SwiftLogistics Middleware - Architecture Documentation

Service Explanations

1. **API Gateway** (`services/api-gateway`)

Port: 3000
Role: Single entry point for all external traffic

What it does:

- **Authenticates** requests using JWT (JSON Web Tokens)
- **Rate limits** incoming requests (prevention against DDoS attacks)
- **Validates** incoming order data using express-validator
- **Routes** requests to appropriate backend services
  - `/api/orders` → Orchestrator Service
  - `/api/driver` → Driver/Manifest endpoints
- Acts as a **security shield** - hides internal architecture from clients

Key Files:

- `index.js` - Express server setup
- `middleware/auth.js` - JWT validation
- `middleware/rateLimiter.js` - Rate limiting (100 req/15min)
- `routes/orders.js` - Order submission/retrieval
- `routes/driver.js` - Driver-specific routes

---

2. **Orchestrator Service** (`services/orchestrator`)

Port: 3001
Role: Transaction manager and order lifecycle coordinator

What it does:

- **Receives** validated orders from API Gateway
- **Persists** order to MongoDB immediately (Status: RECEIVED)
- **Publishes** order to RabbitMQ message queue
- **Returns** 202 Accepted response immediately (non-blocking)
- **Tracks** status of all integration points (CMS, ROS, WMS)
- **Updates** order status as adapters complete their tasks

Key Files:

- `index.js` - Express server + MongoDB connection
- `models/Order.js` - MongoDB schema definition
- `services/messageQueue.js` - RabbitMQ publisher

- `services/orderService.js` - Business logic (create, update, fetch)
- `routes/orders.js` - REST endpoints

Data Flow:

```
Client → Gateway → Orchestrator → MongoDB (save)
                                → RabbitMQ (publish)
                                ← 202 Accepted
```

---

3. **Notification Service** (`services/notification-service`)

Port: 3002
Role: Real-time event broadcaster via WebSockets

What it does:

- **Maintains** persistent WebSocket connections with clients
- **Listens** to RabbitMQ events exchange
- **Broadcasts** real-time notifications to connected clients
  - Order received
  - Processing updates
  - Delivery status changes
- Enables **live tracking** without polling

Key Files:

- `index.js` - HTTP + Socket.io server
- `services/socketManager.js` - WebSocket connection manager
- `services/eventConsumer.js` - RabbitMQ event listener

Example Notification:

```
{
  type: "ORDER_RECEIVED",
  orderId: "ORD-123",
  timestamp: "2026-01-25T10:30:00Z",
  data: { status: "RECEIVED" }
}
```

Integration Adapters (Protocol Translators)

4. **CMS Adapter** (`services/adapters/cms-adapter`)

Protocol: SOAP/XML
Integrates with: Legacy Content Management System

What it does:

- **Subscribes** to `new_order_queue` on RabbitMQ
- **Transforms** JSON order → XML envelope
- **Sends** SOAP request to CMS Mock (Port 4000)
- **Updates** Orchestrator: `cmsStatus: SUCCESS/FAILED`

Key Files:

- `services/queueConsumer.js` - RabbitMQ listener
- `services/soapClient.js` - SOAP client using `soap` library
- `utils/jsonToXml.js` - JSON to XML transformer

Why it exists: CMS is a legacy system that only understands XML/SOAP, not modern JSON.

---

5. **ROS Adapter** (`services/adapters/ros-adapter`)

Protocol: REST/JSON
Integrates with: Route Optimization Service

What it does:

- **Subscribes** to `new_order_queue` on RabbitMQ
- **Extracts** pickup and delivery addresses
- **Calls** ROS REST API to get optimized route
- **Stores** route data back in Orchestrator
- **Updates** Orchestrator: `rosStatus: SUCCESS/FAILED`

Key Files:

- `services/queueConsumer.js` - RabbitMQ listener
- `services/restClient.js` - Axios-based REST client

Response Example:

```
{
  "route": [{"lat": 6.9271, "lng": 79.8612}],
  "distance": 25,
  "duration": 45
}
```

---

6. **WMS Adapter** (`services/adapters/wms-adapter`)

Protocol: Raw TCP Sockets
Integrates with: Warehouse Management System

What it does:

- **Subscribes** to `new_order_queue` on RabbitMQ

- **Opens** TCP socket connection to WMS (Port 4002)
- **Sends** binary/delimited packet with order data
- **Receives** warehouse confirmation (shelf location, pick time)
- **Updates** Orchestrator: `wmsStatus: SUCCESS/FAILED`

Key Files:

- `services/queueConsumer.js` - RabbitMQ listener
- `services/tcpClient.js` - Native Node.js `net` module

Why it exists: WMS is an old system using proprietary TCP protocol, not HTTP.

Mock Systems (For Testing)

7. **CMS Mock** (`services/mocks/cms-mock`)

Port: 4000
Type: SOAP Server

Simulates the legacy Content Management System. Provides WSDL definition and responds to SOAP requests.

8. **ROS Mock** (`services/mocks/ros-mock`)

Port: 4001
Type: REST API

Simulates the Route Optimization Service. Returns mock optimized routes with distance/duration.

9. **WMS Mock** (`services/mocks/wms-mock`)

Port: 4002
Type: TCP Server

Simulates the Warehouse Management System. Accepts TCP connections and returns warehouse confirmations.

Shared Utilities

`shared/utils/logger.js`

Winston-based logging utility for consistent log formatting across all services.

`shared/utils/constants.js`

Centralized constants:

- Order status enums
- Event types
- RabbitMQ queue/exchange names

- HTTP status codes

Architectural Patterns Used

Pattern	Where	Why
API Gateway	Entry point	Hide backend complexity, centralized auth
Pub/Sub	RabbitMQ	Decouple services, async processing
Adapter	CMS/ROS/WMS	Translate protocols to unified format
Event-Driven	Entire system	Non-blocking, scalable architecture

---

Message Flow: Order Submission

```
1. Client submits order (JSON) → API Gateway
2. Gateway validates JWT and data
3. Gateway forwards to Orchestrator
4. Orchestrator saves to MongoDB (Status: RECEIVED)
5. Orchestrator publishes to RabbitMQ
6. Orchestrator returns 202 Accepted to client

--- Parallel Processing (all adapters run simultaneously) ---

7a. CMS Adapter picks up message → Calls CMS SOAP → Updates status
7b. ROS Adapter picks up message → Calls ROS REST → Stores route
7c. WMS Adapter picks up message → Calls WMS TCP → Updates status

8. Each adapter publishes completion event to RabbitMQ
9. Notification Service consumes events
10. Notification Service broadcasts to client via WebSocket
11. Client sees real-time status update
```

Key Benefit: Client gets immediate acknowledgment, system processes in background, client receives updates in real-time!