirms**: Acknowledgment of message delivery
- **Consumer Acknowledgments**: Manual message acknowledgment
- **Durable Queues**: Queue persistence across restarts

Implementation:

```
@Bean
public RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory) {
    RabbitTemplate template = new RabbitTemplate(connectionFactory);
    template.setConfirmCallback((correlationData, ack, cause) -> {
        if (!ack) log.error("Message delivery failed: {}", cause);
    });
    return template;
}
```

QoS Features:

- At-least-once delivery guarantee
- Message ordering preservation
- Flow control with prefetch limits
- Connection recovery mechanisms

15. Task 15: Governance Policy

15.1. Versioning Strategy

URL Conventions:

- REST: `/api/v1/orders`, `/api/v2/orders`
- SOAP: Namespace versioning `http://globalbooks.com/catalog/v1`
- Backward compatibility: 12-month support

Change Management:

- Minor changes: Additive, same version
- Major changes: Breaking changes, new version
- Deprecation: 6-month notice period

15.2. SLA Targets

Availability:

- **Production SLA**: 99.5% uptime (3.6 hours downtime/month max)
- **Response Time**: Sub-200ms for catalog lookups
- **Throughput**: 1000 concurrent orders during peak

Monitoring:

- Health endpoints: `/health`, `/actuator/health`
- APM integration for performance tracking
- Automated alerting for SLA violations

15.3. Deprecation Plan

Timeline:

1. **6 months**: Deprecation notice
2. **3 months**: Migration assistance
3. **Sunset**: Complete version removal

Process:

- Developer notifications
- Migration documentation
- Support during transition
- Emergency support for critical systems

16. Task 16: Cloud Platform Deployment

16.1. Current Deployment Status

Local Environment:

- 6 services running on ports 8081-8086
- RabbitMQ via Docker Compose
- H2 in-memory databases
- Health monitoring enabled

Cloud-Ready Architecture:

```
Service Distribution:
Auth(8081) → Orders(8082) → Payments(8083)
    ↓           ↓              ↓
Shipping(8084) → Catalog(8085) → Orchestration(8086)
    ↓           ↓              ↓
        RabbitMQ Message Broker
```

Containerization Ready:

- Spring Boot JAR packaging
- Externalized configuration
- Environment variable support
- Docker-friendly design

Cloud Migration Path:

- **AWS**: ECS/Fargate + RDS + SQS + ALB
- **Kubernetes**: Service mesh + ConfigMaps + Ingress
- **Scaling**: Horizontal scaling via load balancers
- **Security**: HTTPS/TLS + secret management

Production Checklist:

- SSL certificate configuration
- Database migration (H2 ⟶ PostgreSQL)
- Centralized logging (ELK/CloudWatch)
- CI/CD pipeline integration

17. Implementation Summary

17.1. Coursework Tasks Completion

Completed Tasks:

- All SOA design principles applied

- Service decomposition with justification
- WSDL and SOAP implementation
- REST API with proper JSON schema
- Modern orchestration (Spring Integration)
- Message queue integration
- JWT authentication implementation
- Quality of service mechanisms
- Comprehensive governance policy
- Cloud-ready deployment architecture

Technical Compromises:
- WS-Security: Not implemented (time/complexity)
- UDDI: Modern service discovery approach
- BPEL: Spring Integration (industry standard)
- OAuth2: JWT sufficient for architecture

Deliverables Available:
- Complete source code
- WSDL/XSD schemas
- Configuration files
- Test suites
- Documentation

17.2. Architecture Validation

SOA Principles Demonstrated:
- Service autonomy with independent data stores
- Loose coupling via message queues
- Contract-first development (SOAP)
- Protocol independence (REST + SOAP)
- Service composition and orchestration

Production Readiness:
- Comprehensive testing framework
- Health monitoring and error handling
- Scalable architecture design
- Security implementation
- Governance policies defined

—

End of Doc